PLANNING-BASED APPROACH FOR AUTOMATING SEQUENCE DIAGRAM GENERATION

BY

YASER ALI DIAB SULAIMAN

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

DECEMBER 2012

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **YASER ALI DIAB SULAIMAN** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

Dr. Moataz A. Ahmed (Adviser)

Dr. Mohammad R. Alshayeb (Member)

Dr. Lahouari M. Ghouti (Member)

Dr. Adel F. Ahmed
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

28/5/13
Date

In memory of my cousin Mohammed Nihad
You will forever live young in our hearts and minds

ACKNOWLEDGMENTS

I would have never made it to the end of this journey without the help and support of so many people. Prophet Muhammad (peace be upon him) said: "He who does not thank people does not thank God." This is my humble attempt to say thank you.

Thank you, Dr. Moataz Ahmed, for everything. Thank you for your immense support, continued encouragement, compassionate understanding, and for continually trying to cure me from my two terminal illnesses: perfectionism and procrastination. To say that I was greatly fortunate to work under your supervision is a gross understatement.

Thank you, Dr. Mohammad Alshayeb and Dr. Lahouari Ghouti, for your valuable feedback, kind support, and tremendous help. I could not have wished for better Committee Members.

Thank you, Dr. Muhammad Alsuwaiyel, for giving me a second chance. I would not be writing these words if it was not for you.

Thank you, Mom, Dad, Maher, and Basem for your unconditional love, care, and support.

Thank you, uncle Adib, for relentlessly pushing me out of my comfort zone, for teaching me how to read and write between the lines, and, above all, for being

unorthodox.

Thank you, Abdul Karim, Abdullah, Bader, Ibrahim, Iyas, Riyadh, Saad, Salamah, Tamim, and all my KFUPM friends for every serious discussion, hysterical laugh, and for all the great experiences and memories. I wish you the best of luck and success in your endeavors.

Finally, thank you, Deanship of Scientific Research at King Fahd University of Petroleum and Minerals (KFUPM), for supporting this research under Research Grant IN111013.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME: Yaser Ali Diab Sulaiman

TITLE OF STUDY: Planning-Based Approach for Automating Sequence Diagram Generation

MAJOR FIELD: Computer Science

DATE OF DEGREE: December 2012

During requirement elicitation, the consistency of UML (Unified Modeling Language) use cases against (the independently-developed) class diagrams can be analyzed by trying to develop the sequence diagrams based on those models. But as the complexity of the system being modeled increases, generating the sequence diagrams manually becomes harder. Sequence diagram generation can be automated by treating it as a planning problem and solving it using an automated planning technique. Using such a technique requires expressing goals and actions with their preconditions and postconditions, which is indeed the case when the Design by Contract (DbC) approach is used in developing the models. Based on this similarity, this thesis presents and empirically evaluates a framework for treating the core activity of sequence diagram generation (i.e. determining the sequence of message

passes) as a planning problem and solving it as such. With the increasing support for DbC in modeling tools and programming frameworks, this approach should help in improving the software development process by enabling automatic consistency analysis of use cases against class diagrams through automatic sequence diagram generation.

ملخص الرسالة

الاسم الكامل: ياسر علي دياب سليمان

عنوان الرسالة: مقاربة مبنية على التخطيط لأتمتة توليد المخططات التسلسلية

التخصص: علوم الحاسب الآلي

تاريخ الدرجة العلمية: صفر 1434 هـ

خلال استنباط المتطلبات، يمكن تحليل الاتساق بين حالات الاستخدام (Use Cases) المكتوبة بلغة النمذجة الموحدة (Unified Modeling Language) مع مخططات الأصناف (Class Diagrams) التي طورت بشكل مستقل والمكتوبة بنفس اللغة عن طريق محاولة تطوير المخططات التسلسلية (Sequence Diagrams) بناءً على النماذج السابقة. لكن كلما ازداد تعقيد النظام المراد تصميمه، كلما ازدادت صعوبة إنشاء المخططات التسلسلية يدويًا. من الممكن أتمتة عملية توليد المخططات التسلسلية عن طريق التعامل معها على أنها مشكلة تخطيط وحلها باستخدام خوارزميات التخطيط الآلي. استخدام أسلوب كهذا يتطلب التعبير عن الأهداف والأفعال مع شروطها المسبقة واللاحقة، كما هو الحال بالفعل عندما يتم استخدام مقاربة التصميم بالتعاقد (Design by Contract) عند تطوير النماذج أعلاه. بناء على هذا التشابه، فإن هذه الرسالة تقدّم وتقيّم بالتجربة إطار عمل للتعامل مع النشاط الأساسي في عملية توليد المخططات التسلسلية (أي، تحديد تسلسل تبادلات الرسائل) على أنه مشكلة تخطيط وحلها على هذا النحو. مع الزيادة الحاصلة في دعم التصميم بالتعاقد في أدوات النمذجة وأطر البرمجة، ينبغي لهذه المقاربة أن تساعد في تحسين عملية تطوير البرمجيات من خلال تمكين التحليل التلقائي للاتساق بين حالات الاستخدام ومخططات الأصناف عن طريق توليد المخططات التسلسلية تلقائيا.

CHAPTER 1

INTRODUCTION

Since the 1970s, software engineers have come up with many—perhaps *too* many—software modeling languages and methodologies, some of which are discussed, or touched upon, in Ian Sommerville's *Software Engineering* [1]. These efforts led to the development of function-oriented methods such as Structured Analysis (SA), Structured Design (SD), and Jackson System Development (JSD). Additionally, object-oriented methods such as Booch, Object-Modeling Technique (OMT), and Object-Oriented Software Engineering (OOSE) were proposed. Eventually, many different approaches were incorporated into a single approach that is built around the Unified Modeling Language (UML), which is now a *de facto* standard for object-oriented modeling.

Different UML models are developed to represent different views of the same system: use cases capture the externally visible services of the system to be built, class diagrams express its static structure, and sequence diagrams represent the dynamic interactions within it. In other words, use cases model a functional view, class diagrams model a structural view, and sequence diagrams model a behavioral view of the system. It is worth noting here that the taxonomy of UML diagrams [2, p. 720] does not explicitly include functional diagrams; it recognizes only two major types: structure diagrams, which include class diagrams, and

behavior diagrams, which include use case and sequence diagrams. Nevertheless, use cases can indeed be thought of as functional diagrams. Juhani Iivari [3], citing Ivar Jacobson's *Object-Oriented Software Engineering*, notes that use cases define the functionality of the system and the specific way in which parts of it can be used.

Of course, this multi-view approach of UML can lead to inconsistencies between the different models, especially if they are developed independently, but detecting such inconsistencies is actually an objective of this approach. This can be done by cross-referencing the different models against each other and looking for system's aspects that were captured in some views but not in others. For instance, every use case should be realized by a sequence diagram. The sequence diagrams themselves should not contain classes and methods that do not exist in the class diagram. If some classes or methods remain unused even after realizing all use cases, then either they are redundant or there are missing use cases that were partially captured by the class diagram. The problem is that cross-referencing the different models manually can become a tedious and time-consuming process. So, it would be better if the consistency could be analyzed automatically.

In addition to the difficulty of manually analyzing the consistency of the models, the multi-view approach of UML modeling suffers from the difficulty of manually creating some of the models for large and complex systems. Sequence diagrams are one of those models that can be difficult to create because creating them manually is an error-prone process. The results of the experiments by Yue, Briand, and Labiche [4] support this observation. In their experiments, trained fourth year undergraduate students failed to create 50% of the required messages. Out of the created messages, 25% were inconsistent with the reference diagrams. Therefore, automating that process would be quite helpful and practical.

Automatic consistency analysis has received considerable attention from re-

searchers, but automatic model generation has not. This is apparent from the disparity between the number of the surveyed works that focus on each of these processes. Nevertheless, the recency of the works focusing on automatic model generation suggests that it is starting to get the attention of researchers. Also, based on the works surveyed so far, one can see that the two problems are rarely, if ever, addressed together. Therefore, an approach that focuses on automatic model generation while simultaneously addressing automatic consistency analysis is worth investigating, and it is worth investigating *now*.

The main idea of the proposed approach is to combine AI planning techniques and Design by Contract to automatically generate sequence diagrams from use cases and class diagrams and to automatically analyze their consistency. There is a striking similarity between planning and sequence diagram generation that arises when the other models, namely, use cases and class diagrams, are developed using Design by Contract. In addition, the support for Design by Contract in computer-aided software engineering (CASE) tools and programming frameworks is increasing. For example, Enterprise Architect 8 allows the user to specify preconditions and postconditions for individual methods in a class diagram, albeit as a somewhat simplistic supplement to the method's description. This opens up the possibility for the proposed work to be implemented as a plug-in for such a CASE tool. Also, support for code contracts is now a core feature of the .NET Framework 4 [5]. These observations suggest that more and more software developers and professionals are adopting Design by Contract as they realize its practical benefits. In short, AI planning techniques and Design by Contract seem to be suitable for solving the problems that the current work addresses.

The current work will focus on the models of the requirements phase. The main reason is that use cases, along with conceptual class diagrams, belong to that phase. Sequence diagrams are usually derived from these two types of models, and

then used in the design phase as a starting point for setting up the architecture and component designs.

Another reason for focusing on requirements models is that it is better to detect and fix defects as early as possible. Steve McConnell [6, p. 29] cites studies by researchers at Hewlett-Packard, IBM, and other organizations that show that fixing a defect during system testing or after releasing it costs 10 to 100 times more than when it is done by the beginning of construction. In other words, the cost of fixing a defect increases significantly as the difference between the time it is introduced and the time it is detected increases. As McConnell puts it, "the longer the defect stays in the software food chain, the more damage it causes further down the chain" [6, p. 29]. So in general, the earlier a defect is detected, the less cost it will result in. Requirement defects tend to be the most expensive to fix since requirements are done first. Thus, it is reasonable to focus verification and validation efforts on requirements.

1.1 Research Questions

The current work aims at answering four main research questions:

1. How can sequence diagrams be automatically generated form a set of use cases and a class diagram that were designed by contract?

2. How can this automatic process be used to analyze the consistency between the given use cases and the class diagram?

3. Which of the contract languages, or representations, should be used in the given use cases and the class diagram to enable the processes mentioned above?

4. How do the sequence diagrams that were automatically generated compare

to the ones that were manually generated?

1.2 Main Contributions

Since—to the best of the author's knowledge—there were no published works on applying automated planning to the problem of UML sequence diagram generation when Design by Contract is used to develop the use cases and class diagrams, the research efforts presented in this Thesis had two main objectives:

1. To show that the core activity of sequence diagram generation (that is, determining the sequence of message passes) can indeed be treated as a planning problem and solved as such.

2. To identify and point out the issues that need to be addressed and the problems that need to be solved to advance the application of automated planning to the domain of sequence diagrams and take it to the next level.

In essence, a new research avenue, which seemed to be largely unexplored, emerged. To achieve the first objective, the correspondence between automated planning and Design by Contract was relied upon and the conceptual model of state-transition systems used in automated planning was adapted to the problem of planning messages in sequence diagrams. Along the way, some of the differences between action planning and message-pass planning that makes adapting existing action planners to sequence diagrams difficult were discovered. Based on those differences, a software library for planning messages in sequence diagrams, Communiqué, was developed. Using Communiqué, it was empirically shown that even with a simple conceptual model of restricted state-transition systems, determining the sequence of message passes in sequence diagrams can be treated as a planning problem and can be solved using forward state-search space.

Since the research avenue was new and largely unexplored, the research presented in this Thesis raised more questions than it could answer. Therefore, the second objective of the research efforts was to draw the attention of other researchers in the field to this new area; to shed a light on what else needs to be done to advance the application of automated planning to sequence diagrams.

1.3 Organization of the Thesis

This Thesis is organized as follows. Chapter 2 presents the background necessary to understand and navigate the rest of the Thesis. Chapter 3 presents a survey of the relevant literature and evaluates related approaches. The main idea underlying this Thesis of using automated planning and Design by Contract to plan messages in sequence diagrams is explained and presented in Chapter 4. The same Chapter also discusses the details of Communiqué, the software library that we implemented for that purpose. Chapter 5 contains the details of the experiments that we carried out to evaluate the proposed approach and its implementation. Finally, Chapter 6 concludes the Thesis by discussing the contributions, limitations, and future work.

CHAPTER 2

BACKGROUND

This chapter provides the necessary background needed to navigate and understand the rest of this Thesis. It covers three core knowledge areas: UML Models (including Use Cases, Class Diagrams, and Sequence Diagrams), Design by Contract, and Automated Planning. A reader interested in knowing more about a particular topic under these three areas is encouraged to consult the relevant references used in this Chapter.

2.1 Modeling with UML

2.1.1 Use Cases and Scenarios

As Bruegge and Dutoit [7] note, the concept of use case was made popular by Jacobson et al. in *Object-Oriented Software Engineering: A Use Case Driven Approach* [8]. Use cases of a software system represent its functionality. Use cases describe the system's functionality by capturing the general sequences of interactions between actors, which are entities external to the system, and the system itself.

Since use cases are mostly used to communicate with the client and the users of the system, they are usually written in natural language. There is a number

of different use case templates because there is no standard way to describe use cases textually. Nevertheless, use case templates generally contain the following main fields:

- Use case name or title.
- Participating actors, including the actor that initiates the use case.
- Preconditions (or entry conditions [7]), which are the conditions that must be ensured before initiating the use case.
- Flow of events, which are the sequence of interactions of the use case.
- Postconditions (or exit conditions [7]), which are the conditions that will be true after the completion of the use case.

In UML, use cases are graphically represented in use case diagrams [2, p. 617]. In these diagrams, actors are represented with stick figures, use cases with ovals, and system boundary with a box containing the use cases. For example, Figure 2.1 depicts a use case diagram for a simple watch system [7]. The user (that is, the `WatchUser` actor) can either read the time on the watch (with the `ReadTime` use case) or set the time (with the `SetTime` use case).

While use cases are abstractions that describe general sequences of interactions, scenarios are instances of use cases that represent concrete sequences of actions. In other words, "a scenario is one specific path through a use case." [9, p.146].

2.1.2 Class Diagrams and Object Diagrams

The static structure of a software system can be described in terms of classes, attributes, operations, and associations. A class is an abstraction of a set of objects that share the same attributes, operations, and relationships. An object is a concrete instance of a class that is created, modified, and destroyed while the

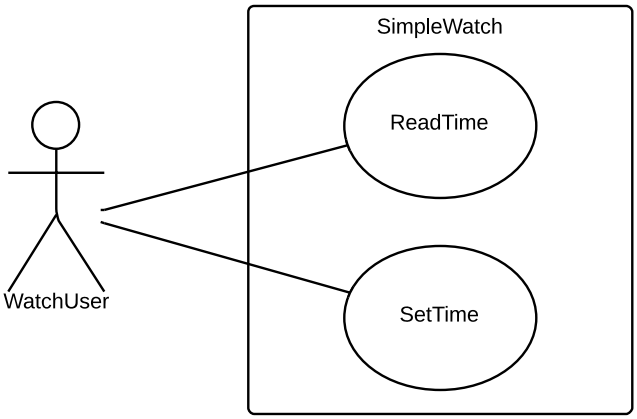

Figure 2.1: A use case diagram for a 2-button watch system.

system is executing. Objects encapsulate state—by having specific values for their attributes and links to other objects—and behavior—by having operations that they can perform. An association between two classes represents a relationship between them that allows their instances to have links between each other.

In UML, such a static structure is represented with a class diagram [2, p. 145]. Classes are depicted by boxes made of three compartments: the first is for the class name, the second is for the attributes, and the last is for the operations. Depending on the level of details needed in the current phase of development, the attributes and operations compartments can be suppressed. Associations between classes are depicted by lines joining the boxes representing the relevant classes. These associations can be unidirectional, in which case the line will have an arrow on one end only, or bidirectional, in which case the line will have arrows on both ends—or by convention, no arrows at all.

As an example of a class diagram, Figure 2.2 depicts part of the structure of the simple watch system [7], in which every `SimpleWatch` has two `PushButton`s,

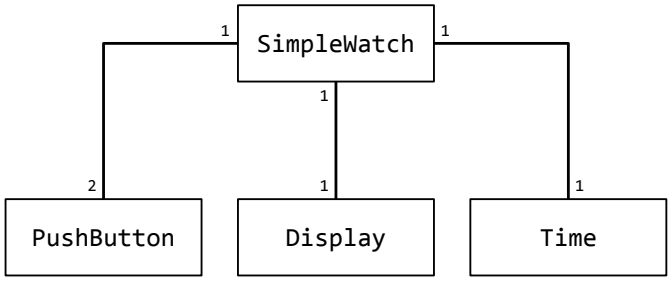

Figure 2.2: A class diagram for a simple watch system.

one `Display`, and one `Time`.

Object diagrams are basically concrete instances of class diagrams, i.e. they include only instances and no classes. In other words, while class diagrams depict classes and associations, object diagrams depict objects and links. As such, an object diagram represents (part of) the structure of the system at a specific point in time.

2.1.3 Sequence Diagrams

The dynamic behavior of a software system can be modeled and visualized using different kinds of interaction diagrams, which mainly describe the communication among sets of objects. Objects communicate by sending messages to each other. An object can request the execution of an operation on a different object by sending it a message, which is composed of a name and arguments (if any). When an object receives a message, it matches the message up with to one of its operations, invokes the operation, and passes it the arguments. It is worth noting here that the semantics of messages and message passing presented here are based upon the discussion of Bruegge and Dutoit [7]. Some authors use "message" in a more general sense. In the official specification of UML, a message defines a specific kind of communication, which can be, for example, raising a signal or creating an instance [2, p. 509].

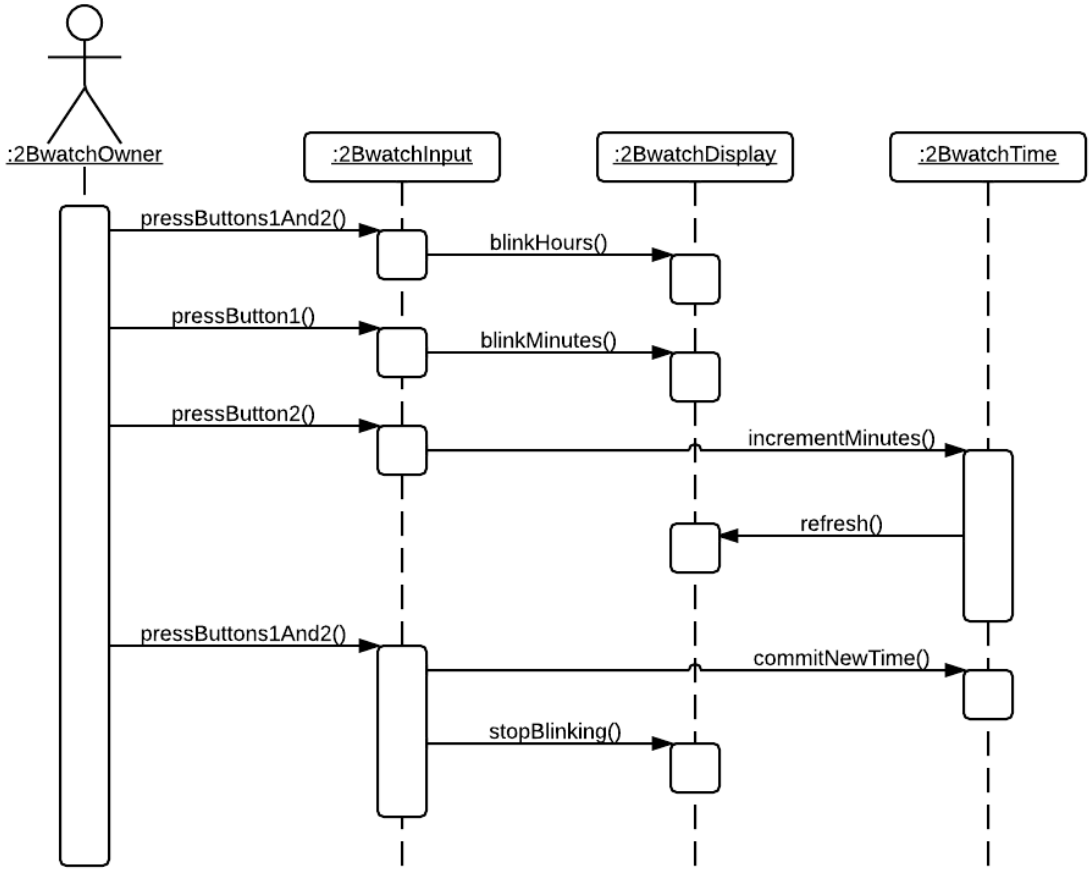

Figure 2.3: A sequence diagram for setting a 2-button watch 1 minute ahead.

The most common kind of UML interaction diagrams is sequences diagrams [2, p. 521]. Sequence diagrams describe interactions between objects by focusing on the sequence of messages that are exchanged between the objects. In these diagrams, objects involved in an interaction are presented horizontally, while time is presented vertically. If the sequence diagram includes an actor that initiates the interaction, the actor is shown in the left-most column. Vertical lines, which are called lifelines, represent the objects. Horizontal arrows between the lifelines represent messages exchanged between the objects. Vertical boxes on the lifelines represent activations (i.e. execution of operations).

As an example, Figure 2.3 depicts a sequence diagram for an actor setting his two-button watch one minute ahead [7].

2.2 Design by Contract

Bertrand Meyer [10] coined the term Design by Contract. In Design by Contract, routines are semantically specified using preconditions and postconditions. A precondition is the condition that must be ensured before calling the routine if it is to function properly; a postcondition is the state that the routine guarantees yielding assuming that it was called with the precondition satisfied. The preconditions and postconditions of a routine define a *contract* that binds it and its callers. In this contract, the routine is the supplier of a service and its callers are the clients. The contract defines obligations and benefits for the supplier and its clients: the precondition is a benefit for the routine and an obligation for the callers; the postcondition is an obligation for the routine and a benefit for the callers.

As an example, consider a square root routine that operates on and produce real numbers. This routine expects its input to be nonnegative. Adopting the notation Meyer uses in his book [10], which is the notation of the Eiffel language [11] and in which he uses a **`require`** clause to express preconditions and an **`ensure`** clause to express postconditions, this constraint can be expressed as a precondition of the routine as follows:

```
sqrt(x: REAL): REAL
  require
    x >= 0
  do
    ...
  end
```

For another example, consider the `pop()` routine of a stack class that can store objects of type `T` and that has a limited capacity. The class developer may

specify that this routine should not be called on an empty stack. This constraint constitutes the precondition of the routine. If the routine is called on a nonempty stack, it will remove the top element. By doing so, it will decrease the number of elements by one and ensure that the stack is not full even if it was so before the call. These guarantees constitute the postconditions of the routine. Adopting the notation of Meyer's again, these constraints and guarantees can be expressed as preconditions and postconditions as follows:

```
pop(): T
  require
    not empty
  do
    ...
  ensure
    not full
    count = old count - 1
  end
```

Design by Contract can be applied to UML models by using the Object Constraint Language (OCL) [12]. OCL is a specification language for UML, not an action language. It is mainly used to write queries to access model elements and their values and state constraints on model elements. UML model elements are annotated with OCL constraints to ensure their proper usage and validity of the whole model. An OCL constraint cannot change the value of a model element and hence is considered side-effect free. It can be used to express preconditions, postconditions, invariants, guard conditions and results of operations.

An OCL constraint typically consists of two parts: the context and a set of OCL expressions. As an OCL constraint highly depends upon which model element is constrained, this information is specified by the first part of the OCL

constraint, that is, its context. An OCL context can either be a class, one of its attributes, or one of its operations, and it can be referenced in the constraint's body by the keyword `self`.

The second part of an OCL constraint consists of a set of OCL expressions, each of which consists of a type, name and body. The frequently used expression types include `inv`, which is used when the body contains a condition that must be met by all instances of a class; `pre`, which is used when the body contains a condition that must be true before executing an operation, i.e. a precondition; and `post`, which is used when the body contains a condition that states what should be true about the state of the system and the changes that occurred after executing an operation, i.e. a postcondition. Since OCL is a query language, it expects a result when querying a property or an operation of the class in context. OCL uses the "`.`" operator when it expects a single-valued result. Multiple expressions in an expression body can be combined by using the boolean operators `and`, `or`, `xor`, `not` and `implies`.

As an example, consider the `pop()` routine used above. Assuming that the stack class has the two boolean routines `isEmpty()` and `isFull()`, along with the integer attribute `count`, the preconditions and postconditions of the `pop()` routine can be expressed in OCL as follows:

```
context Stack::pop(): T
  pre:  not self.isEmpty()
  post: not self.isFull() and
        self.count = self.count@pre - 1
```

2.3 Automated Planning

In the field of artificial intelligence, planning is finding a sequence of actions that will achieve a certain goal. This Section gives an overview of automated planning and is mostly based on the excellent discussions of planning in *Automated Planning: Theory and Practice* by Ghallab, Nau, and Traverso [13] and in Chapters 10 and 11 of *Artificial Intelligence: A Modern Approach* by Russell and Norvig [14].

2.3.1 A Conceptual Model for Planning

The conceptual model that underlies many planning approaches is the general model of state-transition systems. In particular, classical planning relies on the model of restricted state-transition systems. A restricted state-transition system is a triple $\Sigma = (S, A, \gamma)$, where:

- $S = \{s_1, s_2, \cdots\}$ is a finite set of states.
- $A = \{a_1, a_2, \cdots\}$ is a finite set of actions.
- $\gamma : S \times A \to S$ is a state transition function.

A state-transition system can be represented by a directed graph whose vertices are the states in S and whose edges are the state transitions, where there is an edge labeled a from s to s' if $s' = \gamma(s, a)$.

A planning problem for a restricted state-transition system Σ is defined as a triple $\mathcal{P} = (\Sigma, s_0, S_g)$, where:

- $s_0 \in S$ is an initial state.
- $S_g \subseteq S$ is a set of goal states.

Based on this conceptual model, and as Figure 2.4 illustrates, solving a planning problem means finding a sequence of actions that, starting from a particular

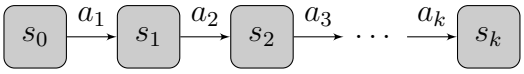

Figure 2.4: A solution to a planning problem as a sequence of actions and state transitions.

state, will achieve a certain goal. More formally, solving a planning problem $\mathcal{P}$ consists of finding a sequence of actions $(a_1, a_2, \cdots, a_k)$ that corresponds to a sequence of state transitions $(s_0, s_1, \cdots, s_k)$ such that $s_1 = \gamma(s_0, a_1), \cdots, s_k = \gamma(s_{k-1}, a_k)$ and $s_k \in S_g$.

2.3.2 Representations for Planning

Automated planners need a description of the problem to be solved. In most—if not all—real-world domains, it would be unrealistic to use a description that explicitly enumerates all the states and state transitions. Instead, the description should make it easy to compute states and state transitions when they are needed.

To accomplish that, Ghallab, Nau, and Traverso [13] distinguish between a planning problem $\mathcal{P}$ and a statement of $\mathcal{P}$, which they define as a triple $P = (A, s_0, g)$. In other words, a statement of a planning problem consists of a set of actions, an initial state, and some specification of goal states. One can consider $\mathcal{P}$ as the semantic specification of the planning problem and P as the syntactic specification.

There is a number of different representations that can be used to describe a planning problem. Some of these representations, which Ghallab, Nau, and Traverso [13] discuss in detail, are:

1. Set-theoretic representation, in which each state is a set of proposition symbols, and each action is an expression that specify which propositions must belong to the state for the action to be applicable and which propositions the action will add or remove from the state to create a new state.

2. Classical representation, which generalizes the set-theoretic representation by using first-order logic.

3. State-variable representation, in which each state is a tuple of values of state variables $\{x_1, \cdots, x_n\}$, and each action is a partial function mapping this tuple into another tuple of values of the state variables.

As Ghallab, Nau, and Traverso [13] note, the classical representation is linked to the Stanford Research Institute Problem Solver (STRIPS) [15], one of the early automated planners. The Action Description Language (ADL) [16, 17], which was introduced later, presented a trade-off between the complexity of computing the state transitions and the expressive power of general logic formalisms. As several automated planners were generalized to representations close to ADL that handle extensions to the classical representation—ranging from simple syntactical extensions to disjunctive preconditions and extended goals—these extensions were then implemented in the Planning Domain Definition Language (PDDL) [18].

In the three representations listed earlier, actions are essentially specified in the same way: by using preconditions and effects. Preconditions are the conditions that must be true in a state for the action to be applicable to that particular state. Effects are the changes that the action make to the state it is applied to. For an example, let us revisit the `pop()` routine that was used in Section 2.2. Treating `pop` as an action that takes a stack as a parameter, and assuming that `count` is a function that computes the number of items in a stack, the `pop` action can be specified in PDDL as follows:

```
(:action pop
   :parameters (?s - Stack)
   :precondition (> (count ?s) 0)
   :effect (decrease (count ?s) 1)
```

)

As Ghallab, Nau, and Traverso [13] point out, the classical and state-variable representations are equivalent in their expressiveness. That is, a planning problem expressed in one of these two representations can be translated, with at most a linear increase in size, into the other. On the other hand, and although the three representations can represent the same set of planning domains, a set-theoretic representation may take exponentially more space than the equivalent classical and state-variable representations.

2.3.3 Planning Algorithms

Using the conceptual model of state-transition systems mentioned in Subsection 2.3.1, an automated planner can search the space of states looking for a sequence of actions that will connect, through state transitions, the initial state s_0 with a goal state $s_g \in S_g$. This is known as state-space search. Using preconditions and effects, the planner can move in the search space in either direction: *progressing* forward from the initial state or *regressing* backward from the goal. The former is known as forward search, while the latter is known as backward search.

In forward search, the three basic steps that the planner need to carry out regardless of the representation used are:

1. Determine if a state is a goal state or not.

2. Find the set of all applicable actions to a state.

3. Compute the successor state resulting from applying an action to a state.

The advantage of regression planning is that it allows the planner to ignore irrelevant actions.

Planning with state-space search is one form of total-order planning. There is also the more flexible partial-order planning, in which a planner generates several subplans for several subgoals independently, and then combines those subplans. This approach is partially ordered because the actions within the subplans are ordered, but the planner does not commit to whether an action in one subplan is before or after an action in another subplan—or at least it will delay the decision as much as possible. The partial-order solution usually corresponds to several total-order plans. Unlike total-order planners, which search the space of states, a partial-order planner usually searches the space of plans: it starts with an empty plan, then it refines the plan until it gets a complete one that solves the problem. In other words, the states that the partial-order planner searches are plans and the actions that it considers are plan refinement steps.

Another planning method, which can be viewed as an extension of partial-order planning, is known as hierarchical task network (HTN) planning. In this approach, the initial plan represents a high-level description of what needs to be done. The plan is refined through action decompositions. These decomposition rules reduce high-level actions to partially ordered sets of lower-level ones, which means that they include knowledge about action implementation. The decomposition continues until only primitive actions remain in the plan. The hierarchical decomposition of HTN planning helps it in generating and handling the large plans required by many real-world applications.

Many of the heuristics for total-order and partial-order planning can suffer from inaccuracies. Planning graphs can be used to overcome this problem in both approaches to planning. A planning graph is a special data structure that consists of a number of levels corresponding to time steps in the plan. Each level contains a set of literals, which are the ones that could be true at that time step, and a set of actions, which are the ones that could have their preconditions satisfied at

that time step. Each level also encodes mutual exclusion relations between literals or actions that cannot co-occur. As an alternative to using planning graphs in total-order or partial-order planners, a specialized algorithm can directly extract a solution from these graphs. One such algorithm is called GRAPHPLAN. It uses a backward search to extract a plan while allowing for some partial ordering between the actions.

Another approach to planning is to translate the problem into propositional axioms and then determine the satisfiability of the resulting conjunctive normal form (CNF) sentence. The resulting propositional sentence will look like this:

$$\text{initial state} \wedge \text{all possible action descriptions} \wedge \text{goal}$$

Every possible action occurrence will have a corresponding proposition symbol in that sentence. A model satisfying the sentence will assign $true$ to the actions that belong to a correct plan and $false$ to the others. Thus, if the original planning problem is solvable, such a model will exist and a plan can be extracted by looking at those symbols that are assigned $true$. If, on the other hand, the original planning problem is unsolvable, then no model will exist. That is, if the planning problem is unsolvable, the sentence will be unsatisfiable.

CHAPTER 3

LITERATURE SURVEY

The literature surveyed for this Thesis can be divided into three general groups: works that focused on generating models automatically, works that focused on analyzing the consistency of different models, and works that connected automated planning on one hand and requirements and knowledge engineering on the other. The works that focused on generating models automatically did not tackle the problem of analyzing the consistency between different models. In addition, these works did not tackle the problem of generating sequence diagrams automatically as a planning one. Instead, they used use case templates, restrictions on natural language, and transformation rules to translate use cases to sequence diagrams. Some of the works that focused on analyzing the consistency of different models used Design by Contract, but they, along with the other consistency analysis works, assumed that these models already exist. That is, they did not address the problem of generating models from others.

The following three Sections will discuss each of the three groups mentioned above separately.

3.1 Automatic Model Generation

Yue, Briand, and Labiche [19] present a systematic review that focuses on transforming textual requirements to analysis models in the context of model-driven development. They review 16 approaches and evaluate their capabilities, support for establishing traceability links, degree of automation, efficiency, and completeness. To do that, they design a conceptual framework and derive a set of evaluation criteria from it. The authors observe that none of the existing approaches is easily applicable on real systems, has less than two transformation steps, or able to automatically generate a complete and consistent analysis model. They suggest a pattern for future approaches that consists of using reasonable restrictions on natural language, an automatic requirements preprocessing technique, and one intermediate model to transform a use case model into a UML model that compromises structural and behavioral aspects. They also suggest some quality characteristics for transformation approaches, like usability, efficiency, scalability, and extensibility.

Yue, Briand, and Labiche [4] follow their own suggestion and propose a technique to automatically derive analysis models, including sequence diagrams, from use cases while maintaining traceability links. Requirements must first be defined manually using the Restricted Use Case Modeling (RUCM) approach, which the authors proposed in an earlier work [20]. The result of RUCM is a textual use case model (UCModel) that is expressed in a restricted natural language. aToucan transforms UCModel into an intermediate model (UCMeta), which is then transformed into the desired analysis model. At the same time, aToucan establishes traceability links between UCModel and the generated analysis model. Yue, Briand, and Labiche devised an evaluation framework to compare the sequence diagrams generated automatically by aToucan to ones generated manually by experts and undergraduate students. The empirical study shows that the auto-

matically generated diagrams were very complete and consistent with the ones generated by experts and that they were more complete than the ones generated by students.

If one ignores the fact that Yue, Briand, and Labiche [4] do not use planning, their work may appear to be quite similar to the current one. On further inspection though, one can see that in their approach, generating a class diagram is a prerequisite for generating the sequence diagram. This means that generating sequence diagrams must follow a linear path that starts with use cases and passes through class diagrams. With such a linear path, detecting inconsistencies by cross-referencing can not be carried out since any defect that is included in the use cases will be carried through to the subsequently generated models. Of course, the traceability links will help in tracing and fixing defects and inconsistencies, but they will not help in detecting them. In contrast, the current work focuses on planning message passes to generate sequence diagrams automatically from use cases and class diagrams, especially if the use cases and class diagrams were developed *independently*. If that was indeed the case, the planning-based approach will open the door to consistency analysis by cross-referencing the different models.

Liwu Li [21] presents a parser that translates a manually normalized use case to message records which may then be used to construct sequence diagrams. Li provides four guidelines for use case normalization. Given a normalized use case, the parser identifies syntactic structures to deduce static information including classes, objects, attributes, and operations, and dynamic information including message sends. The user may need to manually instruct the parser on how to translate some sentences. This, coupled with the fact that the normalization step is manual, makes the approach semi-automatic. Li presents an ATM use case as an example but does not provide empirical results.

3.2 Consistency Analysis

Li, Liu, and He [22] validate UML requirements by checking the consistency between the use case model and the conceptual class model, which contains constraints in the form of preconditions and postconditions. They define five types of consistency: consistency of each use case, consistency of each constraint, consistency between constraints, consistency between use cases, and consistency between use cases and constraints. Their formalization is based on set theory and first-order logic. Consistency is checked by determining if there is a conflict between related requirements. Such conflicts can only arise between use cases that depend on each other or between a constraint that impacts some use case. Informal requirements are converted to formal ones manually. In addition, the authors do not discuss automatic consistency checking explicitly despite that the formalization appears to enable it. It is worth stressing that the class model in this work does not contain methods. That is, this approach checks the consistency between use cases and class diagrams at a higher, more abstract level that does not involve sequence diagrams.

Long et al. [23] propose an algorithm for checking the consistency between a class diagram and a sequence diagram. The algorithm uses a queue and breadth-first search to detect inconsistencies between well-formed class and sequence diagrams. It does that by checking conditions on associations, class names, methods, attributes, and multiplicity. Long et al. also propose an algorithm for generating code in the Relational Calculus of Object Systems (rCOS) language. rCOS is an object-oriented design language equipped with observation-oriented semantics. After the class and sequence diagrams are checked for consistency, the code generating algorithm generates the code skeleton from the class diagram and fills the program text using the sequence diagram. The proposed algorithms do not generate class or sequence diagrams. They also do not use Design by Contract. The

consistency conditions used in the consistency checking algorithm suggest that the approach focuses on, in a sense, the syntax of the models. By contrast, the current work tries to focus on their semantics.

Ghezzi, Mocci, and Salvaneschi [24] address the general problem of automatically comparing two formal specifications of stateful components. They specifically cross-validate algebraic specifications against intensional behavior models using symbolic model checking. The algebraic specifications of a component are transformed into temporal logic formulae, while the intensional behavior model is transformed into a behavioral equivalence model (BEM). The NuSMV model checker compares the temporal logic formulae against the BEM to determine whether the original formal specifications are consistent or not. The authors acknowledge that the approach does not provide a proof of consistency; it provides a comparison of behavioral information under a finite subset of the component's behaviors.

Chanda et al. [25] provide a context-free grammar for use case, activity, and class diagrams. They also propose a verification criteria that includes syntactic correctness, inter-diagram consistency, and requirement traceability. Like the other surveyed works that focus on consistency analysis, this one does not deal with the problem of generating models automatically; it assumes that the diagrams already exist.

Köster, Six, and Winter [26] validate and verify requirements specifications by using refined activity graphs to couple use cases and class models. The use case model is validated first, then the class model is verified against it. Verification consists of determining if the instances of classes can execute the use cases. This approach does not appear to focus on automatic consistency analysis.

de Sousa et al. [27] present an approach to automatically analyze the consistency of requirements using the B method, which is a formal method based on set

theory, language of generalized substitutions, and first-order logic. The approach relies on a controlled natural language to automatically transform use case scenarios and constraints to B specifications. Then, ProB, a model checker for B, analyzes these specifications searching for inconsistencies.

3.3 Requirements Engineering and Automated Planning

The technique of using automated planning to plan messages in sequence diagrams that is proposed and implemented in this Thesis puts it at an intersection between software engineering and automated planning. A related research area is knowledge engineering for planning and scheduling [28]. An example of a work in this area is that of Vaquero et al. [29], in which they report their research efforts to develop itSIMPLE, an integrated design environment for automated planning applications. They view the design process of a planning application project as a series of phases, such as requirements specification, model analysis, plan synthesis and analysis, where each phase requires a different knowledge representation. In itSIMPLE, requirements are first modeled using UML. Then, these UML models are analyzed using Petri nets. Finally, these models are automatically translated to a Planning Domain Definition Language (PDDL) [18] representation and are presented to an automated planner.

The work of Vaquero et al. [29] played an important role in the early stages of our research. As part of demonstrating the basic idea of using Design by Contract and automated planning to automate the generation of sequence diagrams, we [30] used itSIMPLE to model a use case and a class diagram of a simple online banking system. We added the necessary preconditions and postconditions of the methods as Object Constraint Language (OCL) [12] expressions, then used

itSIMPLE itself to translate the models along with their constraints into planning domain and problem descriptions expressed in the Planning Domain Definition Language (PDDL) [18]. Finally, we used the planners bundled with itSIMPLE to generate the plan that solves the planning problem.

The difference between our previous research efforts [30] (which ultimately lead to this Thesis) and those of Vaquero et al. [29] is subtle, yet crucial. Vaquero et al. applied requirements engineering and knowledge engineering to planning, while we are applying automated planning to requirements engineering. In other words, for Vaquero et al., planning was the domain and the generated plans were the ultimate output; for us, planning is the tool and the generated plans are a step towards the ultimate outputs (sequence diagrams).

Based on the difference highlighted above, we were, in a sense, using itSIMPLE backwards: we were trying to use it to solve a problem that it was not designed to solve. Because of that, as we tried to use itSIMPLE on larger examples created by other people, we faced considerable limitations that helped us realize some of the fundamental differences between action planning and message-pass planning. For example, consider the plan shown in Listing 3.1. This plan was generated by the action planners bundled with itSIMPLE for a withdraw use case of a simple online banking system, which we [30] created to demonstrate the basic idea of treating sequence diagram generation as a planning problem. Because the plan was generated by action planners, it contained information corresponding to the message name (e.g. `WITHDRAW`), receiver (e.g. `ACCOUNT`), and parameters (e.g. `AMOUNT`), but lacked any information about the message sender, which is needed to convert the action plan to a sequence of message passes. As Section 4.3 will explain in more details, determining the senders of the messages can not be really done through post processing the action plan or through somehow statically adding the information to the models or states.

```
0: (LOGIN PROFILE) [1]
1: (ACTIVATE ACCOUNT PROFILE) [1]
2: (WITHDRAW ACCOUNT PROFILE AMOUNT) [1]
3: (LOGOUT PROFILE) [1]
```

Listing 3.1: The plan reported by itSIMPLE for a withdraw use case of an online banking system.

Based on the differences between action planning and message-pass planning as well as the difficulties we faced in adapting existing action planners to the domain of sequence diagrams, we decided to change our research direction of using existing planners and to design and implement a planner specific for planing messages in sequence diagrams. Section 4.5 will provide details about Communiqué, the software library we implemented for that purpose.

3.4 Summary

Table 3.1 provides a summary of the literature that was discussed in this Chapter.

Paper	Focus	Main Idea
Liwu Li [21]	Model Generation	A parser to translate a manually normalized use case to message records
Kösters, Six & Winter [26]	Consistency Analysis	Using refined activity diagrams to couple use cases & class models
Li, Liu & He [22]	Consistency Analysis	Using set theory & first-order logic to check consistency between the use case model & the conceptual class model
Long et al. [23]	Consistency Analysis	Using a queue & BFS to detect inconsistencies between well-formed class & sequence diagrams
Chanda et al. [25]	Consistency Analysis	A context-free grammar for use case, activity & class diagrams
Yue, Briand & Labiche [19]	Model Generation	A systematic review focusing on transforming textual requirements to analysis models in the context of MDD
Yue, Briand & Labiche [4]	Model Generation	A technique to automatically derive analysis models from use cases while maintaining traceability links
Ghezzi, Mocci & Salvaneschi [24]	Consistency Analysis	Using symbolic model checking to cross-validate algebraic specifications against intensional behavior models
de Sousa et al. [27]	Consistency Analysis	Using the B method to automatically analyze the consistency of requirements
Vaquero et al. [29]	RE & Automated Planning	itSIMPLE: an IDE for automated planning applications
Sulaiman & Ahmed [30]	RE & Automated Planning	Using itSIMPLE to demonstrate treating sequence diagram generation as a planning problem

Table 3.1: Summary of Literature Survey

CHAPTER 4

SEQUENCE DIAGRAM GENERATION AS A PLANNING PROBLEM

This Chapter presents and explains in detail the core ideas underlying the planning-based approach for automating sequence diagram generation. It first highlights the correspondence between automated planning and Design by Contract. It then discusses the planning formulation needed for planning messages in sequence diagrams. Then, it expounds the differences between action planning and messages-pass planning. The Chapter then discusses the details of the planning algorithm. Finally, it presents the message-pass planner that was developed as part of the research.

4.1 The Correspondence between Automated Planning and Design by Contract

When Design by Contract is used to develop the use cases and the class diagram of a software system, generating the sequence diagrams based on them starts to

look very similar to solving a planning problem. In both cases, and despite some differences in terminology, the same building blocks are present.

In Design by Contract, and as was mentioned in Section 2.2, routines are semantically specified using preconditions and postconditions, where preconditions are the conditions that must be ensured before calling the routine if it is to function properly, and postconditions specify the state that the routine guarantees yielding if it was called with the preconditions satisfied. As was mentioned in Subsection 2.3.2, actions in automated planning are essentially specified by using preconditions and effects, where preconditions are the conditions that must be true in a state for the action to be applicable to that particular state, and effects are the changes that the action make to the state it is applied to. So, methods and their preconditions and postconditions in Design by Contract correspond to actions and their preconditions and effects in automated planning.

Furthermore, and as was mentioned in Subsection 2.1.1, use cases usually have preconditions, which are the conditions that must be ensured before initiating the use case, and postconditions, which are the conditions that will be true after the completion of the use case. One can look at the preconditions and postconditions of a use case as the specification of the initial state and a goal state of a planning problem.

This correspondence between automated planning and Design by Contract, which Table 4.1 summarizes, opens the door to treating the core activity of sequence diagram generation, which is determining the sequence of messages exchanged between the objects involved in the interaction, as a planning problem. This, in turn, paves the way for using automated planning techniques to solve the problem of sequence diagram generation.

The following Section formally ties the correspondence discussed above with the conceptual model for planning that was presented in Subsection 2.3.1.

Automated Planning		Design by Contract
Initial State	$\Leftrightarrow$	Use Case Preconditions
Goal	$\Leftrightarrow$	Use Case Postconditions
Actions	$\Leftrightarrow$	Methods
Action Preconditions	$\Leftrightarrow$	Method Preconditions
Action Effects	$\Leftrightarrow$	Method Postconditions

Table 4.1: Correspondence between automated planning and Design by Contract.

4.2 A Conceptual Model for Planning Messages in Sequence Diagrams

A restricted state-transition system that can be used for the purpose of planning messages in sequence diagrams is the triple $\Sigma = (S, M, \gamma)$, where:

- S is a set of states, where each state $s \in S$ is a set of objects and the links between them.
- $M = \{m_1, m_2, \cdots\}$ is the set of all methods of the objects in s.
- $\gamma : S \times M \rightarrow S$ is a state-transition function.

The following Subsections will discuss the formalization of Σ in terms of states, actions, state transitions, problems, plans, and solutions in more detail.

4.2.1 States

In the context of planning messages in sequence diagrams, a *state* is a set of objects. Since each object is an instance of a class, the object will have specific values for its attributes. These attributes specify what is called the state of the object, but this state is internal and should not be confused with the states that are used in planning. The values of some of an object's attributes may be themselves other objects in the state. These particular attributes represent links between the objects in the state. In short, a state is conceptually an object diagram, which is

an instance of a class diagram. The set S of the state-transition system Σ is the set of all possible object diagrams.

4.2.2 Actions

An *action*, in the context of planning messages in sequence diagrams, is a method of an object of a state. Accordingly, an action is a triple $m = (\text{name}(m), \text{precond}(m), \text{effects}(m))$ where:

- name(m) is the name of the method m.
- precond(m), the *preconditions* of m, is a set of expressions on the objects in the state to which the object of m belongs.
- effects(m), the *effects* (or *postconditions*) of m, is a set of assignments of values to the attributes of the objects in the state to which the object of m belongs.

An action m is applicable in a state s if s satisfies precond(m). That is, if the objects of s are in internal states that meet the conditions in precond(m) and, as a result, make precond(m) evaluate to true in s. The number of applicable actions in a s is equal to the number of methods of the objects of s having preconditions that are satisfied by s. The total number of actions in s is equal to the number of all the methods of all the objects in s.

4.2.3 State Transitions

Since the states in the context of planning messages in sequence diagrams are conceptually object diagrams, the state-transition function γ essentially transforms object diagrams to other object diagrams. The state $s' = \gamma(s, m)$ is the state that results from calling an applicable method m in s. s' can be computed by applying

effects(m), the effects or postconditions of m, to s. If m is not applicable in s, $\gamma(s, m)$ is not defined.

Figure 4.1 shows a simple example of the states and state transitions of the planning model described so far. As mentioned above, a state can be composed of a set of objects and links between them. The two states s_{out} and s_{in} represent the object diagrams where the user is logged out and is logged in respectively. s_{out} contains a single object, `user`, which is an instance of a `User` class that has one boolean attribute, `is_logged_in`, and two methods, `log_in` and `log_out`. s_{in} contains two objects: `user` and `session`, which is an instance of a `Session` class and which gets instantiated for the user upon his login. Invocations of the methods `log_in` and `log_out` cause state transitions from one state to the other. This example highlights the fact that states can grow or shrink as objects are instantiated or terminated.

4.2.4 Problems, Plans, and Solutions

For the purpose of planning messages in sequence diagrams, a *planning problem* is defined as a triple $\mathcal{P} = (\Sigma, s_0, S_g)$, where:

- Σ is the state-transition system defined above.
- $s_0 \in S$ is an initial state, which represents the object diagram specified by the use case preconditions.
- $S_g \subseteq S$ is a set of goal states. This set represents the object diagrams that satisfy the use case postconditions.

Just as Ghallab, Nau, and Traverso [13] distinguish between a planning problem $\mathcal{P}$ and its statement P (to be able to specify $\mathcal{P}$ without enumerating all the states and state transitions), the *statement* of $\mathcal{P}$ is defined here as the triple $P = (M, s_0, g)$, where g is a set of goal conditions: conditions that a state must

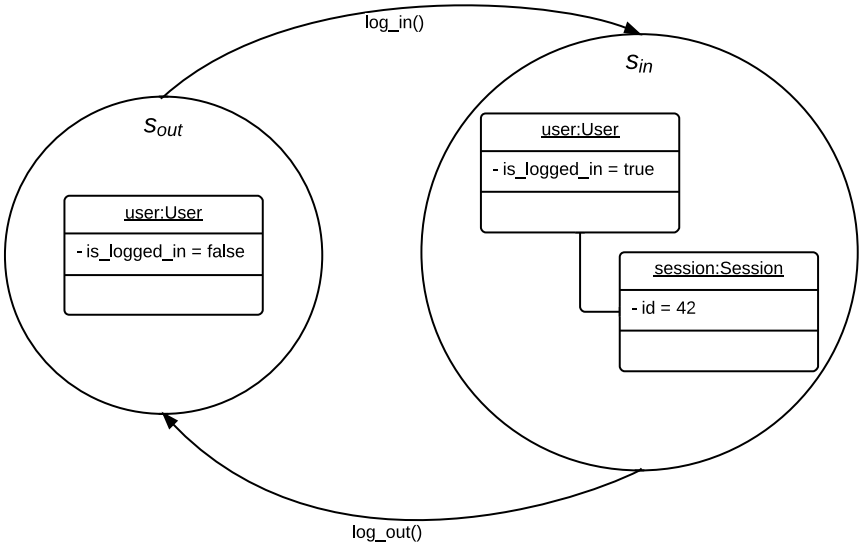

Figure 4.1: States (as object diagrams) and state transitions.

satisfy for it to be a goal state. Using the use case postconditions as the goal conditions, g specifies S_g, which represents all the object diagrams that satisfy the use case postconditions.

For reasons that will become clear in Section 4.3, a plan in the context of sequence diagram generation is not defined as it is usually defined in a general planning context: any sequence of actions. Instead, a *plan* π is defined as any sequence of message passes $\langle \mu_1, \cdots, \mu_k \rangle$, $k \leq 0$, where each message pass μ_i consists of an action m_i, a sender, and a receiver. The length of the plan is $|\pi| = k$.

To be able to denote the state s' that results from applying a plan π to a state s, the state-transition function γ needs to be extended as follows:

$$\gamma(s, \pi) = \begin{cases} s & \text{if } k = 0 \\ \gamma(\gamma(s, m_1), \langle \mu_2, \cdots, \mu_k \rangle) & \text{if } k > 0 \text{ and } m_1 \text{is applicable in } s \\ \text{undefined} & \text{otherwise} \end{cases}$$

To put it in other words, $s' = \gamma(s, \pi)$ is the state that results from applying the methods of π one by one and in the order they are given in π. With this extended γ, a plan π is said to be a *solution* for a planning problem $\mathcal{P}$ having a statement $P = (M, s_0, g)$ if $\gamma(s_0, \pi)$ satisfies g. That is, if applying π to the initial state s_0 produces a goal state (a state that satisfies the set of goal conditions g).

4.3 Differences between Action Planning and Message-Pass Planning

Based on the conceptual model presented at the beginning of Section 4.2, one might initially be tempted to think that solving a planning problem $\mathcal{P}$ in the con-

text of sequence diagrams consists of finding a sequence of methods $(m_1, m_2, \cdots, m_k)$ the execution of which, starting from an initial object diagram s_0, will result in an object diagram that satisfies the given goal conditions (i.e. use case postconditions). But the sequence of methods $(m_1, m_2, \cdots, m_k)$ is not enough to actually generate a sequence diagram. The reason, in a nutshell, is that a sequence diagram depicts more than a sequence of method; it depicts a sequence of message passes. To put it differently, a sequence of methods, just like a sequence of actions, is unidimensional, while a sequence diagram is 2-dimensional.

In a sequence diagram, the vertical dimension shows how the message passes are ordered in time. On the horizontal dimension, each message pass specifies, in addition to the method the execution of which is being requested, a sender and a receiver. So even though the sequence of methods $(m_1, m_2, \cdots, m_k)$ may represent the correct chronological order of the message that needs to be sent, it does not contain any information about the senders and receives of those messages.

Assuming that each message maps to a certain method, determining the receiver of a message is easy. The reason is that methods are not disembodied actions: methods do not exist on their own; they are defined in some class and are executed in some instance of that class. Furthermore, this type of information is static: it does not change from state to state. Since every method essentially always belongs to a specific object, once an automated planner decides that a certain method needs to be executed next, it can determine the object to which the method belongs using the class or object diagram and, in turn, determine the receiver of the message that needs to be sent.

The more challenging task is determining the sender of a message. The reason is that this type of information is dynamic: it may change from one state to the other. An object can send a message to a receiving object if it has a link to it; the object cannot send messages to other objects to which it is not linked. Because

links are dynamic and can change over time, a valid sender of a message in some state may become an invalid sender in the next. Based on this, information about the senders of message cannot be statically added to the problem description in advance. Instead, the automated planner must dynamically infer such information using the links between the objects in the current state.

Another difference between action planning and message-pass planning lies in the planning representations. As powerful as they are, representations for action planning—such as the Stanford Research Institute Problem Solver (STRIPS) [15], the Action Description Language (ADL) [16, 17], and the Planning Domain Definition Language (PDDL) [18]—do not contain constructs that support the concepts of the Unified Model Language (UML) and object-oriented software modeling directly. Of course, some tricks can be used in action planning representations to simulate or support such concepts. For example, a `pop()` method of a `Stack` class that accesses a `count` attribute of `Stack` in its preconditions and postconditions can be modeled in PDDL as a `pop` action that takes a stack as a parameter and uses a `count` function that itself takes a stack. Still, such support for UML and object-oriented software modeling concepts is not as natural and fluid as it is in a representation that was designed with UML and object-orientation in mind, such as the Object Constraint Language (OCL) [12]. Therefore, a message-pass planner that operates on UML models should ideally use OCL or a similar object-oriented representation instead of an action-planning representation.

4.4 The Planning Algorithm

A forward-search algorithm for message-pass planning is given in Algorithm 1. FORWARD-SEARCH takes an initial state s_0, which represent an object diagram, and a set of goal conditions g. It uses a priority queue to save the generated

Algorithm 1 Forward-Search Algorithm for Message-Pass Planning

```
 1: function FORWARD-SEARCH(s_0, g)
 2:     n ← [s_0, the empty plan]
 3:     Enqueue(n, f(n))
 4:     while queue is not empty do
 5:         s, π ← Dequeue-min()
 6:         if s satisfies g then return π
 7:         applicable ← {m | precond(m) is true in s}
 8:         if applicable = φ then next
 9:         for all m ∈ applicable do
10:             n ← [γ(s, m), π.m]
11:             Enqueue(n, f(n))
12:     return failure
```

states and select the next state to be explored, which is the state that minimizes an objective function $f(n)$. Because a priority queue is used, different control strategies (such as depth-first and breadth-first) can be used by changing $f(n)$. This aspect and others related to $f(n)$ are discussed further in Subsection 4.4.1.

When FORWARD-SEARCH finds a state that satisfies g, it returns a solution π, which is the sequence of message passes that leads to that state. If all states were explored and none of them were found to be a goal state, FORWARD-SEARCH returns failure.

4.4.1 The Objective Function $f(n)$

The objective function $f(n)$, where n is a node consisting of a state (object diagram) and the sequence of message passes leading to it, facilitates having different control strategies for exploring the nodes in the search space. Depth-first, breadth-first, and best-first control strategies can be implemented by defining $f(n)$

as follows:

$$f(n) = \begin{cases} -g(n) & \text{for depth-first search} \\ g(n) & \text{for breadth-first search} \\ g(n) + h(n) & \text{for best-first search} \end{cases}$$

The function $g(n)$ is the cost to reach the node n, and is equal to the number of message passes that lead to the object diagram represented by the state contained in n. Because the planning algorithm uses a priority queue to store states in an ascending order of f-values, the objective function $f(n) = -g(n)$ gives deeper nodes higher priority. For the same reason, $f(n) = g(n)$ gives shallower nodes higher priority.

The heuristic function $h(n)$ is the estimated cost of the cheapest path from the node n to a goal node. In other words, it is the planner's guess of the number of message passes still needed to produce an object diagram that satisfies the goal conditions. $h(n)$ and its properties will be discussed further in the next Subsection.

4.4.2 The Heuristic Function $h(n)$

One way to specify the goal conditions is to use a collection of key-value pairs where each key is an object name and the value is the conditions that the object must satisfy. Using this specification, $h(n)$ can be defined as follows:

$$h(n) = \text{the number of objects not satisfying their conditions}$$

That is, a planner using this $h(n)$ will estimate that the number of message passes still needed to produce a goal object diagram from the current one is equal to the number of objects not satisfying their goals. So for example, if there are two unsatisfied objects in the current object diagram, the planner will estimate that

two more message passes are needed to satisfy the two objects. Of course, this is only an estimate and thus it can be wrong. So it may be the case that one more message pass will actually satisfy the two objects together. So $h(n)$ is not admissible: it may overestimate the true cost of reaching the goal.

Since $h(n)$ is not admissible, the planner's best-first search is not optimal and may return a non-optimal solution. One of the experiments presented in Chapter 5 demonstrates the non-optimality of the planner's best-first search with a simple contrived example.

4.4.3 Sender-Selection Rules

As was discussed in Section 4.3, a message-pass planner must dynamically infer, based on the current state, the sender of each message in the current sequence of message passes. Based on the sequence diagrams encountered during the research behind this Thesis, a list of rules of thumbs for message-pass planners to use when selecting a sender was compiled. After the planner determines the receiver of the message under consideration, the planner can use the rules listed below to select a sender for the message.

Rule of Thumb 1 *If the message is the first in the sequence of message passes, select the Actor as the sender.*

Actors initiate use cases to access the functionality provided by a software system. So, in a sequence diagram that models the interactions between objects participating in a use case, the sender of the first message is the actor initiating that use case.

Rule of Thumb 2 *If the receiver of the message is a boundary object, select the Actor as the sender.*

Boundary objects represent the interactions between the user and the system.

Therefore, a message received by a boundary object is most likely sent by the actor.

Rule of Thumb 3 *If the receiver of the message is a dependency object (i.e., an object that received a* <<create>> *message at a previous point in time), select the object's creator (i.e., the object that sent it the* <<create>> *message) as the sender.*

In sequence diagrams, objects may instantiate other objects by sending <<create>> messages. In such a case, the dependent object instantiate the independent object usually to use it by sending it further messages. So, a message received by the independent object is most probably sent by the dependent object that instantiated it in the first place.

Rule of Thumb 4 *If the message has more than one candidate sender, where an object is considered as a* candidate sender *of a message if the object has a link to the receiver of that message and has been previously activated (i.e., it received a message at an earlier point in time), select the candidate sender that was activated most recently (i.e., the last candidate sender to receive a message) as the sender.*

Based on the manually-designed sequence diagrams encountered for this Thesis, the candidate sender of a message most likely to be the correct one (that is, the one selected by the sequence diagram creator) is the most recently activated one.

Rule of Thumb 5 *Using the definition of a candidate sender given in the previous rules, if the message does not have any candidate sender, select the Actor as the sender.*

Since the planner has determined that the message must be sent to make the resulting sequence diagram a solution to the use case at hand, it must select a

sender even if there are no candidate ones. If that is the case, there is always the possibility that the actor is the sender of that crucial message.

As they are rules of thumb, the planner may select a sender other than the one that the software designer originally had in mind. That is, the output of the planner may result in a sequence diagram that is different—in terms of the senders of the messages—from the one that the software designer may have come up with manually. One of the experiments presented in Chapter 5 will demonstrate an example where one of the rules listed above caused the planner to produce an output that is different from the original manually-designed sequence diagram.

4.5 Communiqué: A Message-Pass Planner

Due to the differences between action planning and message-pass planning that were discussed in Section 4.3 as well as the difficulties faced in adapting existing action planners to the domain of sequence diagrams, the decision to implement an automated planner that is specialized for planning messages in sequence diagrams was made.

The automated planner and its enclosing software library, Communiqué, was implemented in Ruby 1.9.3, which is a dynamic object-oriented programming language [31]. The source code of Communiqué, along with the scripts for the experiments used in Chapter 5, is available online.[1]

4.5.1 Class Diagram

Figure 4.5.1 shows the class diagram of Communiqué. Since Ruby is a dynamically-typed language, the types shown in the classes represent intended types. Following the syntax of Ruby for instance variables, attributes are preceded with the `@` sign.

[1]https://github.com/ysulaiman/communique

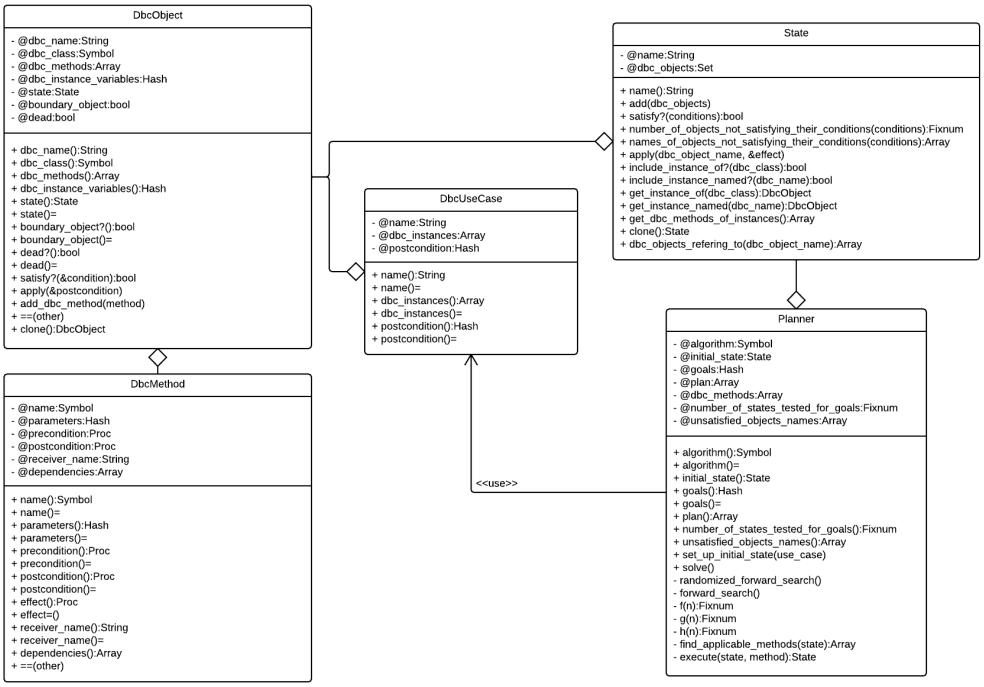

Figure 4.2: The class diagram of Communiqué.

The `DbcMethod` class represents a method with preconditions and postconditions. The `DbcObject` class represents objects the methods of which have preconditions and postconditions. The `DbcUseCase` class represents a use case that is designed following the Design by Contract approach. The `State` class represents the states of the planning model. Those states are basically object diagrams. The `Planner` class is the core class of Communiqué and uses `DbcUseCase`s and `State`s to plan sequences of message passes.

4.5.2 Usage Example

To demonstrate the usage of Communiqué, a simple example will be used. Assume that there is a `User` class that has a single boolean `is_logged_in` attribute and two methods, `log_in` and `log_out`, that set the boolean attribute to `true` and `false` respectively. Assume further that `log_in` must not be called if the user

```
log_in = DbcMethod.new(:log_in)
log_in.precondition = Proc.new { ! @is_logged_in }
log_in.postcondition = Proc.new { @is_logged_in = true }
```

Listing 4.1: The specification of an example log in method in Communiqué.

is already logged in, and that `log_out` must not be called if the user is already logged out.

Instances of `User` can be created as follows:

```
user_instance = DbcObject.new('user', :User, {
  :@is_logged_in => false
})
```

The above essentially creates an instance of `User` named `user` (that can have methods with preconditions and postconditions) and sets the `is_logged_in` to the default value of `false`.

The `log_in` method with its precondition and postcondition can be specified in Communiqué as shown in Listing 4.1. The first line simply creates the method and gives it a name, the second specifies the method's precondition, and the third specifies the postcondition

Before continuing with the example, the way in which preconditions and postconditions are specified and handled in Communiqué is worth elaborating on here.

In Communiqué, preconditions and postconditions are expected to be instances of `Proc`. `Proc`s are code blocks, which are simply chunks of code enclosed between braces or the `do` and `end` keywords, that are converted to objects. These blocks-turned-objects can be, like any other object, stored in variables and passed around as parameters, but unlike non-block objects, they can be executed.

Communiqué uses precondition blocks to determine which of the methods are applicable to the current state. It does that by evaluating the precondition block in the context of the `DbcObject` instance to which the method belongs and seeing

whether the block evaluates to true or to false. In a similar manner, Communiqué uses postcondition blocks to compute the state-transition function $\gamma(s, m)$, that is, the state that results from applying a method m to the current state s. In other words, a precondition or postcondition block in Communiqué is what Paolo Perrotta calls a "*Context Probe*, ... a snippet of code that you dip inside an object to do something in there" [32, p. 83]. Therefore, the contract of a method should generally be written from the point of view of the object to which the method belongs.

By using Ruby code blocks, the need to implement a parser for some contract language was avoided. Instead, Communiqué uses Ruby itself, which is Turing-complete, as the language for the contracts. This gives the user great power and flexibility because it means that the user can write almost whatever he or she wants in the contracts as long as it is valid Ruby code. For example, the user can specify the contracts in a syntax similar to that of the Object Constraint Language (OCL) [12]. For example, the OCL specification of the `pop()` method that was presented under Section 2.2 can be written in Communiqué as follows:

```
pop.precondition = Proc.new { not self.is_empty? }
pop.postcondition = Proc.new { @count -= 1 }
```

Furthermore, Ruby collection classes (such as `Array`, `Hash`, and `Set`) with the methods they provide can be used to simulate OCL collections and the operations. For example, assume that the precondition of a `registerCourse` method of a `Student` class is that the list of courses the student already registered does not contain the new course. This can be expressed in OCL as follows:

```
context Student::registerCourse(c:Course)
  pre: not courses->includes(c)
```

In Communiqué, the precondition above can be written as follows:

```
log_out = DbcMethod.new(:log_out)
log_out.precondition = Proc.new { @is_logged_in }
log_out.postcondition = Proc.new { @is_logged_in = false }
```

Listing 4.2: The specification of an example log out method in Communiqué.

```
registerCourse.precondition = Proc.new do
  not self.courses.include?(registerCourse.parameters[:c])
end
```

There is a caveat though. While the precondition block of a method in Communiqué is expected to be, just like in other contract languages, an expression that evaluates to true or false, the postcondition of the method is expected to be an expression (or a set of expressions) that actually changes something in the state, typically by assigning values to instance variables. If the block does not actually change some thing in the state, $s' = \gamma(s, m) = s$. That is to say, although Communiqué will execute the postcondition block of an applicable method, if the block does not actually change some thing in the state, it will not contribute to solving the planning problem at hand.

Continuing with the example, the `log_out` method with its precondition and postcondition can be specified in Communiqué as shown in Listing 4.2. This method is, in a sense, the inverse of the `log_in` method.

After specifying the methods with their preconditions and postconditions, the methods should be added to the `User` instance as follows:

```
user_instance.add_dbc_methods(log_in, log_out)
```

Assume now that the use case that is to be realized with a sequence diagram is about a user who is initially logged out and wants to log in. The use case can be set up as follows:

```
login_use_case = DbcUseCase.new('Login')
login_use_case.dbc_instances << user_instance
```

Sender Name	Method Name	Parameters Names	Receiver Name
`<Actor>`	`log_in`	—	`user`

Table 4.2: The solution that Communiqué reports for the example Login use case.

```
login_use_case.postconditions = {
  'user' => Proc.new { @is_logged_in }
}
```

The above creates a use case and give it a name, adds the logged out `User` to it, and specify that the `User` should be logged in at the end of the use case. In Communiqué, use case postconditions are specified as a Ruby `Hash`, which is a set of key-value pairs, where the key is an instance name and the value is a code block representing the conditions on that instance.

Using the above models, the planning problem can be set up for a planner and then the planner can be asked to solve the problem as follows:

```
planner = Planner.new
planner.set_up_initial_state(login_use_case)
planner.goals = use_case.postconditions
planner.solve
```

If the planner finds a solution, it returns an `Array` of `Hash`s, where each `Hash` represents a message pass specifying the message's sender, its name, its parameters (if any), and its receiver. Table 4.2 shows the solution for the Login use case in tabular format. The output is simple and consists of a single message pass because the example itself is simple. Of course, the output can be much longer. For an example of a more complex output, refer to Table 5.1 on page 66.

CHAPTER 5

EXPERIMENTS AND RESULTS

A number of different experiments were carried out to validate the proposed approach of using automated planning to plan messages in sequence diagrams that was discussed in Chapter 4, and to assess the capabilities and limitations of Communiqué, its implementation, that was discussed in Section 4.5. Each of the following sections provides details about the purpose of the experiment, its setup, and its results. The full source code of the experiments scripts is available online along with the source code of Communiqué.[1]

It must be mention here that the UML models (especially the class diagrams) that were used in the experiments were not originally designed following the Design by Contract approach. That is, the operations of the classes did not have contracts (i.e. preconditions and postconditions). Because of that, we had to add the contracts to the models as if we were designing the models using Design by Contract. Sometimes, adding a contract to a method of a class entailed adding a new attribute to that class. Most of the time, these added attributes were high-level boolean flags, such as an `is_refreshed` boolean attribute of a

[1]https://github.com/ysulaiman/communique

`WatchDisplay` class.

Also, note that Communiqué does not currently generate actual diagrams: it generates command-line outputs that represent the sequence of message passes of the solution sequence diagrams. The command-line output includes the necessary information for constructing a sequence diagram: the message's sender, its name, its parameters (if any), and its receiver for each message pass. For example, Table 5.1 on page 66 shows the output of Communiqué, in tabular format, for one of the experiments. For conciseness and clarity, the command-line outputs will not be shown for the other experiments. Instead, the sequence diagrams that correspond to those outputs will be shown directly.

5.1 Experiment 1: Simple Diagrams

The purpose of this experiment was essentially to see if Communiqué can produce simple sequences of message passes that match simple sequence diagrams designed by fourth-year undergraduate software engineering students.

The UML models used in this experiment are based upon the Properties Management System by Aman et al. [33]. The system provides a solution that handles aspects of property management, such as advertising properties to customers and automating the maintenance request process.

For this experiment, five use cases were selected: add property, modify property, select featured property, delete property, and modify announcement. Each of these use cases were realized by a single sequence diagram, for a total of five sequence diagrams. Figures 5.1 through 5.5 show the sequence diagrams as they were presented in the Software Requirements Specification of the Properties Management System [33].

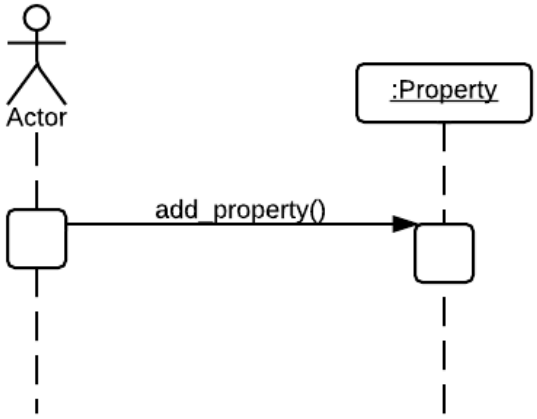

Figure 5.1: The sequence diagram for the add property use case.

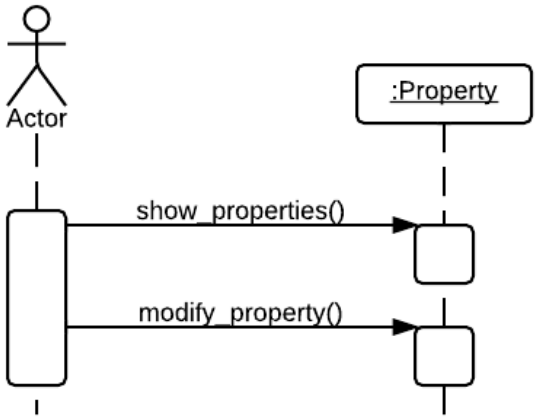

Figure 5.2: The sequence diagram for the modify property use case.

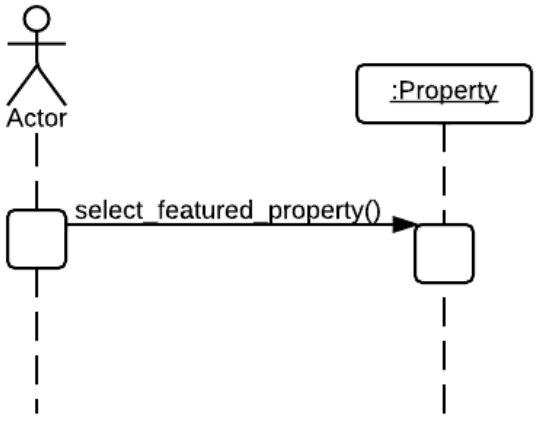

Figure 5.3: The sequence diagram for the select featured property use case.

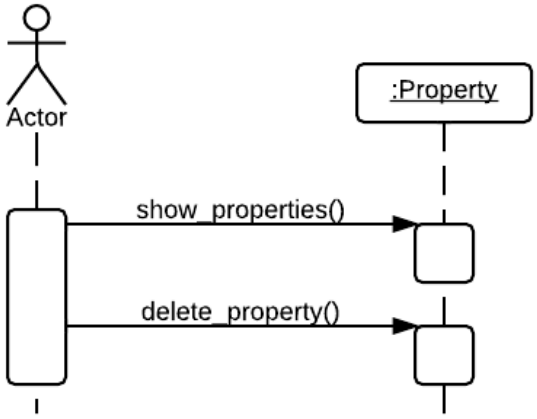

Figure 5.4: The sequence diagram for the delete property use case.

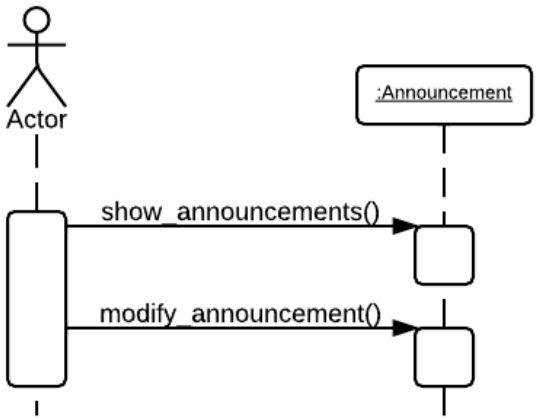

Figure 5.5: The sequence diagram for the modify announcement use case.

5.1.1 Inputs

Listing 5.1 shows the Ruby code that is used to set up the inputs for this experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. That is, it effectively creates an instance of the class digram depicted in Figure 5.6, which is the part of the system's class diagram relevant to the five selected use cases. This instance is used as the initial object diagram for all of the five use cases. The code also sets up the preconditions and postconditions of each of the five use cases.

Listing 5.1: The Inputs for Experiment 1

```
# Account and its methods and contracts.
account_instance = DbcObject.new('account', :Account, {
  :@is_logged_in => false
})
log_in = DbcMethod.new('log_in')
log_in.precondition = Proc.new { !@is_logged_in }
log_in.postcondition = Proc.new { @is_logged_in = true }
log_out = DbcMethod.new('log_out')
log_out.precondition = Proc.new { @is_logged_in }
log_out.postcondition = Proc.new { @is_logged_in = false }
account_instance.add_dbc_methods(log_in, log_out)

# Property and its methods and contracts.
property_instance = DbcObject.new('property', :Property, {
  :@account => account_instance,
  :@properties_are_listed => false,
  :@is_modified => false,
  :@manager_is_notified => false,
  :@is_featured => false,
```

```
  :@deleted => false,
  :@is_added => false
})
show_properties = DbcMethod.new('show_properties')
show_properties.precondition = Proc.new { @account.is_logged_in
    && !@deleted }
show_properties.postcondition = Proc.new {
   @properties_are_listed = true }
modify_property = DbcMethod.new('modify_property')
modify_property.precondition = Proc.new do
  @account.is_logged_in && @properties_are_listed && !@deleted
end
modify_property.postcondition = Proc.new do
  @is_modified = true
  @manager_is_notified = true
end
select_featured_property = DbcMethod.new('
   select_featured_property')
select_featured_property.precondition = Proc.new do
  @account.is_logged_in && !@is_featured && !@deleted
end
select_featured_property.postcondition = Proc.new {
   @is_featured = true }
unselect_featured_property = DbcMethod.new('
   unselect_featured_property')
unselect_featured_property.precondition = Proc.new do
  @account.is_logged_in && @is_featured && !@deleted
end
unselect_featured_property.postcondition = Proc.new {
```

```
   @is_featured = false }
delete_property = DbcMethod.new('delete_property')
delete_property.precondition = Proc.new do
  @account.is_logged_in && @properties_are_listed && !@deleted
end
delete_property.postcondition = Proc.new do
  @deleted = true
  @manager_is_notified = true
end
add_property = DbcMethod.new('add_property')
add_property.precondition = Proc.new { @account.is_logged_in }
add_property.postcondition = Proc.new do
  @is_added = true
  @manager_is_notified = true
end
property_instance.add_dbc_methods(show_properties,
   modify_property, select_featured_property,
   unselect_featured_property, delete_property, add_property)

# Announcement and its methods and contracts.
announcement_instance = DbcObject.new('announcement', :
   Announcement, {
  :@account => account_instance,
  :@announcements_are_listed => false,
  :@is_modified => false,
  :@manager_is_notified => false,
})
show_announcements = DbcMethod.new('show_announcements')
show_announcements.precondition = Proc.new { @account.
```

```
   is_logged_in }
show_announcements.postcondition = Proc.new {
   @announcements_are_listed = true }
modify_announcement = DbcMethod.new('modify_announcement')
modify_announcement.precondition = Proc.new do
  @account.is_logged_in && @announcements_are_listed
end
modify_announcement.postcondition = Proc.new do
  @is_modified = true
  @manager_is_notified = true
end
announcement_instance.add_dbc_methods(show_announcements,
   modify_announcement)

# The add property use case.
add_property_use_case = DbcUseCase.new('Add␣Property')
add_property_use_case.dbc_instances << account_instance <<
   property_instance << announcement_instance
account_instance.is_logged_in = true
add_property_use_case.postconditions = {'property' => Proc.new
   { @is_added && @manager_is_notified }}

# The modify property use case.
modify_property_use_case = DbcUseCase.new('Modify␣Property')
modify_property_use_case.dbc_instances << account_instance <<
   property_instance << announcement_instance
account_instance.is_logged_in = true
modify_property_use_case.postconditions = {'property' => Proc.
   new { @is_modified && @manager_is_notified }}
```

```
# The select featured property use case.
select_featured_property_use_case = DbcUseCase.new('Select␣
   Featured␣Property')
select_featured_property_use_case.dbc_instances <<
   account_instance << property_instance <<
   announcement_instance
account_instance.is_logged_in = true
select_featured_property_use_case.postconditions = {'property'
   => Proc.new { @is_featured }}

# the delete property use case.
delete_property_use_case = DbcUseCase.new('Delete␣Property')
delete_property_use_case.dbc_instances << account_instance <<
   property_instance << announcement_instance
account_instance.is_logged_in = true
delete_property_use_case.postconditions = {'property' => Proc.
   new { @deleted && @manager_is_notified }}

# The modify announcement use case.
modify_announcement_use_case = DbcUseCase.new('Modify␣
   Announcement')
modify_announcement_use_case.dbc_instances << account_instance
   << property_instance << announcement_instance
account_instance.is_logged_in = true
modify_announcement_use_case.postconditions = {'announcement'
   => Proc.new { @is_modified && @manager_is_notified }}
```

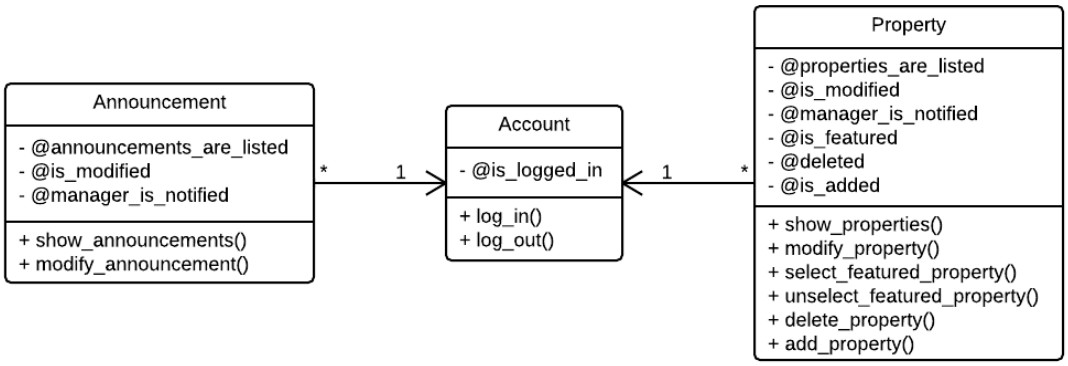

Figure 5.6: The class diagram used for Experiment 1.

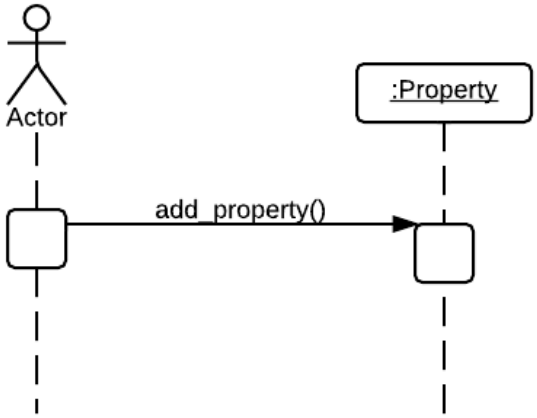

Figure 5.7: The generated sequence diagram for the add property use case.

5.1.2 Outputs

Figures 5.7 through 5.11 show the sequence diagrams that correspond to the outputs of Communiqué when using best-first and breadth-first search for this experiment. These generated sequence diagrams match the ones that were designed manually (Figures 5.1 through 5.5).

5.1.3 Discussion

For each of the five use cases, Communiqué was able to generate a solution. The generated solution differed depending on the control strategy used. When depth-first search was used, the generated solution was not optimal and included

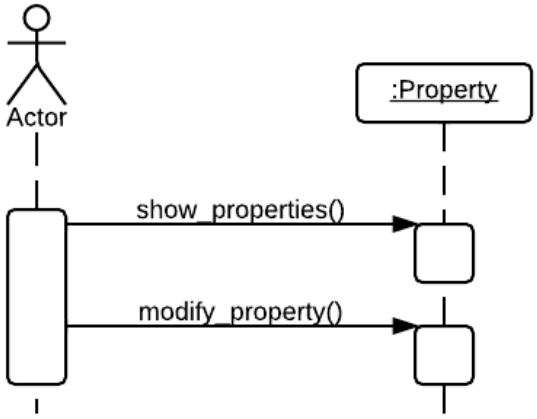

Figure 5.8: The generated sequence diagram for the modify property use case.

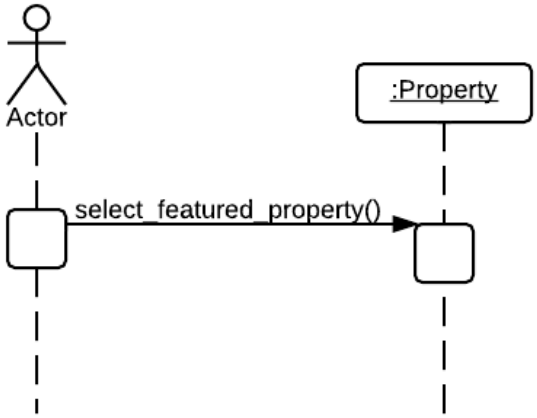

Figure 5.9: The generated sequence diagram for the select featured property use case.

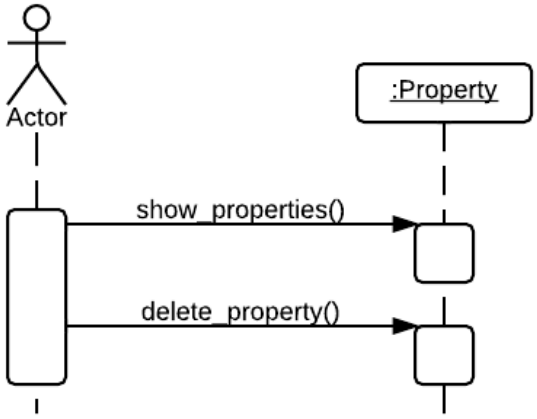

Figure 5.10: The generated sequence diagram for the delete property use case.

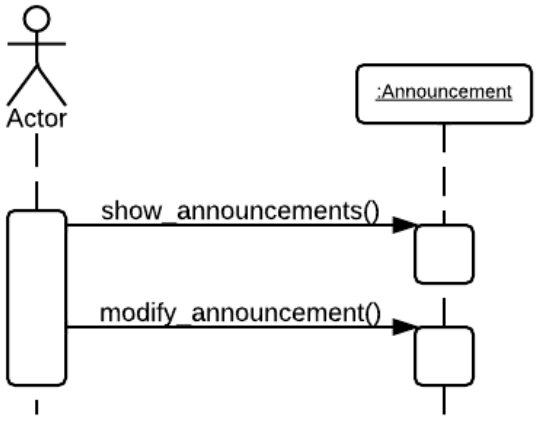

Figure 5.11: The generated sequence diagram for the modify announcement use case.

message passes that are irrelevant to the use case. When breadth-first or best-first was used, the generated solution was optimal and the sequence of message passes matched that in the manually-designed sequence diagram. The difference between breadth-first and best-first was that best-first, as expected, explored less nodes than breadth-first.

5.2 Experiment 2: More Complex Diagrams

The sequence diagrams that were used in the Experiment discussed in Section 5.1 were simple: each sequence diagram involved one object and at most two message passes. We wanted to see if Communiqué can produce a more complex sequence of message passes that matches a complex sequence diagram designed by a software engineering expert. As target sequence diagrams for this experiment, three sequence diagrams were selected from three different systems: the `2Bwatch` system and the `ARENA` system of Bruegge and Dutoit [7], and the weather station system of Ian Sommerville [1].

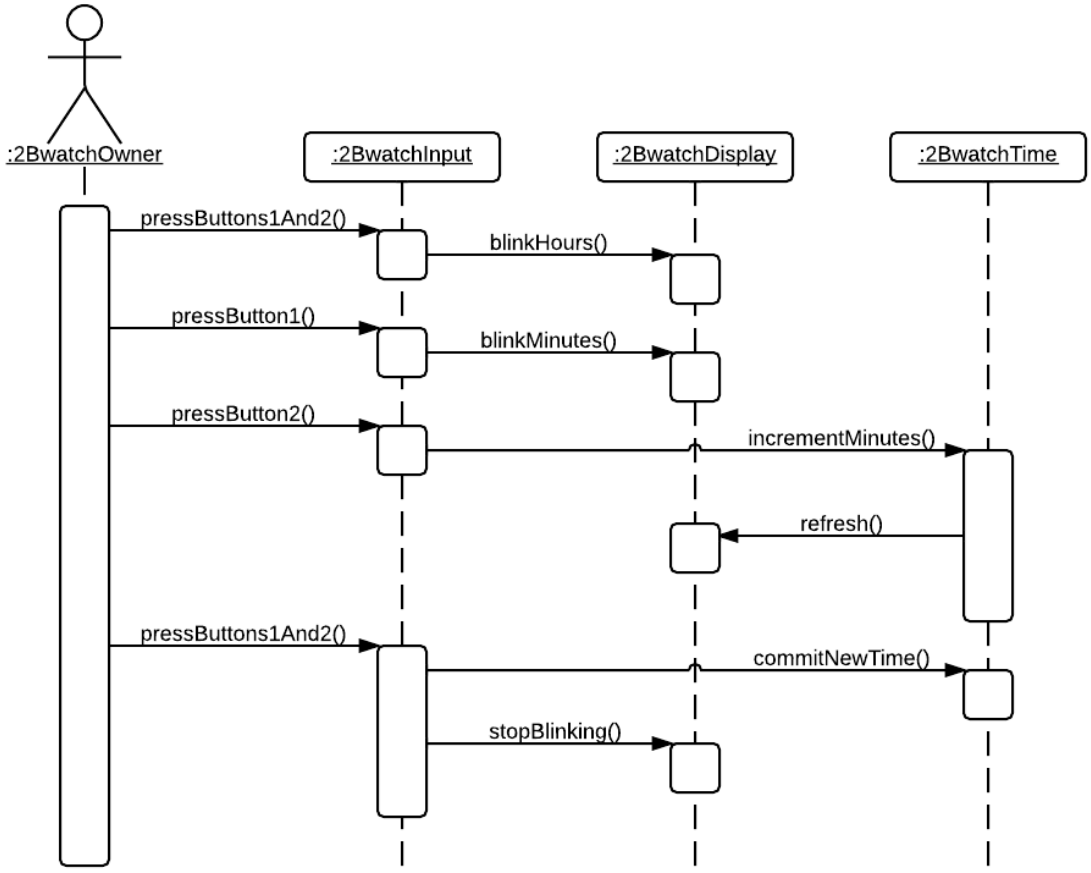

Figure 5.12: The sequence diagram for setting the time on a 2-button watch one minute ahead. This figure is a reproduction of Figure 2-34 of Bruegge and Dutoit [7, p. 94].

5.2.1 `2Bwatch` System

The first sequence diagram picked as a target for this experiment is the one for the `2Bwatch` system (also referred to as `SimpleWatch`) depicted in Figure 2-34 of *Object-Oriented Software Engineering Using UML, Patterns, and Java* by Bruegge and Dutoit [7, p. 94]. `2Bwatch` represents a 2-button watch, the user of which may either consult the time on the watch (with the `ReadTime` use case) or set it (with the `SetTime` use case). The sequence diagram in question, which is reproduced in Figure 5.12 for the reader's reference, is for an actor setting the watch one minute ahead.

Inputs

Listing 5.2 shows the Ruby code that is used to set up the inputs for the `2Bwatch` experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. In other words, it effectively creates an instance of the class digram depicted in Figure 5.13. This instance is used as the initial object diagram for the use case. The code also sets up the postconditions of the use case.

Listing 5.2: The Inputs for the `2Bwatch` Experiment

```
display = DbcObject.new('display', :TwoBWatchDisplay, {
  :@blinking => :none,
  :@is_refreshed => false,
  :@watch => nil
})
time = DbcObject.new('time', :TwoBWatchTime, {
  :@is_minutes_incremented => false,
  :@is_new_time_committed => false,
  :@watch => nil,
})
watch = DbcObject.new('watch', :TwoBWatchInput, {
  :@last_buttons_pressed => [],
  :@mode => :read_time,
  :@display => display,
  :@time => time
})
watch.boundary_object = true
display.watch = watch
time.watch = watch


# TwoBWatchDisplay methods
```

```
blink_hours = DbcMethod.new(:blink_hours)
blink_hours.precondition = Proc.new do
  @watch.mode == :set_time &&
    @watch.last_buttons_pressed == [:button_1, :button_2] &&
    @blinking == :none
end
blink_hours.postcondition = Proc.new { @blinking = :hours }
blink_minutes = DbcMethod.new(:blink_minutes)
blink_minutes.precondition = Proc.new do
  @watch.mode == :set_time &&
    @watch.last_buttons_pressed == [:button_1] &&
    @blinking == :hours
end
blink_minutes.postcondition = Proc.new { @blinking = :minutes }
stop_blinking = DbcMethod.new(:stop_blinking)
stop_blinking.precondition = Proc.new do
  @watch.time.is_new_time_committed &&
    @watch.last_buttons_pressed == [:button_1, :button_2] &&
    @blinking != :none
end
stop_blinking.postcondition = Proc.new { @blinking = :none }
refresh = DbcMethod.new(:refresh)
refresh.precondition = Proc.new { @watch.time.
   is_minutes_incremented }
refresh.postcondition = Proc.new { @is_refreshed = true }
display.add_dbc_methods(blink_hours, blink_minutes,
   stop_blinking, refresh)

# TwoBWatchTime methods
```

```
increment_minutes = DbcMethod.new(:increment_minutes)
increment_minutes.precondition = Proc.new do
  @watch.display.blinking == :minutes &&
    @watch.last_buttons_pressed == [:button_2]
end
increment_minutes.postcondition = Proc.new {
   @is_minutes_incremented = true }
commit_new_time = DbcMethod.new(:commit_new_time)
commit_new_time.precondition = Proc.new do
  @watch.last_buttons_pressed == [:button_1, :button_2] &&
    @watch.display.is_refreshed
end
commit_new_time.postcondition = Proc.new {
   @is_new_time_committed = true }
time.add_dbc_methods(increment_minutes, commit_new_time)

# TwoBWatchInput methods
press_button_1 = DbcMethod.new(:press_button_1)
press_button_1.precondition = Proc.new { @mode == :set_time }
press_button_1.postcondition = Proc.new do
  @last_buttons_pressed = [:button_1]
end
press_button_2 = DbcMethod.new(:press_button_2)
press_button_2.precondition = Proc.new { @mode == :set_time }
press_button_2.postcondition = Proc.new do
  @last_buttons_pressed = [:button_2]
end
press_buttons_1_and_2 = DbcMethod.new(:press_buttons_1_and_2)
press_buttons_1_and_2.precondition = Proc.new do
```

```
  @mode == :read_time || @display.is_refreshed
end
press_buttons_1_and_2.postcondition = Proc.new do
  @mode = case @mode
          when :read_time
            :set_time
          when :set_time
            :read_time
          end
  @last_buttons_pressed = [:button_1, :button_2]
end
watch.add_dbc_methods(press_button_1, press_button_2,
   press_buttons_1_and_2)

set_time_use_case = DbcUseCase.new('Set Time')
set_time_use_case.dbc_instances << display << time << watch
set_time_use_case.postconditions = {
  'display' => Proc.new { @blinking == :none },
  'time' => Proc.new { @is_new_time_committed },
  'watch' => Proc.new { @mode == :read_time }
}
```

Outputs and Discussion

Table 5.1 shows the initial output, in tabular format, of Communiqué for this experiment. Figure 5.14 shows the sequence diagram that corresponds to that output. This generated sequence diagram is different from the original manually-designed one: in the original sequence diagram, a `refresh` message is sent from the `2BwatchTime` instance; In the generated sequence diagram, the `refresh` message

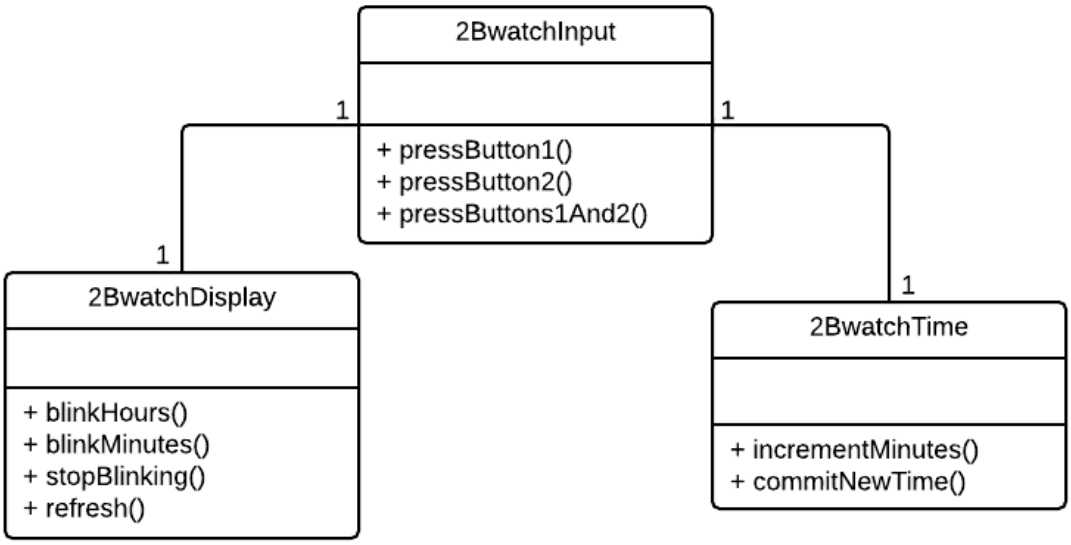

Figure 5.13: The class diagram of `2Bwatch`.

Sender Name	Method Name	Parameters Names	Receiver Name
<Actor>	press_buttons_1_and_2	—	watch
watch	blink_hours	—	display
<Actor>	press_button_1	—	watch
watch	blink_minutes	—	display
<Actor>	press_button_2	—	watch
watch	increment_minutes	—	time
watch	refresh	—	display
<Actor>	press_buttons_1_and_2	—	watch
watch	commit_new_time	—	time
watch	stop_blinking	—	display

Table 5.1: The initial solution that Communiqué reports for the `2Bwatch` experiment.

is sent from the `2BwatchInput` instance.

Further inspection of this difference revealed that there is an inconsistency in the original manually-designed models. The inconsistency is that there is no association in the class diagram (Figure 5.13) between `2BwatchTime` and `2BwatchDisplay` to allow instances of the former to send messages to instances of the latter as is the case in the sequence diagram (Figure 5.12). It may be the case that Bruegge and Dutoit did not show that particular association because they were concentrating in their Figure 2-2 [7, p. 66] on the main class, `2BwatchInput`,

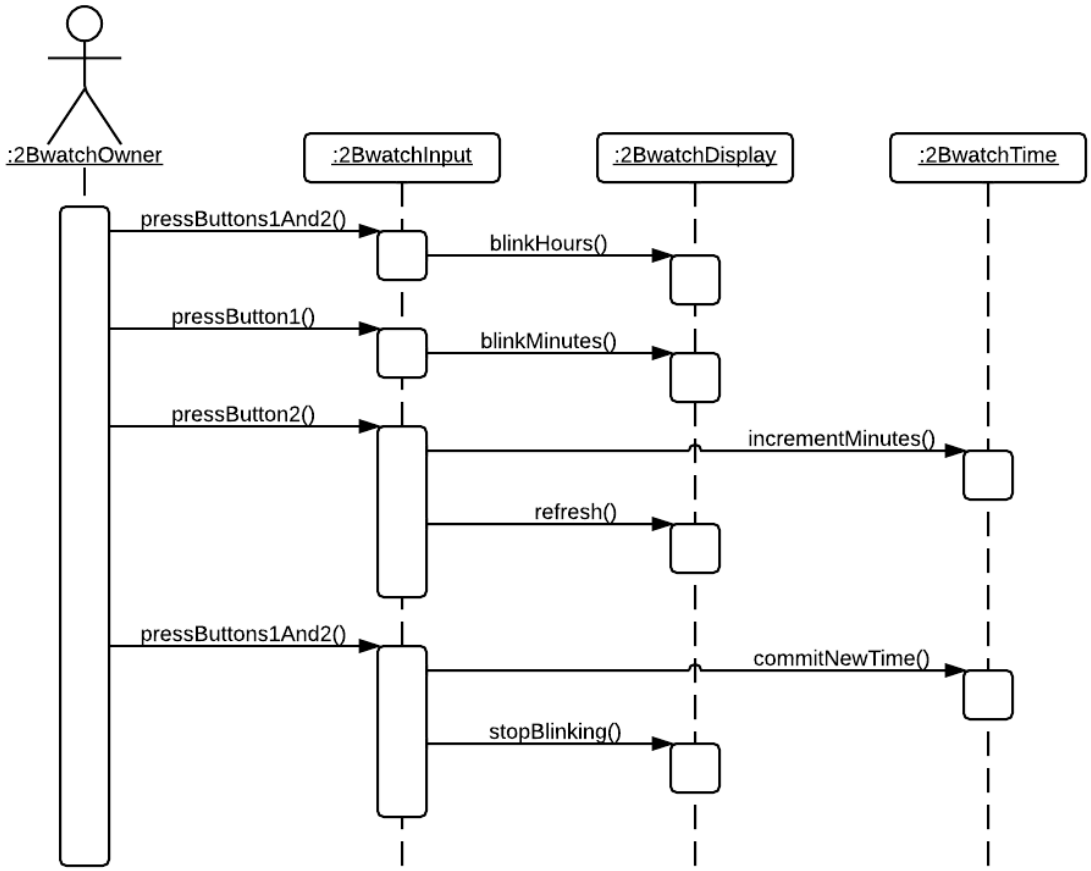

Figure 5.14: The sequence diagram corresponding to Communiqué's initial output.

and its associations only. Regardless, the fact is that there is only one class diagram related to the system under consideration, and from the sequence diagram's viewpoint, it is missing a necessary association. As there was no link from the `2BwatchTime` instance to the `2BwatchDisplay` instance, the planner of Communiqué did not consider the `2BwatchTime` as a valid sender of the `refresh` message and selected the `2BwatchInput` instance instead.

Figure 5.15 shows the sequence diagram corresponding to Communiqué's output after adding the missing association by linking the `2BwatchTime` instance to the `2BwatchDisplay` instance. This time, the planner selected the `2BwatchTime` as the sender of the `refresh` message, matching the textbook's sequence diagram, but it changed the sender of the `stopBlinking` message: instead of the `2BwatchInput` instance, the planner selected the `2BwatchTime` instance.

The reason behind this change is that after adding the missing association, the `stopBlinking` message now had two candidate senders. Using the Rule of Thumb 4 mentioned in Subsection 4.4.3, the planner selected the candidate sender that was activated most recently, namely, the `2BwatchTime` instance.

5.2.2 `ARENA` System

The second sequence diagram picked as a target for this experiment is the one for the `ARENA` system, a web-based system for organizing and conducting tournaments, depicted in Figure 5-26 of *Object-Oriented Software Engineering Using UML, Patterns, and Java* by Bruegge and Dutoit [7, p. 247]. The sequence diagram in question, which is reproduced in Figure 5.16 for the reader's reference, is for a tournament creation workflow of a tournament announcement use case.

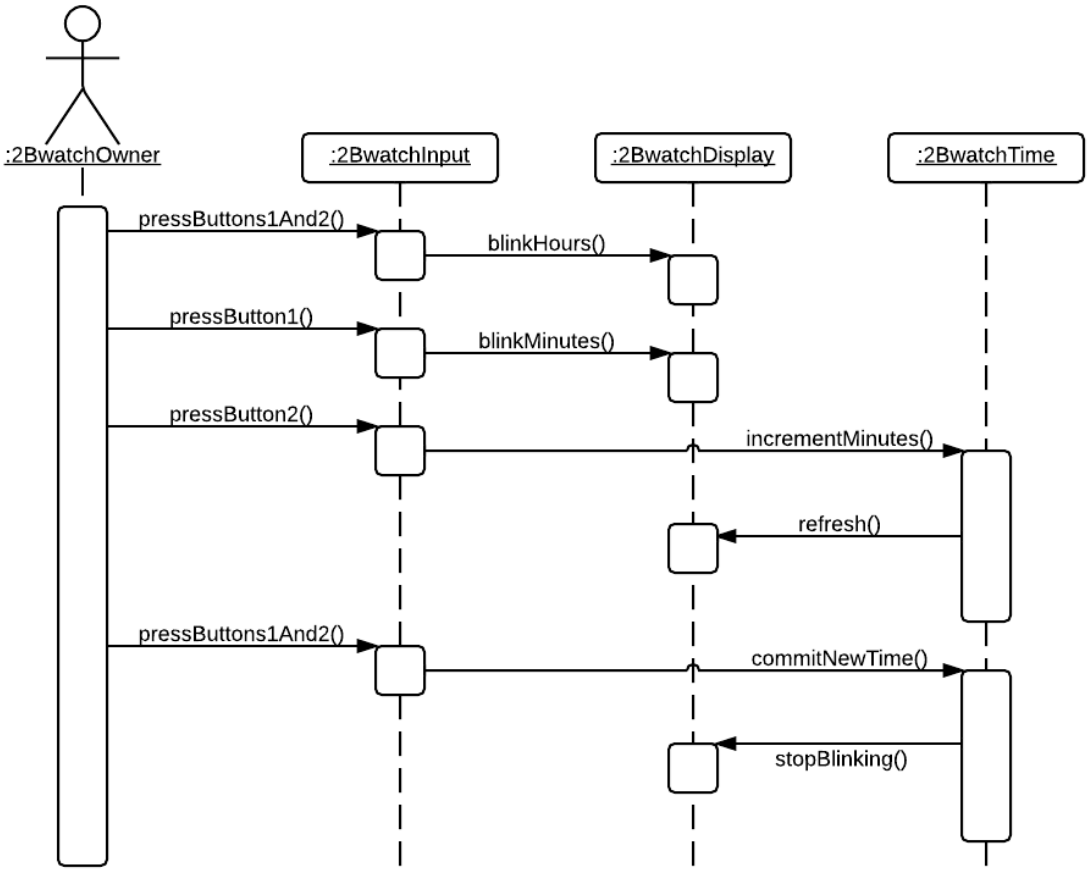

Figure 5.15: The sequence diagram corresponding to Communiqué's output after adding the missing association.

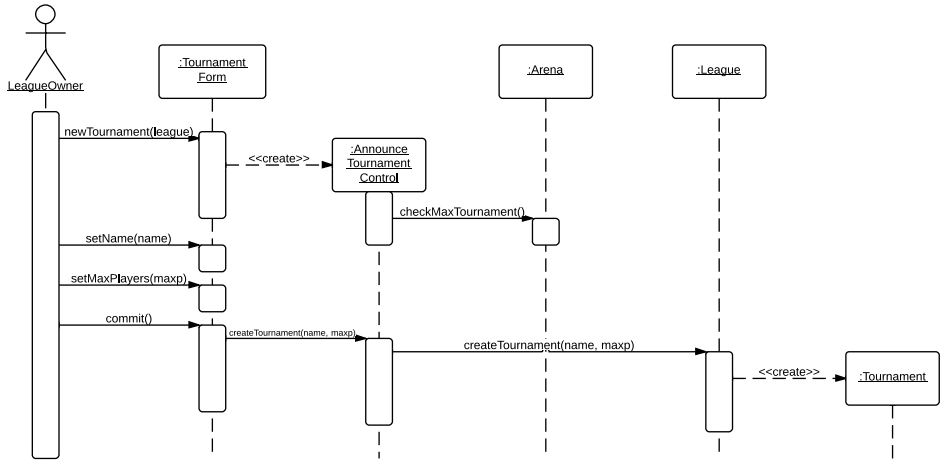

Figure 5.16: The sequence diagram for the tournament creation workflow of the Announce Tournament use case of **ARENA**. This figure is a reproduction of Figure 5-26 of Bruegge and Dutoit [7, p. 247].

Inputs

Listing 5.3 shows the Ruby code that is used to set up the inputs for the **ARENA** system experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. That is, it effectively creates an instance of the class digram depicted in Figure 5.17. This instance is used as the initial object diagram for the use case. The code also sets up the postconditions of the use case.

Listing 5.3: The Inputs for the **ARENA** Experiment

```
arena = DbcObject.new('arena', :Arena, {
  :@is_max_tournament_checked => false
})
league = DbcObject.new('league', :League, {
  :@tournaments => []
})
```

```
tournament = DbcObject.new('tournament', :Tournament, {
  :@league => league
})
tournament.dead = true
league.tournaments.push(tournament)
tournament_form = DbcObject.new('tournament_form', :
   TournamentForm, {
  :@is_name_set => false,
  :@is_max_players_set => false,
  :@is_committed => false,
  :@tournament => nil,
  :@announce_tournament_control => nil
})
tournament_form.boundary_object = true
announce_tournament_control =
  DbcObject.new('announce_tournament_control', :
     AnnounceTournamentControl, {
    :@is_create_tournament_request_recieved => false,
    :@arena => arena,
    :@tournament_form => tournament_form,
    :@league => league
  })
announce_tournament_control.dead = true
tournament_form.announce_tournament_control =
   announce_tournament_control

# TournamentForm methods.
new_tournament = DbcMethod.new(:new_tournament)
new_tournament.parameters = {league: :l}
```

```
new_tournament.precondition = Proc.new { true }
new_tournament.postcondition = Proc.new {} # ?
new_tournament.dependencies.push(announce_tournament_control.
   dbc_name)
set_name = DbcMethod.new(:set_name)
set_name.parameters = {name: :n}
set_name.precondition = Proc.new do
  state.get_instance_of(:Arena).is_max_tournament_checked
end
set_name.postcondition = Proc.new { @is_name_set = true }
set_max_players = DbcMethod.new(:set_max_players)
set_max_players.parameters = {maxp: :m}
set_max_players.precondition = Proc.new { @is_name_set }
set_max_players.postcondition = Proc.new { @is_max_players_set
   = true }
commit = DbcMethod.new(:commit)
commit.precondition = Proc.new { @is_name_set &&
   @is_max_players_set }
commit.postcondition = Proc.new { @is_committed = true }
tournament_form.add_dbc_methods(new_tournament, set_name,
   set_max_players, commit)


#AnnounceTournamentControl methods.
control_create_tournament = DbcMethod.new(:create_tournament)
control_create_tournament.parameters = {name: :n, maxp: :m}
control_create_tournament.precondition = Proc.new do
  self.tournament_form.is_committed
end
control_create_tournament.postcondition = Proc.new do
```

```
  @is_create_tournament_request_recieved = true
end
announce_tournament_control.add_dbc_methods(
   control_create_tournament)

# Arena methods.
check_max_tournament = DbcMethod.new(:check_max_tournament)
check_max_tournament.precondition = Proc.new do
  ! state.get_instance_of(:AnnounceTournamentControl).dead?
end
check_max_tournament.postcondition = Proc.new do
  @is_max_tournament_checked = true
end
arena.add_dbc_methods(check_max_tournament)

# League methods.
league_create_tournament = DbcMethod.new(:create_tournament)
league_create_tournament.parameters = {name: :n, maxp: :m}
league_create_tournament.precondition = Proc.new do
  state.get_instance_of(:AnnounceTournamentControl).
     is_create_tournament_request_recieved
end
league_create_tournament.postcondition = Proc.new { }
league_create_tournament.dependencies.push(tournament.dbc_name)
league.add_dbc_methods(league_create_tournament)

announce_tournament_use_case = DbcUseCase.new('Announce␣
   Tournament')
announce_tournament_use_case.dbc_instances << tournament_form
```

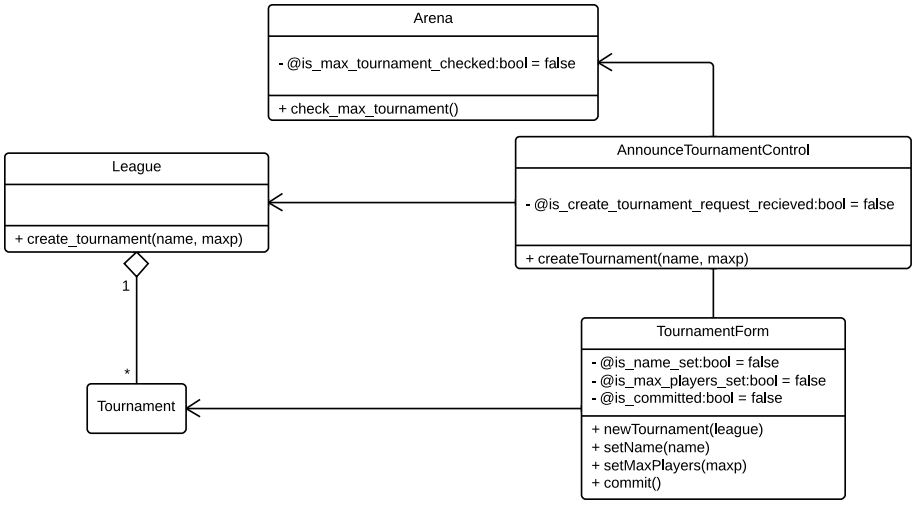

Figure 5.17: The class diagram of **ARENA**.

```
   << announce_tournament_control << arena << league <<
   tournament
announce_tournament_use_case.postconditions = {
  'tournament' => Proc.new { ! dead? }
}
```

Outputs and Discussion

Figure 5.18 shows the sequence diagram corresponding to the output of Communiqué for this experiment. This sequence diagram matches the original one that was designed manually (Figure 5.16).

5.2.3 Weather Station System

The third sequence diagram picked as a target for this experiment is the one for the weather station system depicted in Figure 14.13 of Ian Sommerville's *Soft-*

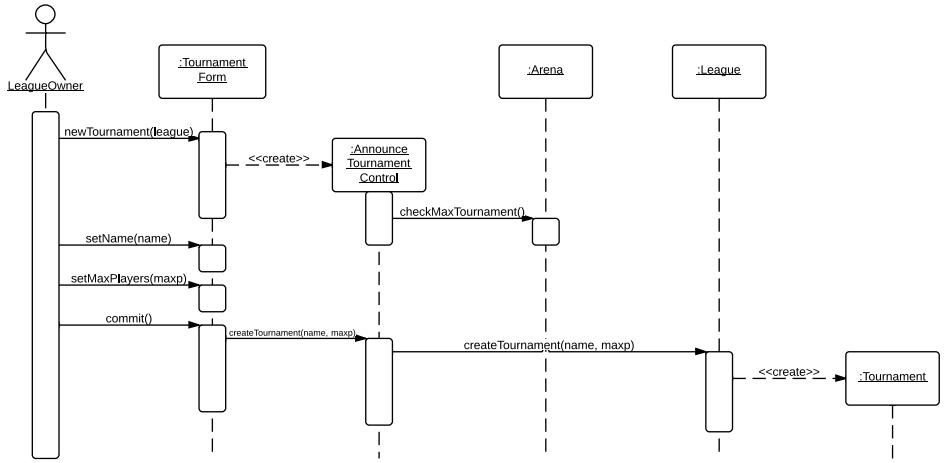

Figure 5.18: The sequence diagram corresponding to Communiqué's output for the tournament creation workflow.

ware Engineering[1]. The sequence diagram in question, which is reproduced in Figure 5.19 for the reader's reference, is for a data collection use case in which an external mapping system requests data from the weather station.

Inputs

Listing 5.4 shows the Ruby code that is used to set up the inputs for the weather station system experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. That is, it effectively creates an instance of the class digram depicted in Figure 5.20. This instance is used as the initial object diagram for the use case. The code also sets up the postconditions of the use case.

Listing 5.4: The Inputs for the Weather Station System Experiment

```
weather_data = DbcObject.new('weather_data', :WeatherData, {
  :@is_data_summarised => false
})
```

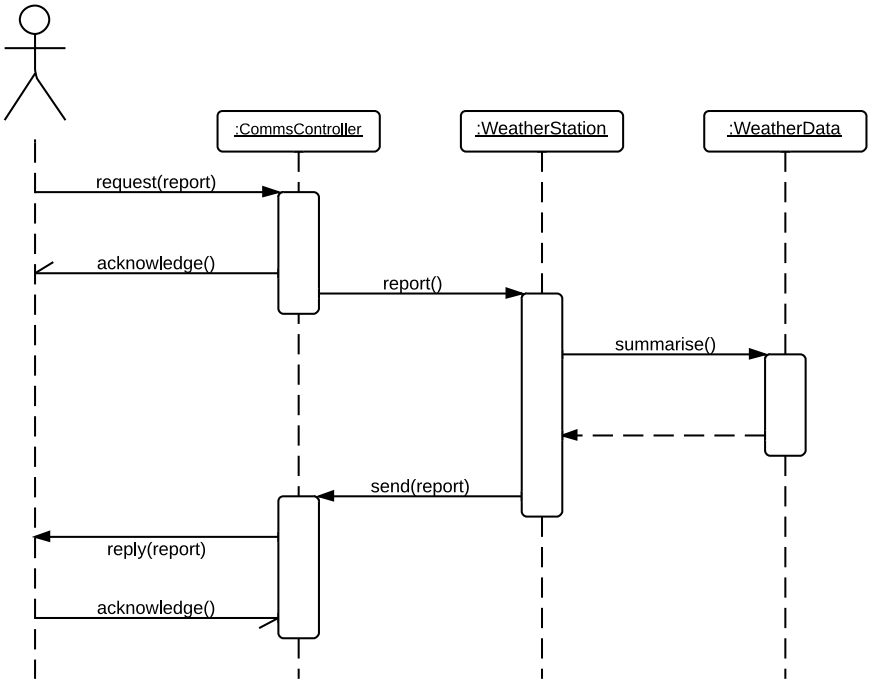

Figure 5.19: The sequence diagram for collecting data from a weather station. This figure is a reproduction of Figure 14.13 of Ian Sommerville's *Software Engineering* [1].

```
comms_controller = DbcObject.new('comms_controller', :
   CommsController, {
  :@is_request_received => false,
  :@is_report_sent => false,
  :@weather_station => nil
})
weather_station = DbcObject.new('weather_station', :
   WeatherStation, {
  :@identifier => :ws1,
  :@is_report_ready => false,
  :@comms_controller => comms_controller,
  :@weather_data => weather_data
})
comms_controller.weather_station = weather_station

# WeatherData methods
summarise = DbcMethod.new(:summarise)
summarise.precondition = Proc.new { ! @is_data_summarised }
summarise.postcondition = Proc.new { @is_data_summarised = true
    }
weather_data.add_dbc_methods(summarise)

# CommsController methods
request = DbcMethod.new(:request)
request.parameters = {report: :r}
request.precondition = Proc.new { true }
request.postcondition = Proc.new { @is_request_received = true
   }
send = DbcMethod.new(:send)
```

```
send.parameters = {report: :r}
send.precondition = Proc.new do
  state.get_instance_named('weather_data').is_data_summarised
     &&
    @weather_station.is_report_ready
end
send.postcondition = Proc.new { @is_report_sent = true }
comms_controller.add_dbc_methods(request, send)

# WeatherStation methods
report = DbcMethod.new(:report)
report.precondition = Proc.new { @comms_controller.
   is_request_received }
report.postcondition = Proc.new { @is_report_ready = true }
weather_station.add_dbc_methods(report)

collect_data_use_case = DbcUseCase.new('Collect␣Data')
collect_data_use_case.dbc_instances << comms_controller <<
   weather_station << weather_data
collect_data_use_case.postconditions = {
  'comms_controller' => Proc.new { @is_report_sent }
}
```

Outputs and Discussion

Figure 5.21 shows the sequence diagram corresponding to the output of Communiqué. The three greyed out messages in the diagram represent the message passes that Communiqué did not generate, mainly because it currently does not support sending messages back to the actor. Other than those three message passes, the

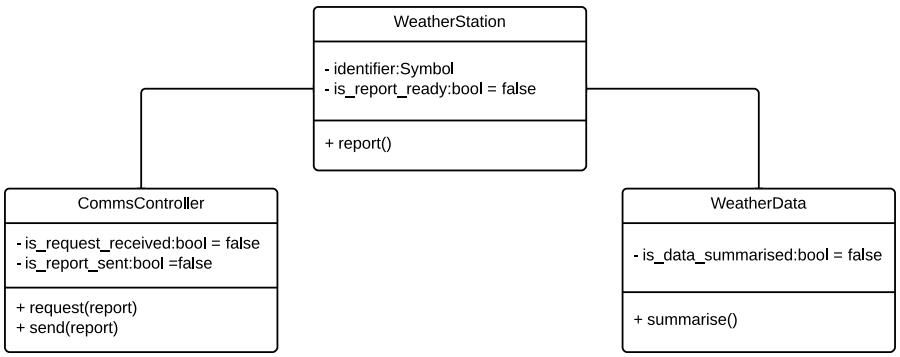

Figure 5.20: The class diagram of the weather station system.

generated sequence diagram matches the one that was designed manually.

5.3 Experiment 3: Effects of Large Class Diagrams

The class diagrams that were used in the previous experiments had a particular feature in common: they were small. A class diagram of a real-world software system will almost certainly be larger and contain more classes and methods. For a particular use case, many—if not most—of those methods will be irrelevant. We wanted to see how a large class diagram would affect the performance of Communiqué's planner.

The UML models used in this experiment are based upon the MeetingsMate system by Al Akel et al. [34]. The system provides a solution for scheduling and managing work meetings. For this experiment, we selected the Finalize Meeting use case of the MeetingsMate system. Figure 5.22 shows the manually-designed sequence diagram that realized that use case.

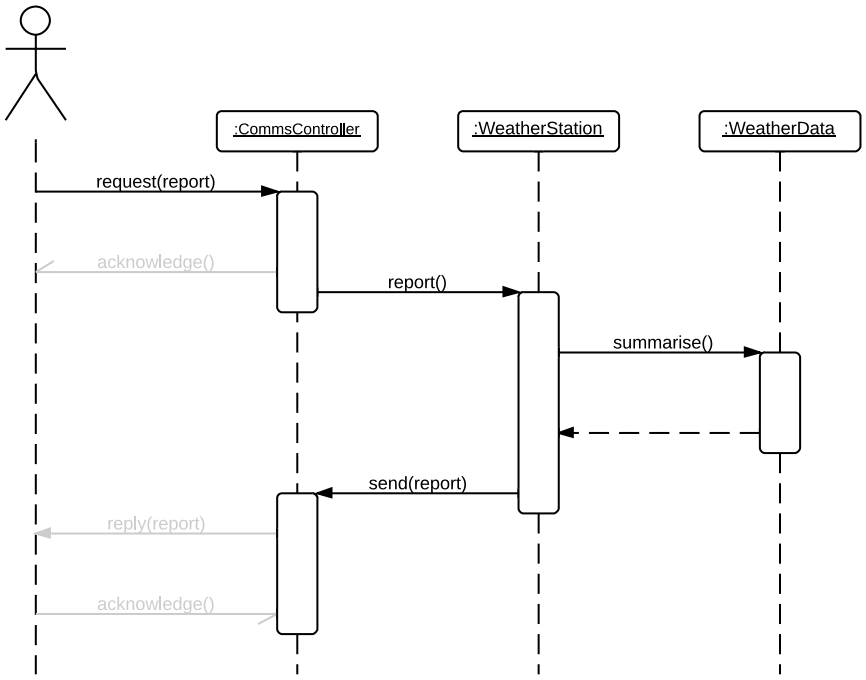

Figure 5.21: The sequence diagram corresponding to Communiqué's output for the data collection use case of the weather station system.

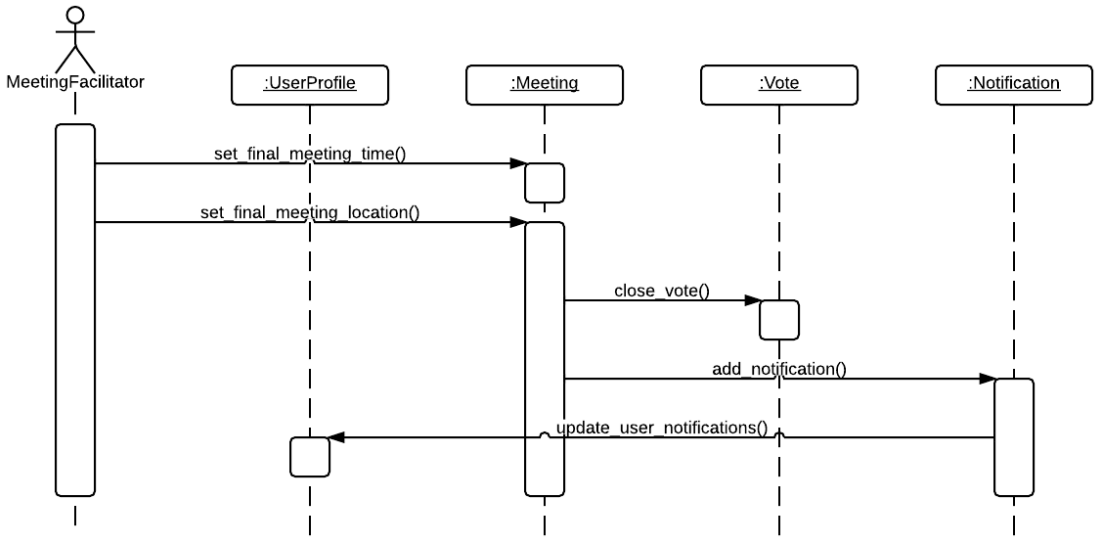

Figure 5.22: The sequence diagram for the Finalize Meeting use case of the MeetingsMate system [34].

5.3.1 Inputs and Experimental Setup

Listing 5.5 shows the Ruby code that is used to set up the inputs for this experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. In other words, it effectively creates an instance of the class digram depicted in Figure 5.23, which is the part of the system's class diagram that is relevant to the selected use case. This instance is used as the initial object diagram for the use case. The code also sets up the use case's postconditions.

Listing 5.5: The Inputs for the MeetingsMate Experiment

```
notification_instance = DbcObject.new('notification', :
   Notification, {
  :@meeting => nil,
  :@user_profile => nil
})
vote_instance = DbcObject.new('vote', :Vote, {
  :@is_closed => false
})
meeting_instance = DbcObject.new('meeting', :Meeting, {
  :@is_final_meeting_time_set => false,
  :@is_final_meeting_location_set => false,
  :@vote => vote_instance,
  :@notification => notification_instance
})
user_profile_instance = DbcObject.new('user_profile', :
   UserProfile, {
  :@is_logged_in => true,
  :@notifications => [],
  :@meeting => meeting_instance
```

```
})

# UserProfile methods
update_user_notifications = DbcMethod.new(:
   update_user_notifications)
update_user_notifications.precondition = Proc.new do
  state.get_instance_of(:Notification).meeting &&
    @meeting.is_final_meeting_time_set &&
    @meeting.is_final_meeting_location_set
end
update_user_notifications.postcondition = Proc.new do
  @notifications << state.get_instance_of(:Notification)
end
user_profile_instance.add_dbc_methods(update_user_notifications
   )

# Notification methods
add_notification = DbcMethod.new(:add_notification)
add_notification.precondition = Proc.new do
  state.get_instance_of(:Meeting) &&
    state.get_instance_of(:Meeting).vote.is_closed
end
add_notification.postcondition = Proc.new do
  @meeting = state.get_instance_of(:Meeting)
  @user_profile = state.get_instance_of(:UserProfile)
end
notification_instance.add_dbc_methods(add_notification)

# Vote methods
```

```
close_vote = DbcMethod.new(:close_vote)
close_vote.precondition = Proc.new do
  ! @is_closed &&
    state.get_instance_of(:Meeting).is_final_meeting_time_set
       &&
    state.get_instance_of(:Meeting).
       is_final_meeting_location_set
end
close_vote.postcondition = Proc.new { @is_closed = true }
vote_instance.add_dbc_methods(close_vote)

# Meeting methods
set_final_meeting_location = DbcMethod.new(:
   set_final_meeting_location)
set_final_meeting_location.precondition = Proc.new {
   @is_final_meeting_time_set }
set_final_meeting_location.postcondition = Proc.new do
  @is_final_meeting_location_set = true
end
set_final_meeting_time = DbcMethod.new(:set_final_meeting_time)
set_final_meeting_time.precondition = Proc.new { true }
set_final_meeting_time.postcondition = Proc.new do
  @is_final_meeting_time_set = true
end
meeting_instance.add_dbc_methods(set_final_meeting_location,
   set_final_meeting_time)

finalize_meeting_use_case = DbcUseCase.new('Finalize␣Meeting')
finalize_meeting_use_case.dbc_instances <<
```

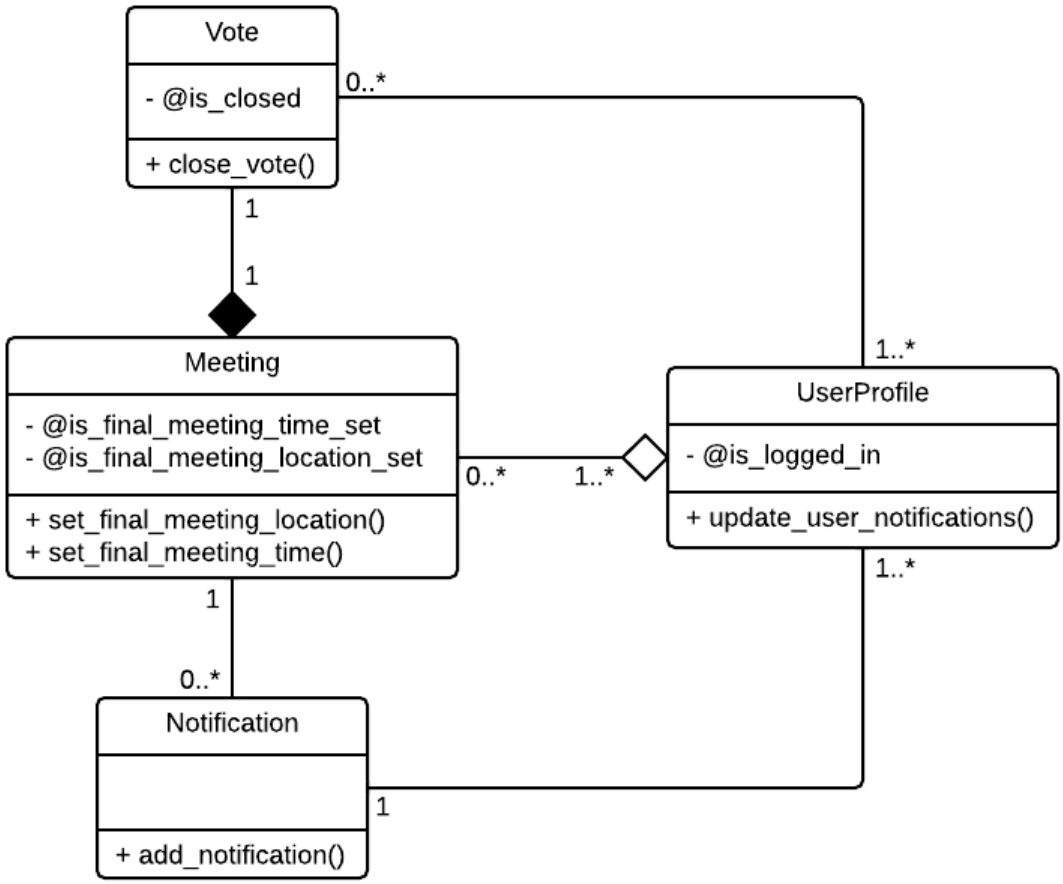

Figure 5.23: The class diagram used for Experiment 3.

```
   notification_instance << vote_instance << meeting_instance
   << user_profile_instance
finalize_meeting_use_case.postconditions = {
  'meeting' => Proc.new { @is_final_meeting_time_set &&
     @is_final_meeting_location_set },
  'vote' => Proc.new { @is_closed },
  'notification' => Proc.new { ! @meeting.nil? },
  'user_profile' => Proc.new { @notifications.include? state.
     get_instance_named('notification') }
}
```

To avoid having to redesign a large class diagram using Design by Contract, a `NoiseGenerator` class that can generate m noise methods was implemented. A noise method is a method that is always applicable (i.e. its precondition is `true`) and that does not change the state at all (i.e. its postcondition is empty). In other words, noise methods are irrelevant methods that do not contribute towards satisfying the postconditions of the use case at hand. By generating m noise methods, adding them to a dummy object, and adding that object to the class diagram, a large class diagram can be, in effect, simulated.

For this particular experiment, the number of noise methods m were varied from 0 to 10. To study the effects of noise methods on the performance of the three search algorithm (depth-first, breadth-first, and best-first), the experiment script effectively created 30 search spaces for each number of noise methods $m = 0, 1, 2, \cdots, 10$. This was achieved by shuffling the current set of methods collected by the planner just before it began to solve the problem at hand. To generate the same sequence of search spaces for the different search algorithms and to make the results reproducible, the pseudo-random number generator that was used for the shuffling was initialized with a fixed seed.

Three quantities were used to measure the performance of Communiqué's planner relative to the number of noise methods:

1. The number of explored nodes, or more precisely, the number of states that the planner checked to see if they satisfy the use case postconditions (*number of goal tests* for short).

2. The execution (real) time (as reported by the `Benchmark` module of Ruby) taken by the planner to solve the problem.

3. The length of the generated plan (that is, the number of message passes of the generated sequence diagram).

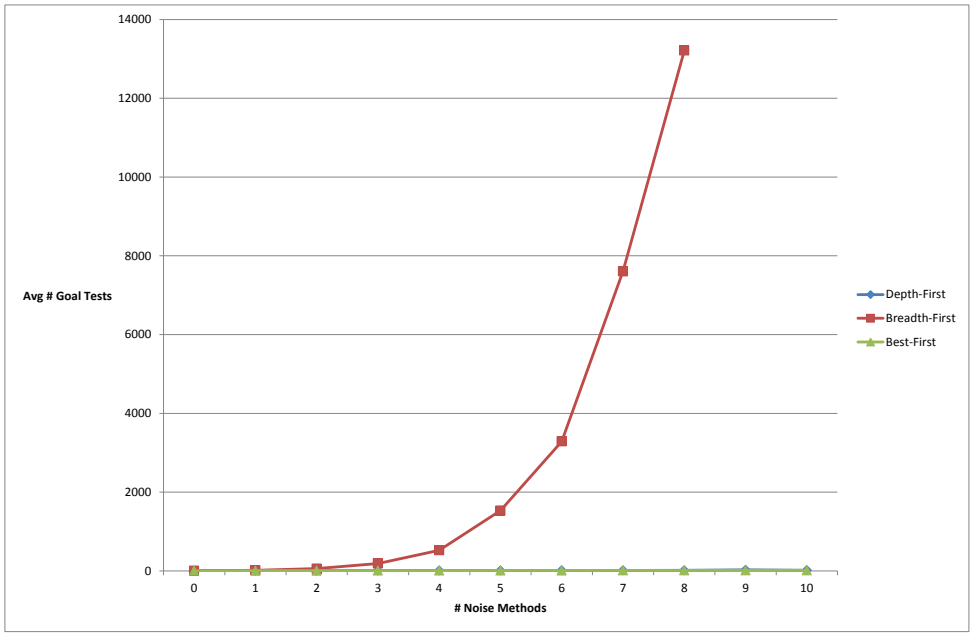

Figure 5.24: The effect of noise methods on the number of goal tests.

5.3.2 Results

Figure 5.24 shows the effects of noise methods on the number of goal tests. As the number of noise methods m increased, the average number of states breadth-first explored grew exponentially. Figure 5.25 compares the performance of depth-first and best-first more clearly. Depth-first and best-first explored, on average, a similar number of states and did not exhibit the exponential growth of breadth-first.

Figure 5.26 shows the effect of noise methods on the execution real time of the three search algorithms. Since this time is directly influenced by the number of explored states, the average time breadth-first took to find the optimal solution grew exponentially as the number of noise methods m increased. Depth- and best-first are compared more clearly in Figure 5.27. Depth-first generally took less time than best-first to find a solution. The extra time best-first took to find

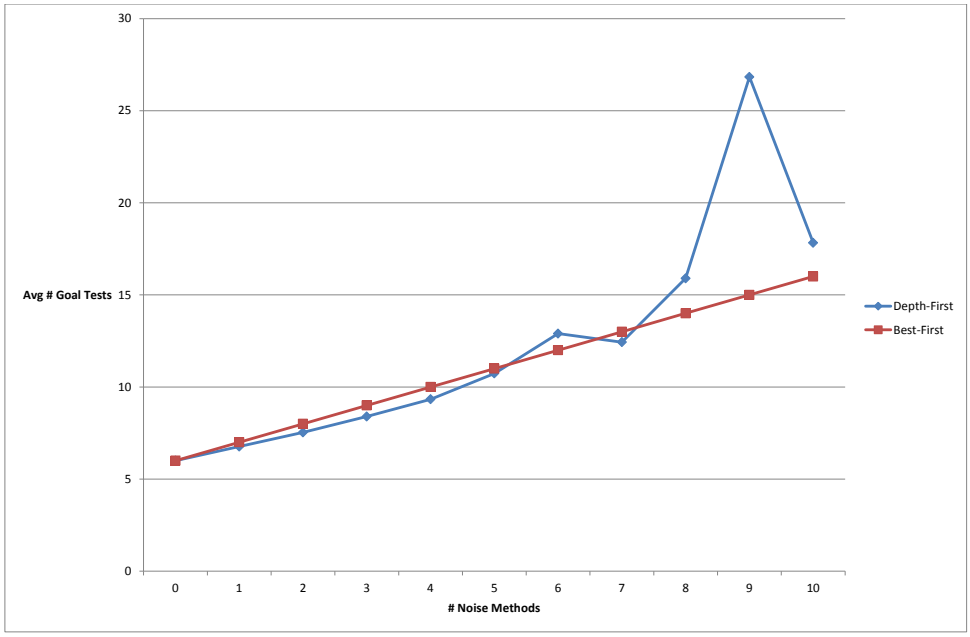

Figure 5.25: The effect of noise methods on the number of goal tests for depth- and best-first.

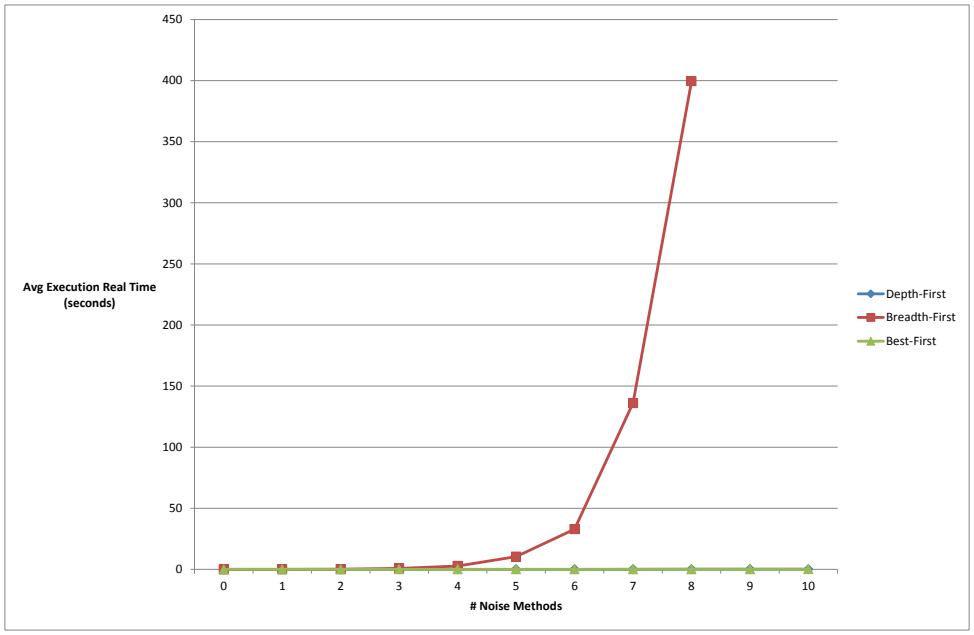

Figure 5.26: The effect of noise methods on the execution real time.

the optimal solution was mainly spent in calculating the heuristic function $h(n)$ discussed in Subsection 4.4.2.

Figure 5.28 shows the effect of noise methods on the length of the generated plan. While breadth-first and best-first always returned the optimal plan (of length 5), the average length of the plan returned by depth-first grew linearly with the number of noise methods m. That is, the output of Communiqué when using breadth-first and best-first always matched the manually-designed sequence diagram shown in Figure 5.22. On the other hand, the outputs obtained using depth-first contained extra noise-method invocations scattered throughout the sequence diagram, and the number of these invocations increased with the increase of noise methods.

To summarize, depth-first generally explored less states and spent less time than both breadth-first and best-first to find a solution, but the length of that

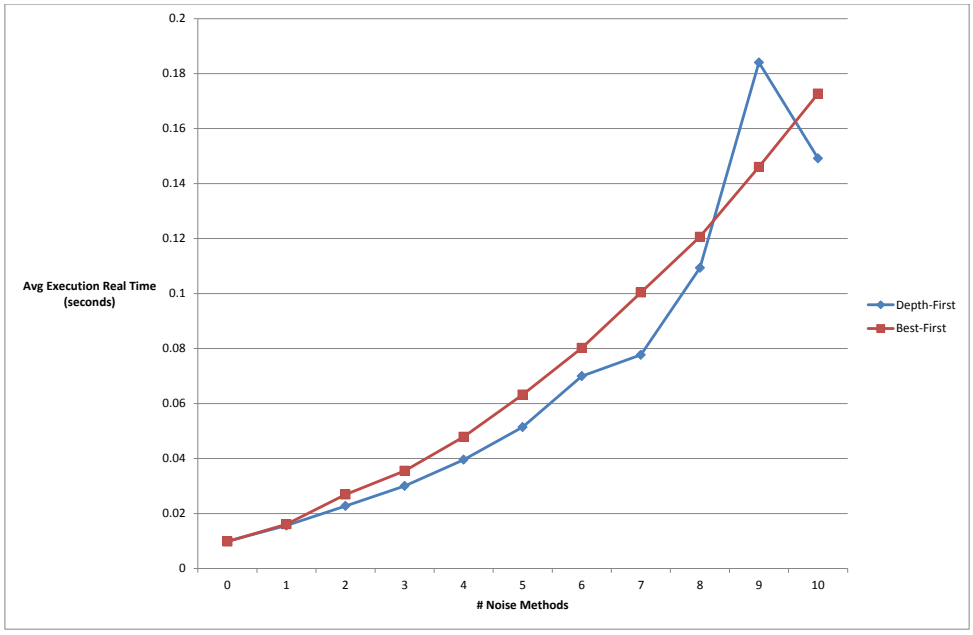

Figure 5.27: The effect of noise methods on the execution real time of depth- and best-first.

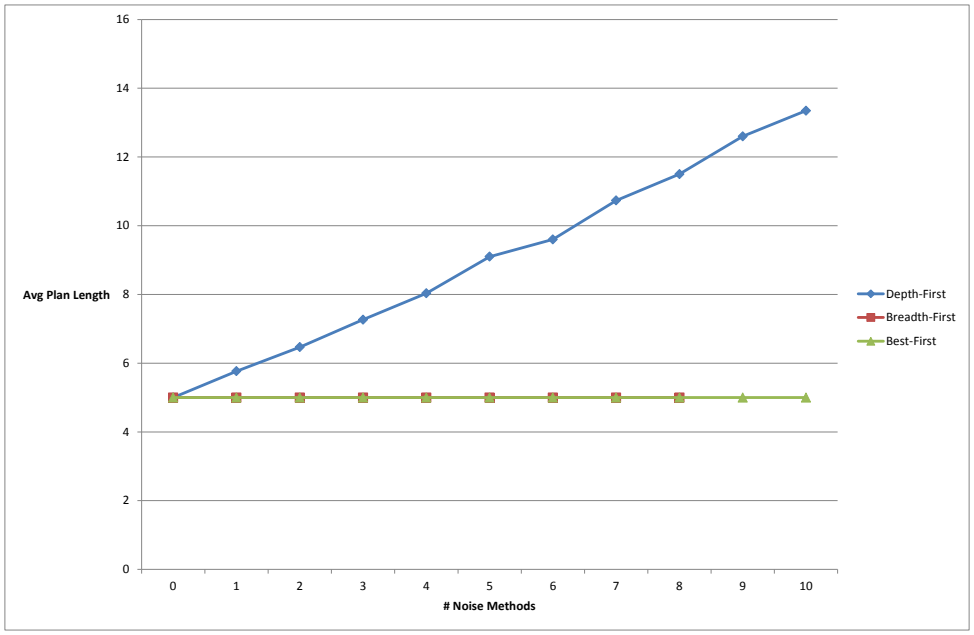

Figure 5.28: The effect of noise methods on the plan length.

solution increased linearly with the number of noise methods. Breadth-first was able to always find the optimal solution, but the number of states it had to explore (and, as a consequence, the time it had to spend) to find that optimal solution grew exponentially with the number of noise methods. Best-first was able to strike a balance by always finding the optimal solution while exploring a number of states (and spending an amount of time) that is close to that of depth-first and that increased linearly with the number of noise methods.

5.4 Experiment 4: Non-Optimality of Best-First Search

The purpose of this experiment is to demonstrate the non-optimality of Communiqué's best-first search. As Subsection 4.4.2 explained, because the heuristic

function $h(n)$ that best-first search uses is not admissible (that is, it may overestimate the true cost of reaching a goal), it may cause the planner to return a non-optimal solution.

5.4.1 Inputs

For demonstration purposes, this experiment uses a simple contrived example. Listing 5.6 shows the Ruby code that is used to set up the inputs for this experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. Figure 5.29 shows the class diagram of which an instance is created by the code. The preconditions and postconditions of the methods are deliberately set up so that there are two solutions: one consists of two message passes (Figure 5.30), while the other consists of three (Figure 5.31). The code shown in Listing 5.6 also sets up the postconditions of the use case so that the goal becomes getting the controller to satisfy the three objects it contains (i.e. setting the `is_satisfied` boolean attribute to true for each of the three objects).

Listing 5.6: The Inputs for Experiment 4

```
object_1 = DbcObject.new('object_1', :Object, {
  :@is_satisfied => false
})
object_2 = DbcObject.new('object_2', :Object, {
  :@is_satisfied => false
})
object_3 = DbcObject.new('object_3', :Object, {
  :@is_satisfied => false
})
controller = DbcObject.new('controller', :Controller, {
  :@is_prepared_to_satisfy_all_objects_at_once => false,
```

```
  :@is_prepared_to_satisfy_object_3 => false,
  :@object_1 => object_1,
  :@object_2 => object_2,
  :@object_3 => object_3,
})

# Controller methods
prepare_to_satisfy_all_objects_at_once =
  DbcMethod.new(:prepare_to_satisfy_all_objects_at_once)
prepare_to_satisfy_all_objects_at_once.precondition = Proc.new
   do
  !@object_1.is_satisfied && !@object_2.is_satisfied && !
     @object_3.is_satisfied
end
prepare_to_satisfy_all_objects_at_once.postcondition = Proc.new
    do
  @is_prepared_to_satisfy_all_objects_at_once = true
end
satisfy_all_objects_at_once = DbcMethod.new(:
   satisfy_all_objects_at_once)
satisfy_all_objects_at_once.precondition = Proc.new do
  @is_prepared_to_satisfy_all_objects_at_once
end
satisfy_all_objects_at_once.postcondition = Proc.new do
  @object_1.is_satisfied = @object_2.is_satisfied = @object_3.
     is_satisfied =
    true
end
satisfy_objects_1_and_2 = DbcMethod.new(:
```

```
   satisfy_objects_1_and_2)
satisfy_objects_1_and_2.precondition = Proc.new do
  !@object_1.is_satisfied && !@object_2.is_satisfied
end
satisfy_objects_1_and_2.postcondition = Proc.new do
  @object_1.is_satisfied = @object_2.is_satisfied = true
end
prepare_to_satisfy_object_3 = DbcMethod.new(:
   prepare_to_satisfy_object_3)
prepare_to_satisfy_object_3.precondition = Proc.new do
  @object_1.is_satisfied && @object_2.is_satisfied &&
    !@object_3.is_satisfied && !
       @is_prepared_to_satisfy_object_3
end
prepare_to_satisfy_object_3.postcondition = Proc.new do
  @is_prepared_to_satisfy_object_3 = true
end
satisfy_object_3 = DbcMethod.new(:satisfy_object_3)
satisfy_object_3.precondition = Proc.new {
   @is_prepared_to_satisfy_object_3 }
satisfy_object_3.postcondition = Proc.new do
  @object_3.is_satisfied = true
end
controller.add_dbc_methods(
   prepare_to_satisfy_all_objects_at_once,
   satisfy_all_objects_at_once, satisfy_objects_1_and_2,
   prepare_to_satisfy_object_3, satisfy_object_3)

satisfy_all_objects_use_case = DbcUseCase.new('Satisfy␣All␣
```

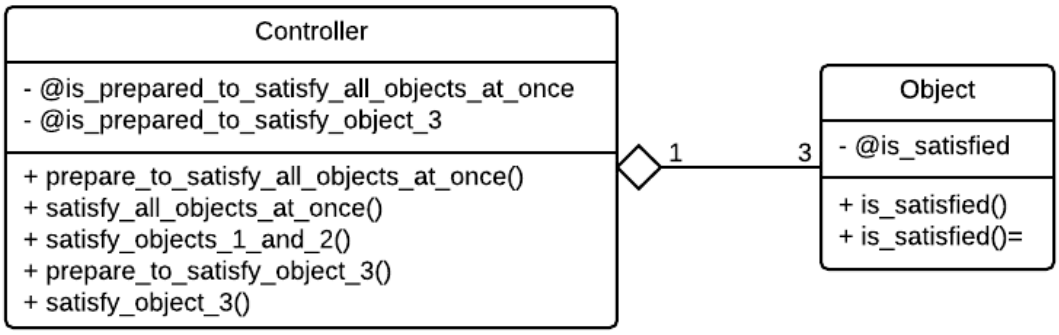

Figure 5.29: The class diagram used for Experiment 4.

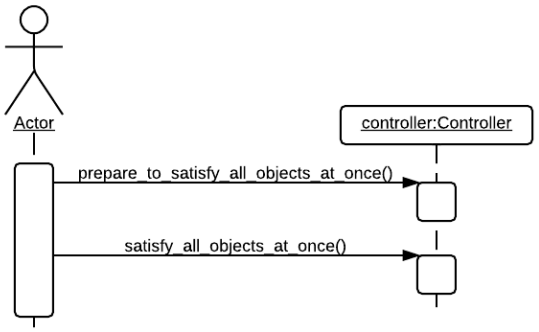

Figure 5.30: The optimal sequence diagram for Experiment 4. This sequence diagram corresponds to the solution returned by depth-first and breadth-first search.

```
   Objects')
satisfy_all_objects_use_case.dbc_instances << controller <<
   object_1 << object_2 << object_3
satisfy_all_objects_use_case.postconditions = {
  'object_1' => Proc.new { @is_satisfied },
  'object_2' => Proc.new { @is_satisfied },
  'object_3' => Proc.new { @is_satisfied },
}
```

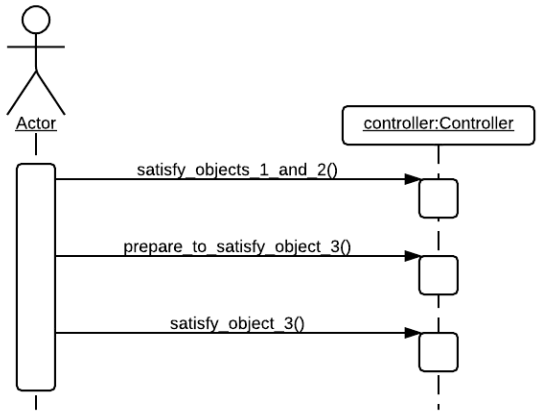

Figure 5.31: The non-optimal sequence diagram for Experiment 4. This sequence diagram corresponds to the solution returned by best-first search.

5.4.2 Outputs and Discussion

Figure 5.30 shows the sequence diagram corresponding to the optimal solution, which is returned by depth-first and breadth-first search, while Figure 5.31 shows the sequence diagram corresponding to the non-optimal solution, which is the one best-first search returns.

Figure 5.32 shows the search space that Communiqué explores in this experiment. Each node represents a state in the search space, where the dark nodes represent goal states. The numbers in front of each state are the values that constitute the objective function $f(n) = g(n) + h(n)$ for that state, where $g(n)$ is the cost (in terms of the number of method calls) of reaching the state, and the heuristic function $h(n)$ is the estimated cost of reaching the nearest goal from the state.

Because $h(n)$ is not admissible (i.e. it may overestimate the true cost to the goal), using best-first as the control strategy of the FORWARD-SEARCH algorithm shown on page 39 will make the planner go down the longer path and return the non-optimal solution. In the first iteration, Communiqué generates the states s_1 and s_2 and enqueues them with the f-values 4 and 2 respectively. In the second

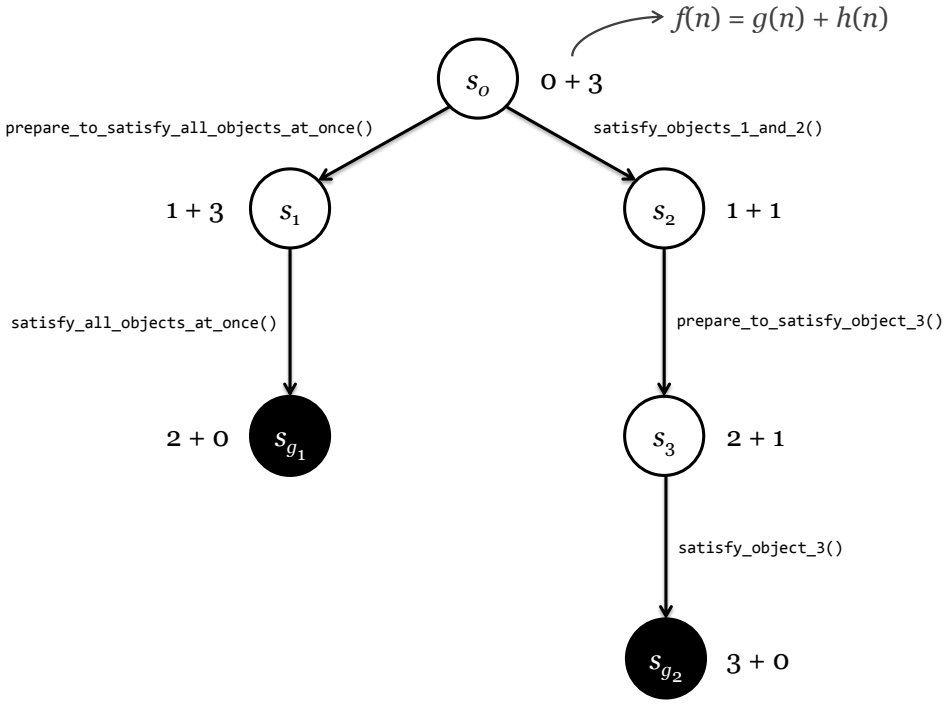

Figure 5.32: The search space for Experiment 4.

iteration, it dequeues s_2 (because it has the lowest f-value), generate its child s_3, and enqueues it with an f-value of 3. In the next iteration, it dequeues s_3 (again, because it has the lowest f-value), generates its child s_{g_2}, and enqueues it with an f-value of 3. In the final iteration, it dequeues s_{g_2} and finds that it is a goal state, so it returns the (non-optimal) solution corresponding to the path s_0, s_2, s_3, s_{g_2}.

Using depth-first as the control strategy leads Communiqué down the other, shorter path (s_0, s_1, s_{g_1}) and results in finding the optimal solution. Note that in this particular experiment depth-first search found the optimal solution because the methods happened to be added to the initial object diagram in an order that lead the search down the shortest path first. Because depth-first is sensitive to the order in which the methods are added, reversing the order in which the methods that lead to the children of s_0 are added will cause depth-first search to go down

the other, longer path and, as a result, return the non-optimal solution.

Using breadth-first as the control strategy also results in finding the optimal solution as FORWARD-SEARCH explores s_0, s_1, s_2, and finally s_{g_1}.

5.5 Experiment 5: Object Instantiation

The purpose of this experiment is to demonstrate the support of Communiqué for object instantiation. The support is currently limited to instantiation dependencies. In UML, dependency is a relationship that basically indicates that a class depends on another: the dependent class (the client) uses the independent class (the supplier) in some way, such as a parameter or local variable of one of the dependent class's methods. A dependency is depicted in a class diagram as a dashed arrow from the client to the supplier. An instantiate dependency (indicated graphically by labeling the arrow with `<<instantiate>>`) means that an operation of the client may create instances of the supplier.

Since instantiate dependencies do not specify which operation of the client creates instances of the supplier, software designers usually consult the sequence diagram involving the client and the supplier to get that information. Communiqué will not have such a sequence diagram to begin with simply because it is actually trying to generate it. Therefore, Communiqué requires the user to specify the objects that a method can create, if any, in the initial object diagram. That is the purpose of the `dependencies` attribute of the `DbcMethod` class in Communiqué.

5.5.1 Inputs

For demonstration purposes, this experiment uses a simple contrived (meta-) example. Listing 5.7 shows the Ruby code that is used to set up the inputs for this

experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. By doing so, it effectively creates an instance of the class digram depicted in Figure 5.33. This instance is used as the initial object diagram for the use case. Note that in this artificial example, the `generate` method of the `SequenceDiagramGenerator` class creates instances of the `Planner` class. The code also sets up the use case's postconditions.

Listing 5.7: The Inputs for the MeetingsMate Experiment

```
sd_postprocessor = DbcObject.new('sd_postprocessor', :
   SequenceDiagramPostprocessor, {
  :@is_sequence_diagram_postprocessed => false
})
sd_generator = DbcObject.new('sd_generator', :
   SequenceDiagramGenerator, {
  :@is_use_case_model_read => false,
  :@is_class_model_read => false,
  :@is_sequence_diagram_generated => false,
  :@sd_postprocessor => sd_postprocessor
})
planner = DbcObject.new('planner', :Planner, {
  :@is_problem_set_up => false,
  :@is_plan_generated => false
})
# planner is a dependency object of the `generate()` method of
# SequenceDiagramGenerator.
planner.dead = true

# SequenceDiagramPostprocessor methods.
postprocess = DbcMethod.new(:postprocess)
```

```
postprocess.precondition = Proc.new do
  state.get_instance_named('planner').is_plan_generated
end
postprocess.postcondition = Proc.new do
  @is_sequence_diagram_postprocessed = true
end
sd_postprocessor.add_dbc_methods(postprocess)

# SequenceDiagramGenerator methods.
read_use_case_model = DbcMethod.new(:read_use_case_model)
read_use_case_model.precondition = Proc.new { true }
read_use_case_model.postcondition = Proc.new do
  @is_use_case_model_read = true
end
read_class_model = DbcMethod.new(:read_class_model)
read_class_model.precondition = Proc.new { true }
read_class_model.postcondition = Proc.new do
  @is_class_model_read = true
end
generate = DbcMethod.new(:generate)
generate.precondition = Proc.new do
  @is_use_case_model_read && @is_class_model_read
end
generate.postcondition = Proc.new do
  @is_sequence_diagram_generated = true
end
generate.dependencies.push(planner.dbc_name)
sd_generator.add_dbc_methods(read_use_case_model,
   read_class_model, generate)
```

```
# Planner methods.
set_up_problem = DbcMethod.new(:set_up_problem)
set_up_problem.precondition = Proc.new do
  state.get_instance_named('sd_generator').
     is_use_case_model_read &&
    state.get_instance_named('sd_generator').
       is_class_model_read
end
set_up_problem.postcondition = Proc.new do
  @is_problem_set_up = true
end
solve = DbcMethod.new(:solve)
solve.precondition = Proc.new do
  @is_problem_set_up
end
solve.postcondition = Proc.new do
  @is_plan_generated = true
end
planner.add_dbc_methods(set_up_problem, solve)

generate_sequence_diagram_use_case = DbcUseCase.new('Generate␣
   Sequence␣Diagram')
generate_sequence_diagram_use_case.dbc_instances <<
   sd_postprocessor << sd_generator << planner
generate_sequence_diagram_use_case.postconditions = {
  'sd_generator' => Proc.new { @is_sequence_diagram_generated
     },
  'sd_postprocessor' => Proc.new {
```

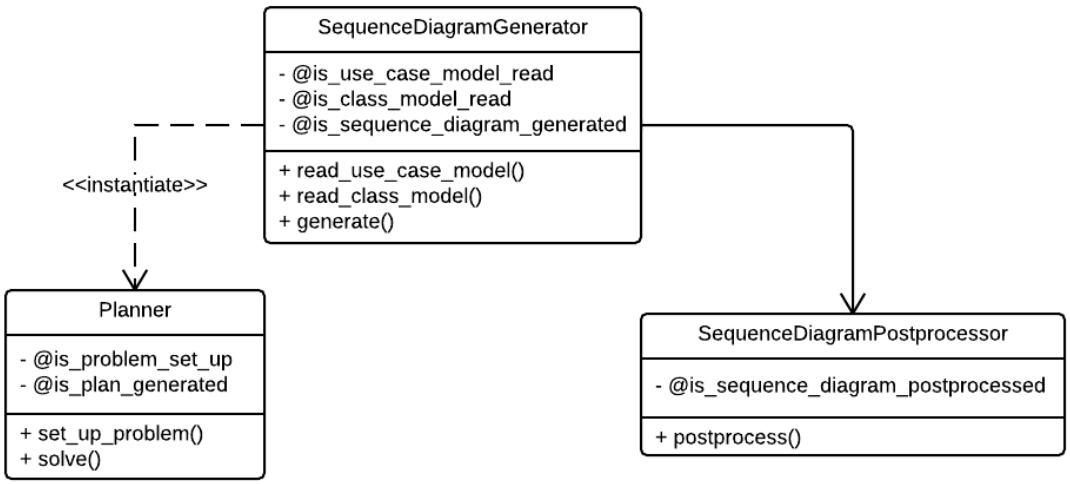

Figure 5.33: The class diagram used for Experiment 5.

```
    @is_sequence_diagram_postprocessed }
}
```

5.5.2 Outputs

Figure 5.34 shows the sequence diagram that corresponds to the solution returned by Communiqué. Communiqué sees that `generate()` depends on `Planner` but there is no `Planner` instance in the current state when `generate()` is called. So, it decides that a `<<create>>` message must be sent from the `SequenceDiagramGenerator` instance (the dependent object) to `Planner` (the independent object).

5.6 Experiment 6: Failure Handling

The purpose of this experiment is to show that Communiqué is capable of pointing out possible sources of inconsistencies that caused it to fail in finding a sequence diagram that satisfies the given use case postconditions. There are a number of reasons that can cause the planner to fail: there could be a mistake in a method's preconditions or postconditions, there could be a mistake in the use

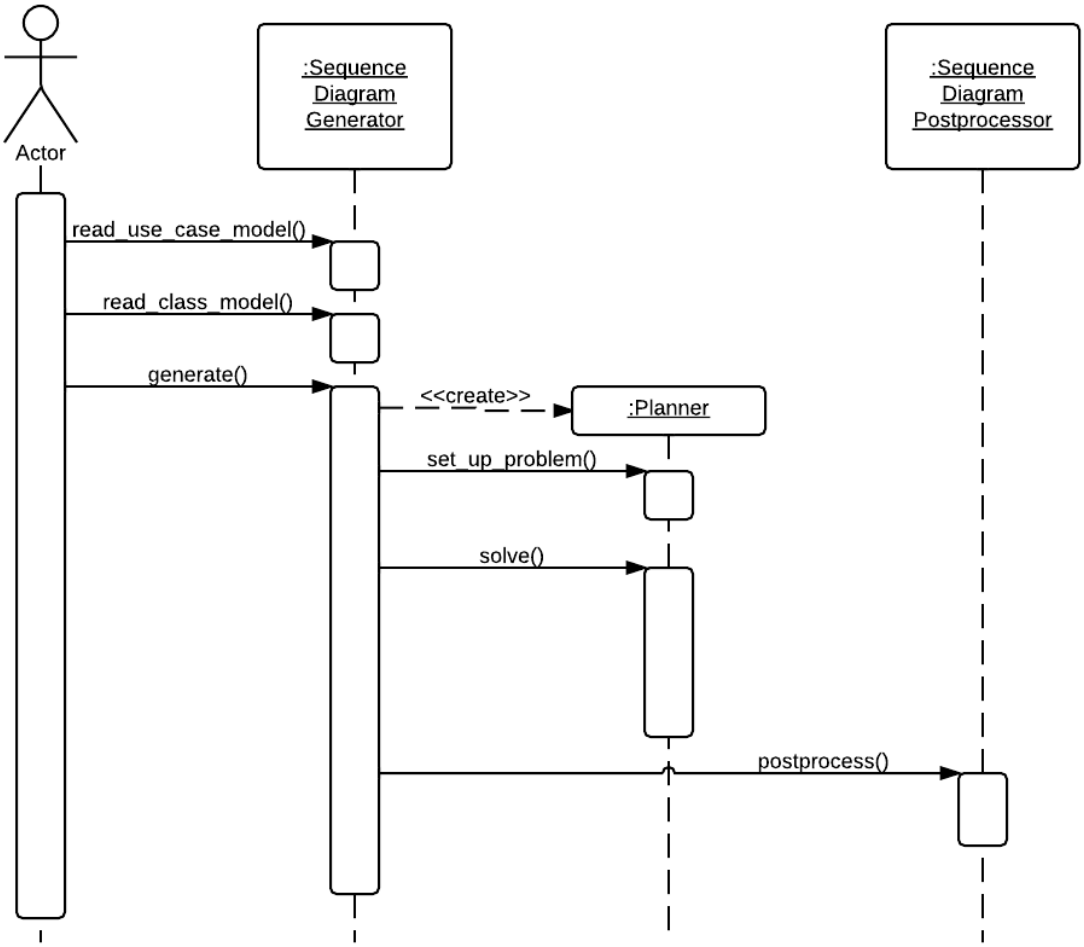

Figure 5.34: The sequence diagram corresponding to the solution returned by Communiqué for Experiment 5.

case's preconditions or postconditions, or a method that is necessary for satisfying the use case's postconditions could be missing altogether.

By keeping track of the best state it has seen so far during its search for a solution, the planner can point out the possible inconsistency sources in case it fails to find a solution sequence diagram. In such a case, the planner uses that state to report the names of the objects it was not able to satisfy. Using this information, the software developer can go back and check the classes of those objects, their methods, and their contracts for mistakes or omissions.

5.6.1 Inputs

For demonstration purposes, this experiment uses a simple contrived (meta-) example. Listing 5.8 shows the Ruby code that is used to set up the inputs for this experiment. The code defines the instances, the methods, and the contracts that are used in this experiment. In other words, it effectively creates an instance of the class digram depicted in Figure 5.35. This instance is used as the initial object diagram for the use case. Notice that an inconsistency is intentionally introduced by not adding a `save_diagram` method to the `SequenceDiagramGenerator` class. This method is necessary for achieving one of the use case's postconditions.

Listing 5.8: The Inputs for the Failure Handling Experiment

```
sequence_diagram_generator = DbcObject.new('
   sequence_diagram_generator', :SequenceDiagramGenerator, {:
   @is_diagram_saved => false})
planner = DbcObject.new('planner', :Planner, {:@is_done_solving
    => false})
planner.dead = true


# SequenceDiagramGenerator methods
```

```
generate = DbcMethod.new(:generate)
generate.precondition = Proc.new { true }
generate.postcondition = Proc.new {}
generate.dependencies.push(planner.dbc_name)
sequence_diagram_generator.add_dbc_methods(generate)
# Intentionally introduce an inconsistency by not adding a
# `SequenceDiagramGenerator#save_diagram()` method.

# Planner methods
solve = DbcMethod.new(:solve)
solve.precondition = Proc.new { true }
solve.postcondition = Proc.new { @is_done_solving = true }
planner.add_dbc_methods(solve)

solve_and_save_use_case = DbcUseCase.new('Solve␣and␣Save')
solve_and_save_use_case.dbc_instances <<
   sequence_diagram_generator << planner
solve_and_save_use_case.postconditions = {
  'sequence_diagram_generator' => Proc.new { @is_diagram_saved
     },
  'planner' => Proc.new { @is_done_solving }
}
```

5.6.2 Outputs

Listing 5.9 shows the command-line output of Communiqué for this experiment. Upon failure, Communiqué uses the best state it has seen so far during its search to report the names of the objects that remained unsatisfied. That is, any object that is in a state that does not satisfy the conditions on this object expressed in

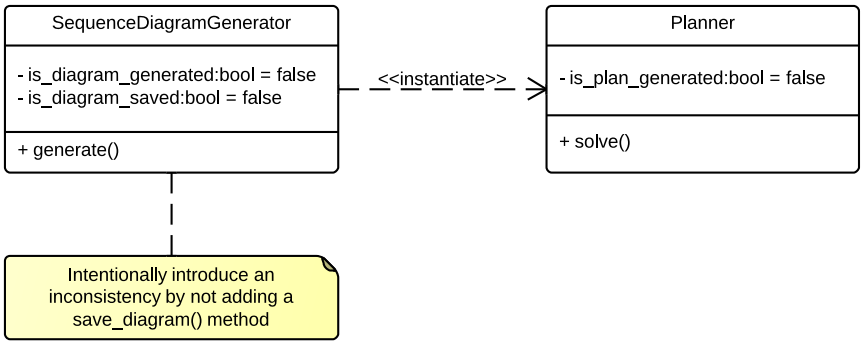

Figure 5.35: The class diagram used for Experiment 6.

```
Failed to satisfy the goals of the following objects:
sequence_diagram_generator
```

Listing 5.9: The Command-Line Output of Communiqué for Experiment 6

the use case's postconditions. In this particular experiment, the unsatisfied object is `sequence_diagram_generator`.

5.7 Summary

Table 5.2 summarizes the experiments presented in this Chapter. Out of the nine manually-designed sequence diagrams that were obtained from other sources and used as target sequence diagrams, Communiqué was able to match seven of them. Note that Experiments 4, 5, and 6 did not use sequence diagrams designed by other researcher and instead used contrived examples because their purpose was to demonstrate features of Communiqué other than the ability to match manually-designed sequence diagrams.

Experiment	# Sequence Diagrams	Match?	Comments
# 1	5	Yes	Depth-first was susceptible to irrelevant methods.
# 2 (`2Bwatch`)	1	No	There was a mistake in the original models.
# 2 (`ARENA`)	1	Yes	
# 2 (Weather Station)	1	No	Messages to the actor are not supported.
# 3	1	Yes	Depth-first was susceptible to irrelevant methods.
# 4	N/A	N/A	Experiment used a contrived example.
# 5	N/A	N/A	Experiment used a contrived example.
# 6	N/A	N/A	Experiment used a contrived example.

Table 5.2: Summary of the Experiments

CHAPTER 6

CONCLUSION

6.1 Main Contributions

Since—as far we were able to tell—there were no published works on applying automated planning to the problem of UML sequence diagram generation when Design by Contract is used to develop the use cases and class diagrams, the research efforts behind this Thesis had two main objectives:

1. To show that the core activity of sequence diagram generation (that is, determining the sequence of message passes) can indeed be treated as a planning problem and solved as such.

2. To identify and point out the issues that need to be addressed and the problems that need to be solved to advance the application of automated planning to the domain of sequence diagrams and take it to the next level.

In essence, we saw a door that emerged under the right conditions and that, to the best of our knowledge, no other researcher tried to open. To achieve the first objective, we relied upon the correspondence between automated planning and Design by Contract and adapted the conceptual model of state-transition systems used in automated planning to the problem of planning messages in sequence

diagrams. Along the way, we discovered some of the differences between action planning and message-pass planning that makes adapting existing action planners to sequence diagrams difficult. Based on those differences, we implemented Communiqué, a software library for planning messages in sequence diagrams. Table 6.1 compares Communiqué as a message-pass planner to existing action planners. Using Communiqué, we empirically showed that even with a simple conceptual model of restricted state-transition systems, determining the sequence of message passes in sequence diagrams can be treated as a planning problem and can be solved using forward state-search space.

	Action Planners	Communiqué
State Representation	Typically, predefined state variables	A set of objects and their links
Specification Language	STRIPS, ADL, or PDDL	OCL-Like Ruby expressions
Constrains	Preconditions only	Preconditions plus semantic class relationships
Creation of New State Components	Typically, not supported	Object instantiation using instantiate dependencies

Table 6.1: Action Planners vs. Communiqué as a Message-Pass Planner

As the door that we were trying to open was new, we knew that we will raise more questions than we can answer. Therefore, the second objective of our research efforts was to draw the attention of other researchers in the field to this new door; to shed a light on what else needs to be done to advance the application of automated planning to sequence diagrams.

To this effect, Section 6.2 will reiterate the limitations of our work, and Section 6.3 will discuss possible future work.

6.2 Limitations

While what follows below are limitations, they are part also a main part of the contribution of our research because one of the major objectives of this Thesis was to identify and point out the issues that need to be addressed further to advance the application of automated planning to the domain of sequence diagrams. The limitations can be divided into two types. The more challenging limitations are ones coming from the domain itself and my require a change of the planning approach. These limitations are discussed in Subsections 6.2.1 and 6.2.2 below. The rest are limitations of the current implementation of Communiqué that can be addressed using the same approach it currently uses. These limitations are discussed in the rest of the Subsections.

6.2.1 Instantaneous State Transitions

The conceptual model of restricted state-transition systems (which was discussed in Section 4.2) that takes the differences between action planning and message-pass planning (which were discussed in Section 4.3) into account is adequate for determining the sequence of message passes in a sequence diagram. On its own, however, that model is not adequate for determining the method activations of

the sequence diagram. The reason is that "time is abstracted away in the state-transition model" [13, p. 13]. That is to say, the state transitions are instantaneous.

What the above means in practice is that, as Figure 6.1 demonstrates, Communiqué effects the postconditions of a method earlier than it ideally should. So in Figure 6.1, Communiqué effects the postconditions of m_1 when the message is sent at time t_1 instead of at the end of m_1's activation at time t_2. Sometimes, this may cause Communiqué to return a solution that is, from the point of view of sequence diagrams, redundant. That is, the sequence of message passes may contain a message pass that is needlessly repeated in the sequence. In some cases, the problem could be fundamentally caused by methods that are not modular: the methods may be trying to achieve too much by asserting too many postconditions. In such cases the solution would be to break down the method into smaller, more focused ones. In other cases, the methods themselves would be modular, but the planner returns redundant solutions simply because it does not handle time explicitly.

6.2.2 Sender Selection Inaccuracies

As was discussed in Section 4.3, a challenging task in automatic sequence diagram generation is determining the sender of the message because links between the objects in a state are dynamic: they may change from one state to the other. So, a valid sender of a message in some state may become an invalid sender in the next. Because of this, information about the senders of message cannot be statically added to the problem description in advance. Instead, the automated planner must dynamically infer such information using the links between the objects in the current state.

As the rules discussed in Subsection 4.4.3 that Communiqué uses are rules

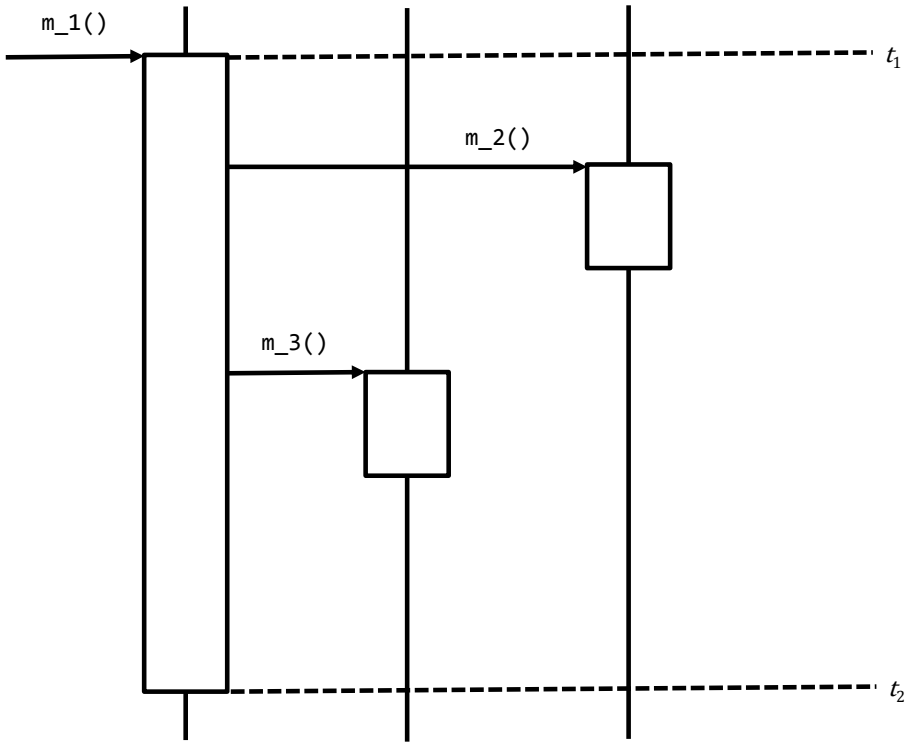

Figure 6.1: The effects of instantaneous state transitions on message-pass planning.

of thumb, the planner may select a sender other than the one that the software designer originally had in mind. That is, the output of the planner may result in a sequence diagram that is different—in terms of the senders of the messages—from the one that the software designer may have come up with manually. This was demonstrated with an actual example in Section 5.2.

6.2.3 Non-Optimality of Communiqué's Best-First Search

Since the heuristic function $h(n)$ that Communiqué uses is not admissible (that is, it may overestimate the true cost of reaching a goal), Communiqué's best-first search is not optimal.

6.2.4 Limited Message Types

The current implementation of Communiqué assumes that a message is a (synchronous) method call (or operation invocation). UML Sequence diagrams can have other types of messages, such as asynchronous calls, asynchronous signals, and create, delete, or reply messages. Although Communiqué supports, in effect, object instantiation and can generate solutions that contain `<<create>>` messages, it does not support `<<destroy>>` messages (this issue is related to the limitation of instantaneous state transitions discussed in Subsection 6.2.1).

6.2.5 Limited Actor Support

Currently, Communiqué assumes that there is only one actor in the sequence diagram. In other words, Communiqué does not inherently support multiple actors. Also, while Communiqué supports, in effect, sending messages from the actor, it does not support sending messages to it. One could create classes that represent actors and add their instances to the initial object diagram, but the assumptions

underlying the current implementation of Communiqué, especially those underlying the sender-selection rules discussed in Subsection 4.4.3, may result in solutions representing sequence diagrams that do not match the manually-designed ones.

6.2.6 No Support for Combined Fragments

Communiqué does not currently support combined fragments, which are basically logical groupings of sets of messages in sequence diagrams. These fragments include alternatives (`alt`), loops (`loop`), and options (`opt`).

6.2.7 Possible Bias in Experiments

As was pointed out in Chapter 5, the UML models used in the experiments were not originally designed following the Design by Contract approach. Because of that, we had to add the contracts to the models ourselves as if we were designing the models using Design by Contract. Of course, if the original software designers were to use Design by Contract, they may have specified contracts different from ours. So, although we tried our best to not to tailor the contracts to suite Communiqué, we acknowledge that we may have introduced some kind of bias in the contracts and, consequently, the experiments.

6.2.8 Using Ruby for Inputs and Raw Outputs

By using Ruby code blocks, we were able to use the Ruby language itself, which is Turing-complete, as the language for the contracts. This enables the user of Communiqué to specify the contracts in a syntax similar to that of the Object Constraint Language (OCL) [12], or any other syntax of his or her choosing (as long as it is valid Ruby code). The reliance on Ruby, however, may prevent software designer and developer who do not know Ruby from using Communiqué.

6.3 Future Work

6.3.1 Handle Time Explicitly

To address the limitation of instantaneous state transitions, the planning model must explicitly represent time. That is the case in temporal planning, which extends the planning model, as Ghallab, Nau, and Traverso [13] note, in either of two ways:

1. By including time explicitly in the representation of state-transition systems, such as the case in timed automata.

2. By taking a time-oriented view in which the dynamics of the system are represented by a set of functions of time that describe the evolution of state variables.

6.3.2 Design an Admissible Heuristic

Designing an admissible heuristic and using it with best-first search will make Communiqué an optimal planner.

6.3.3 Try Other Planing Algorithms and Approaches

Forward state-space search is only one of the many different automated planning algorithms. Another algorithm that uses the same planning model of state-transition systems is backward state-space search. Even more different, instead of searching a space of states, plan-space planning searches a space of plans. It would be interesting to see how the other planning approaches and algorithms can be applied to the problem of planning messages in sequence in particular, and the problem of sequence diagram generation in general.

6.3.4 Use XML Metadata Interchange for Inputs and Outputs

We ultimately plan to implement the proposed technique as a plug-in to a Computer-Aided Software Engineering (CASE) tool. To ease the integration with existing CASE tools, we plan to use the XML Metadata Interchange (XMI) [35], which is an international standard for sharing metadata and models—especially UML models—using XML (Extensible Markup Language), as the format of the initial inputs and final outputs of the message-pass planner.

REFERENCES

[1] I. Sommerville, *Software Engineering*, 9th ed. Pearson Education Limited, 2011.

[2] Object Management Group, "OMG Unified Modeling Language (OMG UML), Superstructure," 2010. [Online]. Available: http://www.omg.org/spec/UML/2.3/

[3] J. Iivari, "Object-orientation as structural, functional and behavioural modelling: a comparison of six methods for object-oriented analysis," *Information and Software Technology*, vol. 37, no. 3, pp. 155–163, 1995. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/0950584995989267

[4] T. Yue, L. C. Briand, and Y. Labiche, "Automatically Deriving UML Sequence Diagrams from Use Cases," Simula Research Laboratory, Tech. Rep., 2010.

[5] "What's New in the .NET Framework 4," http://msdn.microsoft.com/en-us/library/ms171868.aspx, accessed 7 May 2011.

[6] S. McConnell, *Code Complete, Second Edition*. Microsoft Press, 2004.

[7] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering using UML, Patterns and Java*, 3rd ed. Prentice Hall, 2010.

[8] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, 2004.

[9] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[10] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall PRT, 1997.

[11] ——, *Eiffel: The Language*, 1st ed. Prentice Hall, 1991.

[12] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison Wesley, 2003.

[13] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*, 1st ed. Morgan Kaufmann, 2004.

[14] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Education, 2009.

[15] R. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, no. 3/4, pp. 189–208, 1971.

[16] E. P. D. Pednault, "ADL: exploring the middle ground between STRIPS and the situation calculus," in *Proceedings of the first international conference on Principles of knowledge representation and reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 324–332. [Online]. Available: http://dl.acm.org/citation.cfm?id=112922.112954

[17] ——, "ADL and the state-transition model of action," *Journal of Logic and Computation*, vol. 4, no. 5, pp. 467–512, 1994.

[18] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL — the planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep., 1998.

[19] T. Yue, L. C. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," *Requirements Engineering*, Aug. 2010. [Online]. Available: http://www.springerlink.com/index/10.1007/s00766-010-0111-y

[20] T. Yue, L. Briand, and Y. Labiche, "A Use Case Modeling Approach to Facilitate the Transition towards Analysis Models: Concepts and Empirical Evaluation," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5795, pp. 484–498. [Online]. Available: http://www.springerlink.com/index/10.1007/978-3-642-04425-0

[21] L. Li, "Translating use cases to sequence diagrams," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering.* IEEE Comput. Soc, 2000, pp. 293–296. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=873681

[22] X. Li, Z. Liu, and J. He, "Consistency Checking of UML Requirements," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05).* IEEE, 2005, pp. 411–420. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1467923

[23] Q. Long, Z. Liu, X. Li, and H. Jifeng, "Consistent Code Generation from UML Models," in *2005 Australian Software Engineering*

Conference, no. 60173003. IEEE, 2005, pp. 23–30. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1401997

[24] C. Ghezzi, A. Mocci, and G. Salvaneschi, "Automatic Cross Validation of Multiple Specifications: A Case Study," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, D. S. Rosenblum and G. Taentzer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6013, pp. 233–247. [Online]. Available: http://www.springerlink.com/index/10.1007/978-3-642-12029-9

[25] J. Chanda, A. Kanjilal, S. Sengupta, and S. Bhattacharya, "Traceability of Requirements and Consistency Verification of UML UseCase, Activity and Class diagram: A Formal approach," in *Proceeding of International Conference on Methods and Models in Computer Science*, 2009, pp. 1–4.

[26] G. Kösters, H.-W. Six, and M. Winter, "Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications," *Requirements Engineering*, vol. 6, no. 1, pp. 3–17, Feb. 2001. [Online]. Available: http://www.springerlink.com/index/10.1007/PL00010354

[27] T. C. de Sousa, J. R. Almeida, S. Viana, and J. Pavón, "Automatic analysis of requirements consistency with the B method," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 2, p. 1, Mar. 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1734103.1734114

[28] T. S. Vaquero, R. Silva, and J. C. Beck, "A Brief Review of Tools and Methods for Knowledge Engineering for Planning & Scheduling," in *Proceedings of the ICAPS2011 Workshop on Knowledge Engineering for Planning and Scheduling*, R. Barták, S. Fratini, L. McCluskey, and T. S. Vaquero, Eds., Freiburg, Germany, 2011, pp. 7–14.

[29] T. S. Vaquero, J. R. Silva, F. Tonidandel, and J. C. Beck, "itSIMPLE: Towards an Integrated Design System for Real Planning Applications," *The Knowledge Engineering Review*, pp. 1–15, 2011. [Online]. Available: http://tidel.mie.utoronto.ca/pubs/KERtvaquero.pdf

[30] Y. Sulaiman and M. Ahmed, "Automating UML Sequence Diagram Generation by Treating it as a Planning Problem," in *The 18th International Conference on Distributed Multimedia Systems (DMS 2012)*. Knowledge Systems Institute, 2012, pp. 124–129.

[31] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, 3rd ed. The Pragmatic Programmers, 2012.

[32] P. Perrotta, *Metaprogramming Ruby: Program Like the Ruby Pros*, 1st ed. The Pragmatic Programmers, 2010.

[33] A. Aman, K. Al-Alshaikh, M. Alhomaidan, M. Al-Butairi, S. Al-Garni, S. Al-Bader, and S. Al-Refai, "Properties Management System," 2011, software Engineering Senior Project SRS.

[34] I. Al Akel, A. Al Kalaji, L. Labani, F. Al Hazemi, A. Ba Haziq, and H. Al Zahrani, "MeetingsMate," 2011, software Engineering Senior Project SRS.

[35] Object Management Group, "XML Metadata Interchange Specification Version 2.0.1," 2005. [Online]. Available: http://www.omg.org/spec/XMI/

Vitae

- Name: Yaser Ali Diab Sulaiman
- Nationality: Syrian
- Date of Birth: 6 October 1984
- Education:
  - Master of Science in Computer Science, King Fahd University of Petroleum and Minerals, Saudi Arabia, December 2012
  - Bachelor of Science in Computer Science, King Fahd University of Petroleum and Minerals, Saudi Arabia, January 2008
- Work Experience:
  - System Analyst, Al Mashariq, Dammam, Saudi Arabia
  - Software Developer, Free Dimension IT, Khobar, Saudi Arabia
  - Software Engineer, Futureware, Khobar, Saudi Arabia
- Research Interests:
  - Machine Learning
  - Neural Networks
  - Graph Theory
- Contact Details:
  - Email: `yaser.a.sulaiman@gmail.com`
  - Permanent Address: Apartment 3, Avicenna Dental Center Building, King Abdul Aziz Street, Dammam, Saudi Arabia