Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Characterization of Implied Scenarios as Families of Common Behavior

Caio Batista de Melo

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientadora
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Characterization of Implied Scenarios as Families of Common Behavior

Caio Batista de Melo

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Brasília, 31 de agosto de 2018

Dedicatória

Dedico este trabalho a você que está lendo. Espero que este lhe ajude em seus próprios trabalhos e estudos, assim como muitos outros ajudaram para que este fosse concluído.

Agradecimentos

Aos meus professores, que me propiciaram a oportunidade de chegar até aqui, sempre contribuindo para o meu crescimento pessoal e profissional. Também aos meus amigos, que me acompanharam nesta caminhada onde criamos alegres lembranças. E principalmente à minha família, pelo constante apoio e por me colocar no caminho de grandes conquistas.

Resumo

Sistemas concorrentes enfrentam uma ameaça à sua confiabilidade em comportamentos emergentes, os quais não são incluídos na especificação, mas podem acontecer durante o tempo de execução. Quando sistemas concorrentes são modelados a base de cenários, é possível detectar estes comportamentos emergentes como cenários implícitos que, analogamente, são cenários inesperados que podem acontecer devido à natureza concorrente do sistema. Até agora, o processo de lidar com cenários implícitos pode exigir tempo e esforço significativos do usuário, pois eles são detectados e tratados um a um. Nesta dissertação, uma nova metodologia é proposta para lidar com vários cenários implícitos de cada vez, encontrando comportamentos comuns entre eles. Além disso, propomos uma nova maneira de agrupar estes comportamentos em famílias utilizando uma técnica de agrupamento usando o algoritmo de Smith-Waterman como uma medida de similaridade. Desta forma, permitimos a remoção de vários cenários implícitos com uma única correção, diminuindo o tempo e o esforço necessários para alcançar maior confiabilidade do sistema. Um total de 1798 cenários implícitos foram coletados em sete estudos de caso, dos quais 14 famílias de comportamentos comuns foram definidas. Consequentemente, apenas 14 restrições foram necessárias para resolver todos os cenários implícitos coletados coletados, aplicando nossa abordagem. Estes resultados suportam a validade e eficácia da nossa metodologia.

Palavras-chave: dependabilidade, cenários implícitos, sistemas concorrentes, LTSA, Smith-Waterman, clusterização hierárquica

Abstract

Concurrent systems face a threat to their reliability in emergent behaviors, which are not included in the specification but can happen during runtime. When concurrent systems are modeled in a scenario-based manner, it is possible to detect emergent behaviors as implied scenarios (ISs) which, analogously, are unexpected scenarios that can happen due to the concurrent nature of the system. Until now, the process of dealing with ISs can demand significant time and effort from the user, as they are detected and dealt with in a one by one basis. In this paper, a new methodology is proposed to deal with various ISs at a time, by finding Common Behaviors (CBs) among them. Additionally, we propose a novel way to group CBs into families utilizing a clustering technique using the Smith-Waterman algorithm as a similarity measure. Thus allowing the removal of multiple ISs with a single fix, decreasing the time and effort required to achieve higher system reliability. A total of 1798 ISs were collected across seven case studies, from which 14 families of CBs were defined. Consequently, only 14 constraints were needed to resolve all collected ISs, applying our approach. These results support the validity and effectiveness of our methodology.

Keywords: dependability, implied scenarios, concurrent systems, LTSA, Smith-Waterman, hierarchical clustering

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

A useful way to model distributed systems' specifications is to use scenarios. A scenario depicts how different components interact to achieve a common goal. Message Sequence Charts [3] and UML sequence diagrams [4] are two commonly used methods to design and display these scenarios. Both of these techniques use the idea of components that send each other messages, that is, two different components that need to work together to achieve a goal can interact synchronously or asynchronously with each other through message passing.

Although widely used, a scenario specification can describe only partial behaviors of a system, while its implementation has full behaviors [5]. Such limitations can lead to a common fault in scenario specifications, which are implied scenarios. Implied scenarios are unexpected behaviors that can emerge at runtime due to components' interactions that are implied in the specification [5]. That is, different components believe they are behaving correctly on its own, but the composition of all their actions together is not included in the original specification. More formally, "implied scenarios indicate gaps in a scenario-based specification that are the result of specifying the behavior of a system from a global perspective yet expecting the behavior to be provided in a local component-wise fashion by independent entities with a local view of the system" [6].

Even though implied scenarios can lead to unexpected behaviors, they are not always unacceptable behaviors [7]. An implied scenario can be a positive scenario that was overlooked in the original specification or be indeed an unacceptable behavior. For the former, it can be simply included in the specification, while the latter has to be constrained. Therefore, implied scenarios should be detected and validated with stakeholders [8] to define how to deal with them. If left untreated these scenarios can cause damage if they lead to unwanted behaviors [9]. Particularly, implied scenarios can affect the reliability [10,11] and security [12] of a system. Thus, it is desirable to deal with these implied scenarios before the system is up and running to prevent unwanted behavior. However, the process of detecting implied scenarios has been proved to be undecidable by Chakraborty et al. [13], meaning that there is no guarantee that the detection of all implied scenarios of a given system will ever stop in polynomial time.

1.2 Problem

Several approaches to detect implied scenarios have been devised (e.g., [5–7,14–16]). Most of them, however, do not go further on the process of dealing with implied scenarios, that is, they stop their methodologies after detecting implied scenarios. In other words, they neither suggest a solution nor try to find the root of the underlying cause. By doing so, it can lead the user to spend a lot of time analyzing a large number of implied scenarios. Some exceptions, such as Uchitel et al. [6], Song et al. [5], and Moshirpour et al. [16], show the cause to the user and suggest solutions for the problem to be fixed. However, these approaches can still output a large number of implied scenarios, which can be cumbersome to the user. Since these approaches do not further investigate the correlation between the implied scenarios, they could misguide the user on how to deal with such implied scenarios at large.

1.3 Research Questions and Contributions

Based on these problems, three research questions are raised, which motivates the contributions of the dissertation.

Research Question 1 Can we provide a heuristic to restrict the problem space of implied scenarios analysis?

To answer this first question, a new methodology is proposed to fill in the literature gap, which is achieved by finding common behaviors among implied scenarios that lead the system to unexpected behavior. The process of detecting the common behaviors consists of two major steps: (i) collect multiple implied scenarios and (ii) detect common behaviors among them. For the first step, we extend the approach by Uchitel et al. [6], where we automated the process of collecting implied scenarios, that is, instead of detecting a single IS, our approach iteratively collects multiple distinct ISs without the need of user interaction. From these collected implied scenarios, their underlying core common behaviors are extracted. By these means, we restrict the problem space of the undecidability of implied scenario detection by treating the implied scenarios as a group, instead of individually. Therefore, we answer Research Question 1, as we successfully restricted the problem space that the user needs to analyze.

Research Question 2 Can we semantically group the common behaviors in a sound manner?

Following, we group the common behaviors that were detected. This is achieved by hierarchically clustering them, and in order to calculate the distance among the common behaviors we use one of the several string matching algorithms that exist in the literature: the Smith-Waterman algorithm [17], widely used in bioinformatics research. This algorithm is used to identify the best local alignment between genetic sequences, that is, it tries to find which parts of the two DNA sequences have the most in common. Thus, it can be successfully used to find the underlying sequences that are shared among common behaviors. By using this algorithm, we were able to find semantically defined groups of common behaviors, which helps the user analysis of the system. Therefore, by these means we are able to semantically group the common behaviors, answering Research Question 2. Furthermore, we compare the clustering results with the ones obtained using the Levenshtein distance [18], in order to support the choice for the Smith-Waterman algorithm.

Research Question 3 Is it possible to state there is an underlying cause among the groups of common behaviors?

Next, a manual analysis of the groups is required. By doing so, it is possible to identify underlying causes among common behaviors by looking at the alignments obtained with the Smith-Waterman algorithm. Therefore, it is possible to define groups of common behaviors that share the same underlying cause. We call such groups *families of common behaviors*. If the underlying cause of a family is undesirable, that is, it deviates from the wanted behavior of the system, it is a fault. Thus, if it is not treated, it may lead to system failures. As a proof of concept, all underlying causes were considered to be faults.

After that, each family of common behaviors was dealt with in a single fix. In other words, we were able to resolve all common behaviors in the family, by constraining their underlying cause and not each one individually. By these means, we are able to successfully identify the underlying cause among a family of common behaviors. As a result, we contribute to answering both Research Question 3 and a research question previously raised by Uchitel et al. [8]: *"should the entire implied scenario be constrained or is the* *unaccepted situation due to a specific subsequence of actions that appear in the implied scenario?"* In fact, our results explicitly show that it is possible to prevent unaccepted situations by constraining specific underlying causes of common behaviors, which we can detect by characterizing families.

Finally, in order to evaluate the proposed methodology, we performed seven case studies with system specifications reported in the literature. Throughout these case studies, we collected a total of 1798 implied scenarios, which demanded nearly 37 hours of execution. From these implied scenarios, our methodology was able to come up with only 14 families of common behaviors, where each family required a single constraint and each system specification required at most three constraints to prevent all collected implied scenarios. Additionally, our methodology shows that the same 14 families could have been found with 424 collected implied scenarios instead of the original 1798. This would reduce the timespan of the detection process from nearly 37 hours to under 24 minutes. Thus, the main contribution of this dissertation is the methodology which facilitates the analysis of multiple implied scenarios.

1.4 Outline

The rest of this dissertation is structured as follows: Chapter 2 introduces and details technical concepts used in the methodology; Chapter 3 discusses the related work in the literature; Chapter 4 explains the proposed methodology in further details and uses an example system to illustrate the approach; Chapter 5 shows and discusses the results obtained through seven case studies; finally, Chapter 6 summarizes the contributions from this dissertation and introduces some ideas for future work.

Chapter 2

Background

In this chapter, definitions and technical concepts that are used throughout this dissertation will be laid out and explained with examples, where applicable.

2.1 Scenarios

"Scenarios describe how system components, the environment and users work concurrently and interact in order to provide system level functionality" [6]. Simply put, a scenario is a description of a system's action. It describes what the user expects from the system when interacting with it. We can model entire systems based solely on scenarios that it needs to execute, this is called a positive scenario-based model by Uchitel et al. [6].

The *Boiler System* [6] will be used as a running example. The *Boiler System* is a system that controls the temperature inside a boiler, according to the measured pressure by its sensor. It has the following components:

Actuator variates the temperature inside the boiler;

Control tells the actuator to act according to the last pressure measured;

Database stores the measured pressures;

Sensor measures the pressure inside the boiler.

This system performs four scenarios, and these are all accomplished by interactions between the components. Below these scenarios are shown and the interactions between the system's components are described:

Initialise : *Control* tells *Sensor* to start monitoring the pressure;

Terminate : *Control* tells *Sensor* to stop monitoring the pressure;

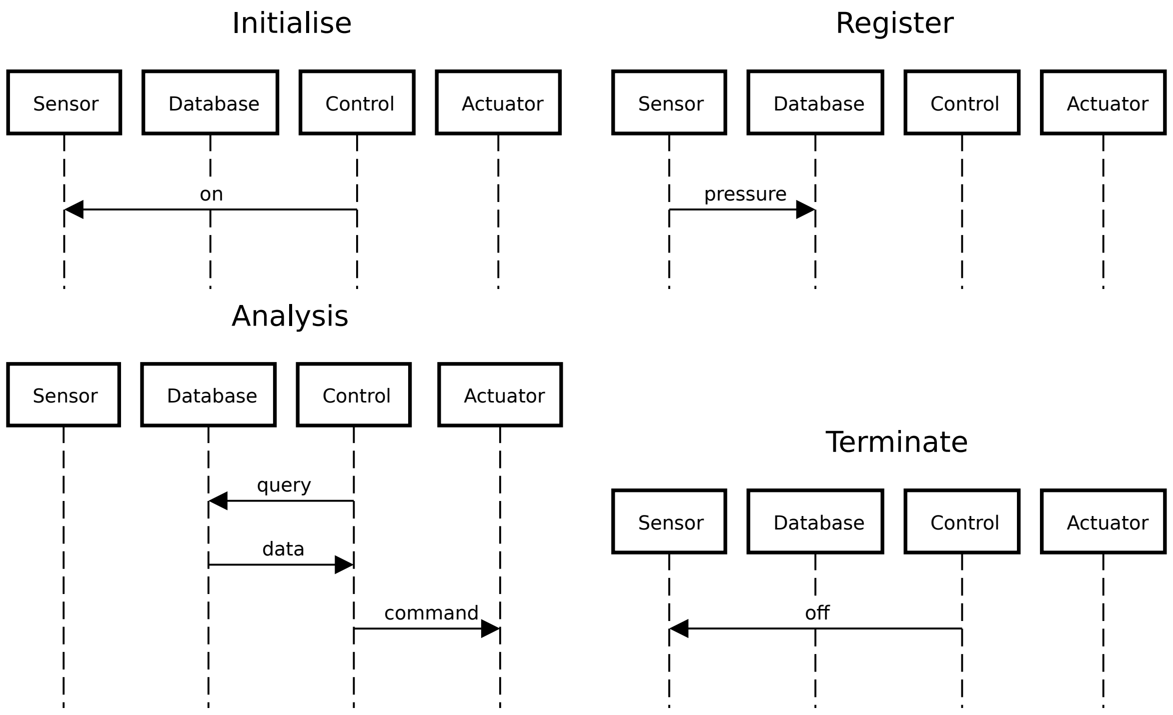


Figure 2.1: bMSCs representing the Boiler's scenarios.

Register : *Sensor* sends *Database* the current pressure so it is stored and can be queried later on;

Analysis : *Control* queries *Database* for the latest pressure and tells *Actuator* to alter the boiler's temperature accordingly.

By implementing these four scenarios, we have a system that was modeled on a scenario-based way.

2.1.1 Message Sequence Chart

A message sequence chart (MSC) is a simple and intuitive graphical representation of a scenario [3]. It explicitly shows the interactions between components, by showing each one of those as a message sent from one component to another. We can use MSCs to show the *Boiler System*'s scenarios described above as in Figure 2.1. As it can be seen, it is a convenient way to exemplify the interactions that happen for a scenario to be achieved.

The MSCs shown in Figure 2.1 are said to be basic message sequence charts (bMSCs), which describe a finite interaction between a set of components [6]. In the *Initialize* scenario for instance, the *Control* instance is sending the message *on* to the *Sensor* instance. A bMSC does not necessarily convey an order to the messages. However, in our case, there is only one bMSC with more than one message. Therefore, the other ones have only one possible order of execution, which is sending their only message.

For the *Analyse* bMSC however, there are three messages that could lead to more possible orders of execution. In this case, it is important to note that an instance of a bMSC has to follow the order on which the messages are sent or received. For instance, the *Database* instance can only send the *data* message after the *query* message is received. Hence, this scenario only has one possible order as well. Formally, Definition 1 shows how bMSCs are defined by Uchitel et al. [6].

Definition 1 [Basic Message Sequence Chart] A basic message sequence chart (bMSC) is a structure $b = (E, L, I, M, instance, label, order)$ where: E is a countable set of events that can be partitioned into a set of send and receive events that we denote $send(E)$ and $receive(E)$. L is a set of message labels. We use $\alpha(b)$ to denote L. I is a set of instance names. $M : send(E) \to receive(E)$ is a bijection that pairs send and receive events. We refer to the pairs in M as messages. The function instance: $E \to I$ maps every event to the instance on which the event occurs. Given $i \in I$, we denote $\{e \in E | instance(e) = i\}$ as $i(E)$. The function label: $E \to L$ maps events to labels. We require for all $(e, e') \in M$ that $label(e) = label(e')$, and if $(v, v') \in M$ and $label(e) = label(v)$, then $instance(e) = instance(v)$ and $instance(e') = instance(v')$. $order$ is a set of total orders $\leq_i \subseteq i(E) \times i(E)$ with $i \in I$ and $\leq_i$ corresponding to the top-down ordering of events on instance i.

An extension of bMSCs are high-level message sequence charts (hMSCs), which provides the means for composing bMSCs [6]. These can be used to show the possible paths of execution of a system, that is, the possible continuations after each bMSC, in a way that it is visually and easily understood. As an example, in Figure 2.2 the hMSC for the *Boiler System* is shown. It is possible to observe that the scenarios are ordered in a way that the system delivers correct service. The formal definition of hMSCs by Uchitel et al. [6] is presented in Definition 2.

Definition 2 [High-Level Message Sequence Charts] A high-level message sequence chart (hMSC) is a graph of the form (N, E, s_0) where N is a set of nodes, $E \subseteq (N \times N)$ is a set of edges, and $s_0 \in N$ is the initial node. We say that n is adjacent to n' if $(n, n') \in E$. A (possibly infinite) sequence of nodes $w = n_0, n_1, ...$ is a path if $n_0 = s_0$, and n_i is adjacent to n_{i+1} for $0 \leq i < |w|$. We say a path is maximal if it is not a proper prefix of any other path.

Finally, the hMSC depicted in Figure 2.2 is a special kind of hMSC, which is called a Positive Specification (PSpec). Simply put, a PSpec contains an hMSC, a set of bMSCc, and a bijective function that maps one node from the hMSC to a single bMSC. The formal

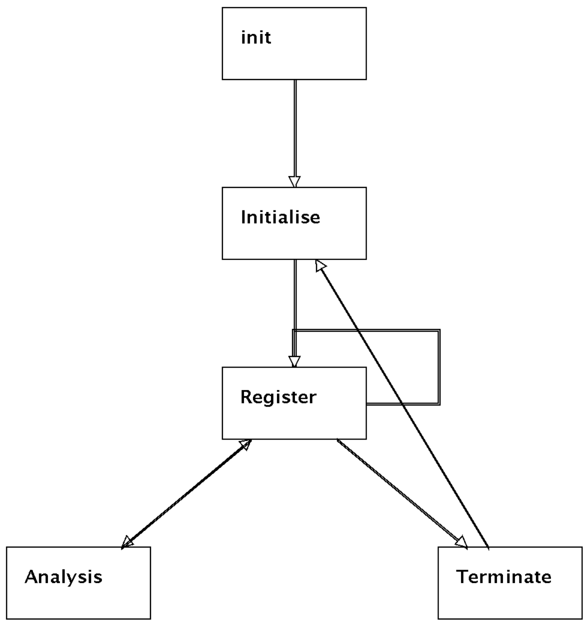


Figure 2.2: hMSC of the Boiler system specification.

definition by Uchitel et al. [6] is presented in Definition 3.

Definition 3 [Positive Message Sequence Chart Specification] A positive message sequence chart (MSC) specification is a structure $PSpec = (B, H, f)$ where B is a set of bMSCs, $H = (N, A, s_0)$ is a hMSC, and $f : N \to B$ is a bijective function that maps hMSC nodes to bMSCs. We use $\alpha(PSpec) = \{l | \exists b \in B.l \in \alpha(b)\}$ to denote the alphabet of the specification.

2.1.2 Labeled Transition System

A labeled transition system (LTS) is a finite state machine that has an intuitive easily grasped semantics and a simple representation [19]. They can be used to represent the expected order of messages exchanged by each component of a distributed system.

An LTS is a directed graph, with nodes and edges, where each node represents a state of the system, and each edge a transition from one state to another. Edges are labeled with the message(s) that are exchanged for that transition to happen. Finally, there is a unique node that represents the initial state of the system (state 0), which will be denoted in red. More formally, Definition 4 shows the definition of LTSs by Uchitel et al. [6].

Definition 4 [Labeled Transition Systems] Let $States$ be the universal set of states where state π is the error state. Let $Labels$ be the universal set of message labels and $Labels_\tau = Labels \cup \tau$ where τ denotes an internal component action that is

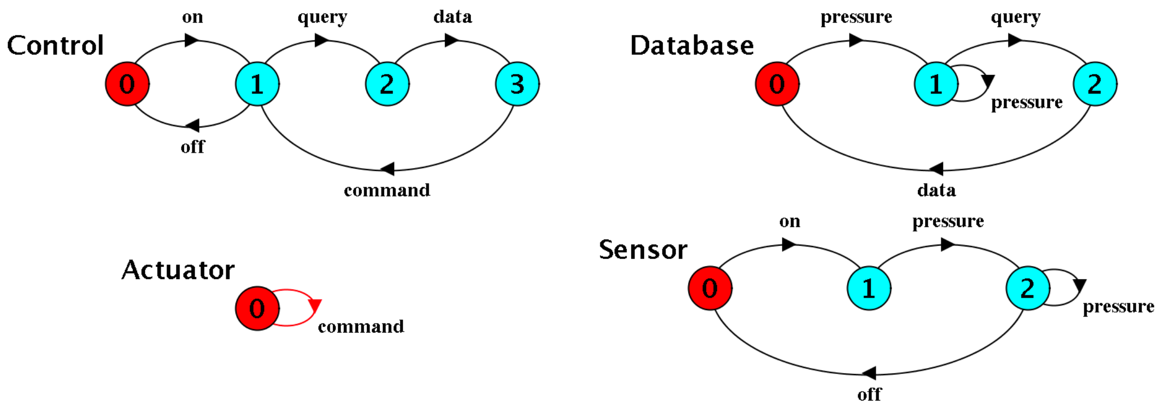


Figure 2.3: Each Boiler component's LTS representing the messages exchanged.

unobservable to other components. An LTS P is a structure $(S, L, \triangle, q)$ where $S \subseteq States$ is a finite set of states, $L = \alpha(P) \cup \tau$, $\alpha(P) \subseteq Labels$ is a set of labels that denotes the communicating alphabet of P, $\triangle \subseteq (S \backslash \{\pi\} \times L \times S)$ defines the labeled transitions between states, and $q \in S$ is the initial state. We use $s \xrightarrow{l} s'$ to denote $(s, l, s') \in \triangle$. In addition,we say that an LTS P is deterministic if $s \xrightarrow{l} s1$ and $s \xrightarrow{l} s2$ implies $s1 = s2$.

Figure 2.3 shows the LTS for each component of the Boiler system. For instance, the LTS for the *Sensor* shows that this component, the first message to be exchanged must be *on*, then *pressure*, and so on. Note that each LTS contains only the messages exchanged by that component (e.g., the LTS for the *Actuator* has only one state and one transition, because that component only exchanges one message throughout all scenarios).

Finally, it is possible to combine different LTSs by doing a parallel composition of them. The resulting LTS represents the expected order of messages in the entire system. Figure 2.4 shows the resulting LTS of the parallel composition of the LTSs in Figure 2.3. Formally, the definition of parallel composition by Uchitel et al. [6] is presented in Definition 5. The only exceptions to the rules presented are that state π is used instead of states (x, π) and (π, x) for all $x \in States$.

Definition 5 [Parallel Composition of LTS] Let P_1 and P_2 be LTSs where $P_i = (S_i, L_i, \triangle_i, q_i)$. Their parallel composition is denoted $P_1||P_2$ and is an LTS $(S, L, \triangle, q)$ where $S = S_1 \times S_2 \cup \{\pi, \epsilon\}$, $L = L_1 \cup L_2$, $q = (q_1, q_2)$, and $\triangle$ is the smallest relation in $(S\backslash\{\pi\}) \times L \times S$ that satisfies the following rules where $x \xrightarrow{a}_i y$ denotes $(x, a, y) \in \triangle_i$:

$$\frac{s \xrightarrow{a}_1 t}{(s, s') \xrightarrow{a} (t, s')}(a \notin \alpha(L_2)), \frac{s \xrightarrow{a}_2 t}{(s, s') \xrightarrow{a} (t, s')}(a \notin \alpha(L_1)),$$

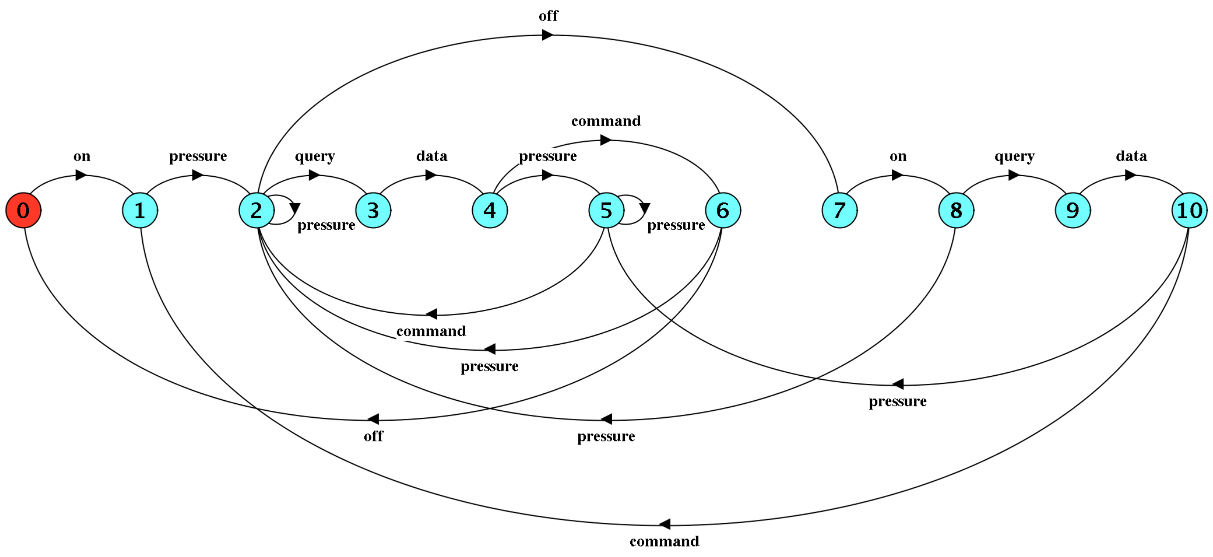


Figure 2.4: Resulting LTS of the parallel composition of the LTSs in Figure 2.3.

$$\frac{s \xrightarrow{a}_1 t \quad s' \xrightarrow{a}_2 t'}{(s, s') \xrightarrow{a} (t, t')} (a \notin (\alpha(L_1) \cap \alpha(L_2)) \backslash \{\tau\}).$$

Finite State Process Notation

The Finite State Process (FSP) is a notation used to specify the behavior of concurrent systems to the *Labelled Transition System Analyzer* (LTSA) tool[1] [?]. A FSP specification generates LTSs, such as the ones in Figures 2.3 and 2.4. FSP specifications contain two sorts of definitions: primitive processes (e.g., individual components) and composite processes (e.g., parallel compositions).

For primitive processes, only the notion of *states*, *action prefix*, and *choice* will be used. A *state* represents a state in an LTS, and is named Qi (i = 0,1,2...). It is denoted by Qi = (a), where a is either an *action prefix* or a *choice* and represents the existing transitions that leave this state. The *action prefix* represents a transition in an LTS, and is denoted by a -> b, where a is a message and b is either a message or a state. It indicates that after a is exchanged, the LTS will either change to state b or wait for message b to be exchanged. Finally, *choice* is denoted by $a \mid b$, where a is an *action prefix* and b is either an *action prefix* or a *choice* . It indicates that more than one transition exist leaving that state. Figure 2.5 shows the FSPs that generates the corresponding LTSs in Figure 2.3.

Finally, the only composite process used in this dissertation will be parallel composition. It is analogous to the parallel composition of LTSs, and it is detonated by $(a||b)$, where a is a primitive process, and b is either a primitive process or a parallel com-

[1]Available at: https://www.doc.ic.ac.uk/ltsa/.

```
Control = Q0,
    Q0  = (on -> Q1),
    Q1  = (off -> Q0
          |query -> Q2),
    Q2  = (data -> Q3),
    Q3  = (command -> Q1).

Actuator = Q0,
    Q0  = (command -> Q0).
```

```
Database = Q0,
    Q0  = (pressure -> Q1),
    Q1  = (pressure -> Q1
          |query -> Q2),
    Q2  = (data -> Q0).

Sensor = Q0,
    Q0  = (on -> Q1),
    Q1  = (pressure -> Q2),
    Q2  = (off -> Q0
          |pressure -> Q2).
```

Figure 2.5: Each Boiler component's FSP.

position. The FSP corresponding to the parallel composition of the Boiler components ($Boiler = (Control||Database||Actuator||Sensor)$) generates the LTS in Figure 2.4.

2.1.3 Implied Scenarios

An implied scenario (IS) is a scenario that was not included in the system's specification, but it occurs in every implementation of the specification [20]. It is a result from implementing actions that are global to the system, in a local level to the components that executes them. Because of this implementation, a component might not have enough information locally to decide whether or not the action should be prevented, therefore it is always performed.

An implied scenario can be classified as positive or negative [6]. A positive implied scenario is one that although it was not included in the system specification and its behavior was not expected, it has a desired behavior. Thus, a positive IS represents an unexpected but acceptable behavior. In this case, the system's specification is usually extended with this new scenario. On the other hand, a negative implied scenario is a scenario that was not expected and its observed behavior is harmful to the system's execution. That is, the system is not performing the correct service, or in other words, performing a failure. Thus, a negative IS represents an unexpected unacceptable behavior.

However, in the present work, this distinction will be disregarded, and therefore all ISs will be treated as a failure for simplicity since the characterization of a positive IS requires domain-expert knowledge. Although this can introduce an unnecessary cost to the system, as we might add constraints to the system in order to restrict behaviors that could instead be included, this analysis is not in the scope of this work, as our goal is to demonstrate the possibility of resolving various ISs at once.

Because of the nature of concurrent systems, implied scenarios may not happen in every system run, as messages are not synchronized and traces of execution (order of the

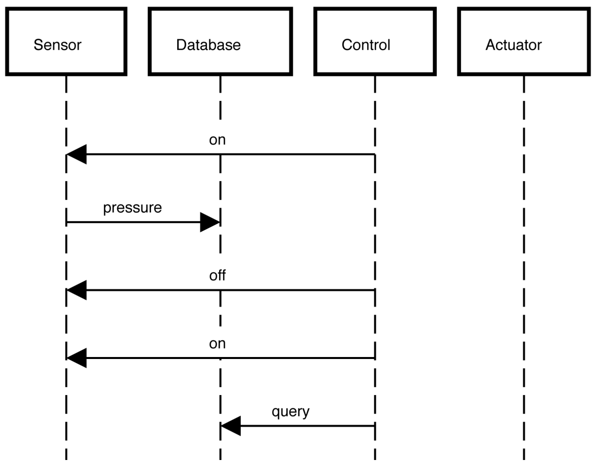


Figure 2.6: An implied scenario from the boiler system.

messages on the MSC) could be different, even though the same course of action is sought.

As an example, in Figure 2.6 an implied scenario in the *Boiler System* is presented. The unexpected part, that is, the cause of the implied scenario, is that *Control* tries to execute the *Analysis* scenario before a pressure from the current system run is registered. Therefore, it will tell the *Actuator* to variate the temperature according to a pressure that does not represent the system's current state.

Finally, to formally define what is an implied scenario, we use Definitions 6 to 10 from Uchitel et al. [6]. These definitions show that an implied scenario is a system execution that is not modeled in *PSpec* but arises in every architecture model of *PSpec*. That is, it is an unexpected execution that happens in all implementations of *PSpec*.

Definition 6 [Execution] Let $P = (S, L, \triangle, q)$ be a LTS. An execution of P is a sequence $w = q_0 l_0 q_1 l_1 ...$ of states q_i and labels $l_i \in L$ such that $q_0 = q$ and $q_i \xrightarrow{l_i} q_{i+1}$ for all $0 \leq i < |w/2|$. An execution is maximal if it cannot be extended to still be an execution of the LTS. We also define $ex(P) = \{w \mid w \text{ is an execution of } P\}$.

Definition 7 [Projection] Let w be a word $w_0 w_1 w_2 w_3 ...$ and A an alphabet. The projection of w onto A, which we denote $w|A$, is the result of eliminating from the word w all elements w_i in A.

Definition 8 [Trace and Maximal Trace] Let P be a LTS. A word w over the alphabet $\alpha(P)$ is a (maximal) trace of P if there is an (maximal) execution $e \in ex(P)$ such that $w = e|\alpha(P)$. We use $tr(e)$ to denote the projection of an execution on the alphabet of a LTS. We also define $tr(P) = \{w \mid w \text{ is a trace of } P\}$ and $L(P) = \{w|w$ is a maximal trace of $P\}$.

Definition 9 [Architecture Models] Let $PSpec = (B, H, f)$ be a positive MSC specification with instances I, and let A_i with $i \in I$ be LTSs. We say that an LTS A is an architecture model of $PSpec$ only if $A = (A_1|| \ldots ||A_n), \alpha(A_i) = \alpha(i)$, and $L(PSpec) \subseteq L(A)$.

Definition 10 [Implied Scenarios] Given a positive MSC specification $PSpec$, a trace $w \notin L(PSpec)$ is an implied scenario of $PSpec$ if for all trace y and for all architecture model A of $Pspec, w.y \in L(A)$ implies $w.y \notin L(PSpec)$.

2.2 Clustering

Another important background required in this work is the one regarding the notion of clustering. Clustering is a data-mining technique used to group datapoints in a dataset. In other words, it is a method to group elements in an unsupervised way. After obtaining the separate groups, the user still has to analyze the results and figure out why those elements were clustered together. A simple definition of the clustering process is given in Matteucci [1]: "the process of organizing objects into groups whose members are similar in some way".

A good clustering result is such that the cluster elements are very similar to each other, and very dissimilar to other clusters' elements. This way a good separation between clusters is observed and the elements within a cluster are clearly similar.

In Figure 2.7 a simple clustering process is shown, where on the left side of the image the elements are all separate, and on the right side, after clustering, four clusters can be clearly seen.

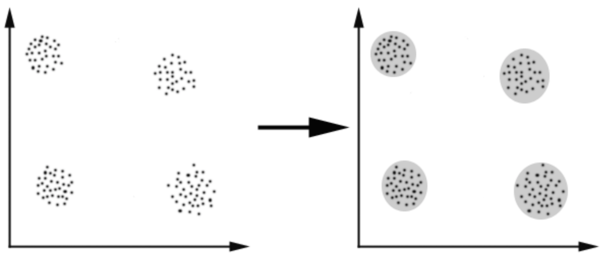


Figure 2.7: Example of a clustering technique, taken from [1].

2.2.1 Hierarchical Clustering

As defined by Ward [21], Hierarchical Grouping – later called Hierarchical Clustering by Johnson [22] – is *"a procedure for forming hierarchical groups of mutually exclusive subsets, each of which has members that are maximally similar with respect to specified characteristic"*. There are two types of hierarchical clustering [23]: agglomerative and divisive. The former starts with N clusters, containing 1 element each, and groups clusters one by one until there is only one cluster. The latter starts with 1 cluster, containing all N elements, and splits the existing clusters until there are N clusters, containing 1 element each.

The agglomerative hierarchical clustering is a method where members of a dataset (datapoints) are grouped hierarchically, with the most similar datapoints (or groups of datapoints) being merged before the less similar ones. This similarity is often measured by a distance metric, which means that the lower the score between two datapoints, the more similar they are.

This process is recursive [22] and consists of four steps: (i) calculate the similarity (or distance) between all members of the current dataset; (ii) find the most similar pair between those members; (iii) replace those two members with a new one, which merely is the two grouped together; (iv) go to (i) if there is more than one member in the current dataset. In the end, there will be a single member of the dataset, which is a group containing all individual datapoints that made up the original dataset. The interesting result, however, is being able to see the step-by-step grouping of members, which facilitates the detection of subgroups in the dataset.

The divisive hierarchical clustering is a method where all members start in the same cluster, and then are split into smaller clusters until each member is in a cluster by itself. One possible way to achieve this is by using the *DIvisive ANAlysis Clustering* (DIANA) [24], where the largest cluster is broken down in every step. First, the element e that is most dissimilar to the other ones is selected and removed from that cluster; Then, the remaining elements that are more similar to e than to the other remaining ones are also moved to that new cluster. This process is repeated until all elements have been separated.

However, because the initial merges of small-size clusters in the agglomerative approach correspond to high degrees of similarity, its results are more understandable than the ones obtained by the divisive approach [25]. Therefore, as the clustering results will serve as a reference to the user in our methodology, it is essential that its results be understandable and help the user to analyze the elements grouped. Thus, the agglomerative approach is used and from here on hierarchical clustering will be used to refer to the agglomerative approach.

Coordinates of example datapoints in x and y axis.

	A	B	C	D	E
x	1	0	2	4.5	6
y	2	1	1	6	6

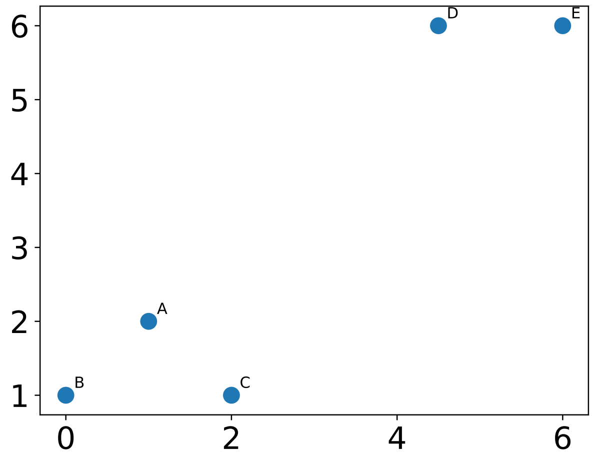


Figure 2.8: Plot of example datapoints.

As an example, let us consider five datapoints in a 2-dimensional plot. These points are represented in Figure 2.8 and their coordinates are shown in Table 2.1. To apply the algorithm, it is required to have the similarity between the elements. Table 2.2 shows the Euclidean distance between these points. With this information, the first pair to be grouped could be either A and B or A and C, as they have the smallest distance between them with 1.41. Without loss of generality, A and B will be the first elements to be grouped.

The question now is how to calculate the similarity between a group of datapoints ({A, B}) and other datapoints. In fact, there are multiple ways to achieve this, such as: (i) *complete linkage clustering* [26], which calculates the distance between a group of elements E and another element e' as the maximum possible distance of $e \in E$ and e'; (ii) *single linkage clustering* [26], where the distance between a group of elements E and another element e' is the minimum possible distance of $e \in E$ and e'; and (iii) *Ward's*

Euclidean distances between example datapoints.

	A	B	C	D	E
A	0				
B	1.41	0			
C	1.41	2.0	0		
D	5.32	6.73	5.59	0	
E	6.40	7.81	6.40	1.5	0

Euclidean distances after first recursion.

	{A ,B}	C	D	E
{A, B}	0			
C	2.0	0		
D	6.73	5.59	0	
E	7.81	6.40	1.50	0

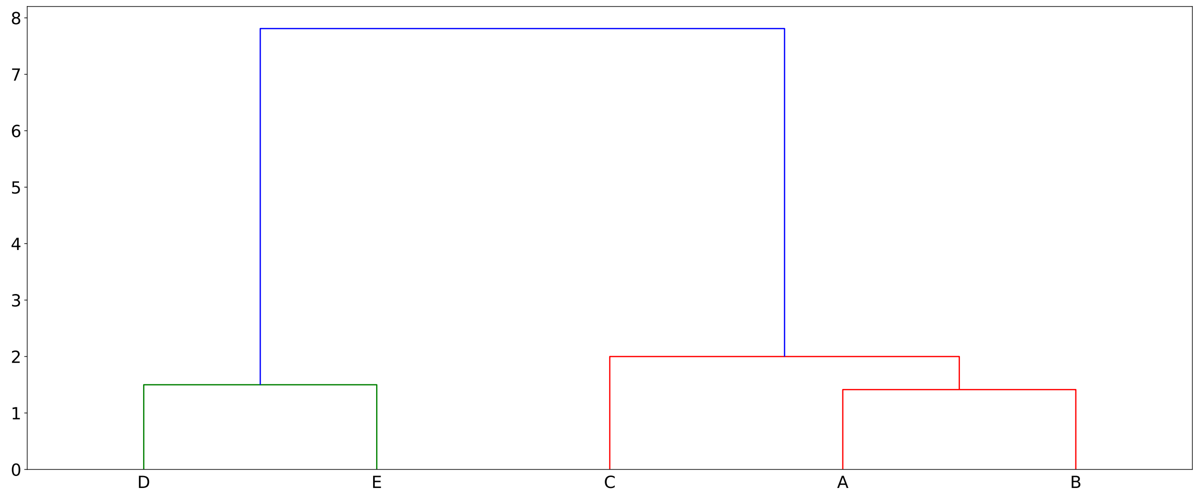


Figure 2.9: Dendrogram showing the order of grouping.

method [21], which calculates the increase in variance if two clusters were merged. The method used to exemplify will be the “complete linkage clustering” [26].

By using the complete linkage method, the highest distance between a member of the group and another member is kept, that is, the distance between {A, B} and C will be 2.0, as that is the maximum value between 1.41 (dist(A, C)) and 2.0 (dist(B, C)). This way, the distances after the first recursion are shown in Table 2.3. The process is then repeated with these new distances, and the next members to be grouped will be D and E, as these datapoints have a distance of 1.50, which is the lowest distance on this current dataset.

The final result of the whole process is shown in Figure 2.9. This kind of graph is called a dendrogram, which provides a useful visual way to analyze hierarchical clusters, allowing a different analysis of the dataset. For instance, on this example, it is clear that two groups are formed, as A, B, and C are much closer to one another than they are to D and E. Similarly, D and E are more similar to each other than to the other 3 points.

2.3 Smith-Waterman Algorithm

The Smith-Waterman algorithm (SW), first introduced by Smith & Waterman in [17], proposes to *"find a pair of segments, one from each of two long sequences, such that there is no other pair of segments with greater similarity"* [17]. In other words, SW tries to find the best local alignment between two sequences, local in the sense that this alignment can be shorter than the sequences, that is, it might find only a small part of the sequences where they are most similar. It is widely used in bioinformatics research to calculate the similarity between genetic sequences.

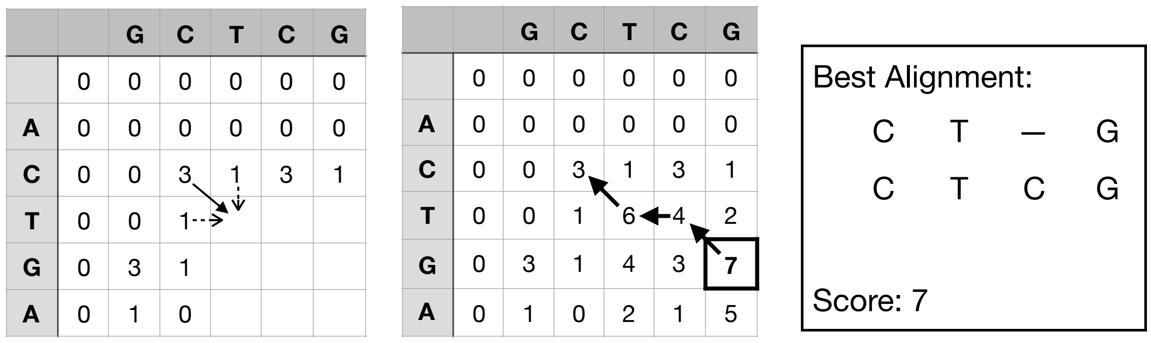
The algorithm can be broken down into two steps: (i) calculate the scoring matrix; and (ii) traceback the best alignment from the highest value in the matrix.

Therefore, firstly it is needed to calculate the scoring matrix (SM). The SM is a $(n+1)$ by $(m+1)$ matrix, where n and m are the lengths of the sequences to be compared. The first column and row are filled with zeros, while the rest of the matrix is filled according to the recurrence equation shown in Equation (2.1), where, A and B are the sequences being compared, and $s(A_i, B_j)$ checks if A_i and B_j are the same element. If they are it returns $MATCH$, if not it returns $MISMATCH$. Lastly, GAP, $MATCH$, and $MISMATCH$ are defined by the user.

$$SM_{i,j} = max \begin{cases} SM_{i-1,j} + GAP \\ SM_{i,j-1} + GAP \\ SM_{i-1,j-1} + s(A_i, B_j) \\ 0 \end{cases} \tag{2.1}$$

Simply put, each cell $SM_{i,j}$ is calculated with basis on previously calculated values – $SM_{i-1,j}$, $SM_{i-1,j-1}$, and $SM_{i,j-1}$ – but only one of these values is used at most. If the upper-left diagonal value ($SM_{i-1,j-1}$) is used, then it indicates that both sequences are reading their elements (i.e., A_i, B_j); thus, it is considered whether $A_i = B_j$, and a value ($MATCH$ or $MISMATCH$) is used to calculate $SM_{i,j}$ accordingly. However, if the upper value ($SM_{i-1,j}$) is used, only A is reading its element, as it goes from A_{i-1} to A_i while B_j is constant. Analogously, for the left value ($SM_{i,j-1}$) A_i is constant and B_{j-1} changes to B_j. For these two last cases, a penalty (GAP) for not considering elements from both sequences is used to calculate $SM_{i,j}$.

After the whole SM has been populated, we can find the best local alignment. This is achieved by doing a traceback from the highest value found in the SM, which is the score of the best alignment. First, we have to find the highest value in SM, which represents the end of the best alignment. Let us assume (k, l) is such position. Starting from this position, the traceback is executed by finding the highest value among $SM_{k-1,l}$, $SM_{k-1,l-1}$,



		G	C	T	C	G
	0	0	0	0	0	0
A	0	0	0	0	0	0
C	0	0	3	1	3	1
T	0	0	1			
G	0	3	1			
A	0	1	0			

(a) single cell calculation

		G	C	T	C	G
	0	0	0	0	0	0
A	0	0	0	0	0	0
C	0	0	3	1	3	1
T	0	0	1	6	4	2
G	0	3	1	4	3	7
A	0	1	0	2	1	5

(b) scoring matrix

Best Alignment:

C	T	–	G
C	T	C	G

Score: 7

(c) alignment and score

Figure 2.10: Example results for the Smith-Waterman algorithm.

and $SM_{k,l-1}$. The cell with the highest value among those is a part of the alignment, and thus it is included in the traceback. This procedure is repeated with each new cell included until the next new cell has a value of zero. Finally, the alignment is the reversed sequence of cells included (as the first cell represents the end of the alignment), where movements in the horizontal consider only one sequence, vertical movements consider the other sequence, and diagonal movements consider both sequences.

As an example, let A = ACTGA, B = GCTCG, $MATCH$ = 3, $MISMATCH$ = -3, and GAP = -2. Figure 2.10a shows the calculation for a single cell, in this case, $SM_{4,4}$. The arrows represent which neighbors cells that can help to fill this one. From the left ($SM_{4,3}$) and above ($SM_{3,4}$) cells, the GAP penalty would apply, so both send a value of 1+(-2). For the diagonal cell ($SM_{3,3}$), we need to check if A_i is equal to B_j. In this case, both are 'T', and thus, we would use the $MATCH$ value. Therefore, this cell would send 3+3. Now that we have calculated all possible values using the neighbors, we pick the maximum from {-1, -1, 6, 0} and update this cell.

The whole scoring matrix is shown in Figure 2.10b. The highlighted cell is the highest value calculated. Thus we start from that position (4,5). From there on, we find the highest values among the top, left, and upper-left neighbors and do so until we get to a cell filled with 0. The arrows demonstrate this process until the last non-zero value is found.

After this traceback, we can find the best alignment and score. The score is merely the highest value in the scoring matrix, which is 7 in this case. To find the alignment, we reverse the traceback found and add the positions read from A and B. Note that if the move is either horizontal or vertical, then a GAP must be added. Figure 2.10c shows both the score and the best alignment found.

Although in this dissertation SW will be used to find similar sequences of messages instead of genetic ones, no adjustments are required from the original algorithm. For instance, two sequences of messages from A' and B' the Boiler System could be aligned by simply changing the input from A and B to A' and B' in the above example.

Chapter 3

Related Work

The work that originally introduced Implied Scenarios (ISs) was done by Alur et al. [27]. They proposed a method to detect a single IS given a set of basic message sequence charts (bMSCs) but did not support high-level message sequence charts (hMSCs), thus eliminating the analysis of infinite system behaviors. After an IS was detected, the user has to adjust the system specification and can do a new search for ISs. One of the main differences from our approach is that we allow the user to use hMSCs, which allows the user to specify infinite system behaviors. Also, our approach can collect multiple scenarios at once, which is not possible with their approach. Finally, our approach goes into further analysis regarding the implied scenarios by trying to find common behaviors among them. By these means, we can reduce the problem space that the user has to analyze.

Uchitel et al. [6, 8, 20, 28, 29] later extended the work by Alur et al. [27]. Their work used hMSCs to accommodate loops and compositions of bMSCs, allowing infinite system behaviors. The detection process also detected only one IS at a time, but allowed the user to include or remove the detected IS automatically, if the user classifies it as positive or negative. Their work, however, may suffer from state explosion problem. Although our proposed methodology extends the detection process of the works by Uchitel et al. for the first steps, we further advance on the analysis process by finding common behaviors among multiple implied scenarios.

Letier et al. [30] continue Uchitel et al. work with the notion of *Input-Output implied scenarios* (I/O ISs). I/O ISs are a particular kind of ISs, where components are restricting messages that were supposed to be only monitored. According to [30], however, they cannot do so. Because this approach is based on Uchitel et al. [6], it shares the same drawbacks when compared to our proposal. Our work goes further on the analysis process by finding common behaviors among multiple implied scenarios, besides being able to collect multiple implied scenarios automatically. However, they detect a special kind of implied scenarios that we do not; the I/O implied scenarios.

Muccini [7] proposes a different method to detect ISs, by analyzing non-local choices. Succinctly, non-local choices are points in an MSC specification where a component is not sure about which message to send, as it does not have the necessary information. His approach does not suffer from state explosion, because it applies a structural, syntactic, analysis of the specifications as opposed to the behavioral, model-based analysis by Uchitel et al. [6]. He also claims to discover and display all the implied scenarios in the system specification with a single execution. However, all ISs detected are due to non-local choices, and since "non-local choices are special cases of implied scenarios" [29], not all implied scenarios might have been found. Furthermore, he does not go into further analysis, whereas our approach searches for common behaviors among the implied scenarios, which can lead to a considerable reduction in the problem space.

Castejón and Braek [31] propose a different method to detect ISs, by using UML 2.0 collaborations [4]. In such collaborations, components can be offered roles to execute actions. However, each component only plays ideally one role at a time. Their approach analyses the collaboration diagrams, searching for points where components are offered roles while they are already busy playing another. Therefore, these points represent ISs in the specification. However, their approach also detects only one IS at a time and does not further analyze the results obtained.

Chakraborty et al. [13] proposes a similar method to detect ISs as Uchitel et al. [6]. The difference lies in that they extend the traces of execution (i.e., sequences of messages exchanged) until the end of the system execution, whereas in the approach by Uchitel et al. the traces are stopped at the message an IS is detected. However, their approach only detects ISs, as it does not further analyses the result nor suggests solutions. Finally, it is proved in this work that "the problem of detecting implied scenarios for general MSG's is undecidable", where a Message Sequence Graph (MSG) is merely an hMSC.

The work by Song et al. [5] proposes a method to not only detect ISs but also find which point in the specification leads to them. It supports different kinds of communication between components, unlike all the above works which use synchronous communication, and also supports I/O implied scenarios. They propose a method to detect ISs by using graphs that represent the required order of messages for each component. Thus, when inconsistencies are found, they represent ISs, and edges that make such inconsistencies are pointed as the causes. Therefore, this approach can point the user to the fault underlying the implied scenarios. However, their work does not find similarity among the implied scenarios. Thus, it does not allow the user to reduce the problem space that has to be analyzed.

Moshirpour et al. [16, 32] propose another method to detect ISs. They model states for each agent (i.e., component) that represents their local views during each step of the execution – “let the current state of the agent to be defined by the messages that the agent needs in order to perform the messages that come after its current states” [32]. Then, they analyze all modeled states to find special states where “the agent becomes confused as to what course of action to take” [16]. These special states are similar to non-local choices, where an agent does not have enough local information to decide the correct action. Therefore, ISs occur when there exist such special states. Their approach detects one IS at a time, does not suggest solutions to detected ISs, and does not go into further analysis after ISs are successfully detected.

Fard et al. [33] propose a different method to detect ISs. They cluster the interactions between components to find points in the specification where a lack of information might happen. Also, this is the first work in the literature that classifies implied scenarios into different categories, and they use such classifications to propose a generic architecture refinement method for them. Their approach also points to the probable cause of ISs, similarly to Song et al. [5]. However, their work does not find similarity among the implied scenarios; thus it does not allow the user to reduce the problem space that has to be analyzed.

Reis [34] proposes a method to generate test cases based on detected implied scenarios of a system. Similarly to our methodology, their methodology uses the approach by Uchitel et al. to detect the implied scenarios and tries to group the implied scenarios in order to generate fewer test cases than one for each implied scenario. This grouping is achieved by converting graphs of the exchanged messages to basic regular expressions, that is, the graphs are converted into sequences where the loops are annotated, so it is possible to find implied scenarios that differ from each other because of loops. However, the implied scenarios are detected and analyzed on a one-by-one basis, which does not restrict the problem space the user needs to analyze. Furthermore, no solutions are suggested to resolve the implied scenarios.

Table 3.1 shows the key points of comparison among existing approaches and our proposal. Most of these works focus on the detection of ISs; however, few of them do a further analysis after the detection of the ISs. Taking this into consideration, our methodology does not introduce a novel method to detect ISs, as there are many works with this goal that can be used. Its focus lies on the analysis after detecting the ISs, which can be improved to require less time and effort from the user to achieve better results.

Table 3.1: Related work comparison

Research	**Year**	**Type of detected errors**	**Modeling**	**Solutions suggested**	**Restrict problem space analysis**
Alur et al.	1996	implied scenario	State Machine	x	x
Muccini	2003	various implied scenarios due to non-local choices	Modeling NL choice	x	x
Uchitel et al.	2004	implied scenario	FSP + LTS	arch. refinement, FSP constraint	x
Letier et al.	2005	implied scenario	FSP + LTS	arch. refinement, FSP constraint	x
Castejón and Braek	2006	implied scenario	Sub-role sequences	x	x
Chakraborty et al.	2010	implied scenario	State Machine	x	x
Song et al.	2011	implied scenario	Graph comparison	provides reasons	x
Moshirpour et al.	2012	implied scenario	State Machine	x	x
Fard and Far	2014	async. concatenation, various implied scenarios	Interaction Modeling	generic arch. refinement	x
Reis	2015	implied scenario	FSP + LTS	x	x
Our Work	**2018**	**various implied scenarios, common behaviors among ISs**	**FSP + LTS**	**characterization, arch. refinement, FSP constraint**	**yes**

Chapter 4

Methodology

In this chapter, our proposed methodology will be laid out and explained in details. First, in Section 4.1 an overview of the entire methodology is presented, including step-by-step details. In Section 4.2, the steps that require more in-depth explanations are thoroughly detailed. Finally, in Section 4.3 one guiding example is used to exemplify our approach.

4.1 Overview

Our proposed methodology consists of seven steps, which are shown in Figure 4.1. The methodology starts at step 1, where the user models the scenarios of the system. This step requires user interaction, as he/she needs to use their domain expertise to model such scenarios correctly. As a proof of concept, we used the LTSA-MSC tool [28] to allow the user to perform this step. After the system is modeled, it is possible to detect implied scenarios (ISs) in the specification, which is also achieved by using the LTSA-MSC tool through Uchitel et al. [6] approach.

However, in order to detect various ISs without the need of user input after each one, the detection process of the LTSA-MSC tool was adapted [1]. The adapted version keeps the original detection process of the LTSA-MSC tool. However, instead of interacting with the user after each IS is detected, as it happens with the original tool, the adapted version iteratively collects various ISs and exports all detected ISs to a file, without the need of user input throughout this process.

Therefore, in step 2, various ISs are collected from the system specification and exported. After this process completes and exports the ISs, the next step (3) is to detect common behaviors (CBs) among the ISs. Nonetheless, it is possible that no IS is detected in the system modeled by the user. If that is the case, and no ISs were collected, the process finishes, as there are no elements to be analyzed.

[1]The adapted version is available at https://github.com/cbdm/Implied_Scenarios.

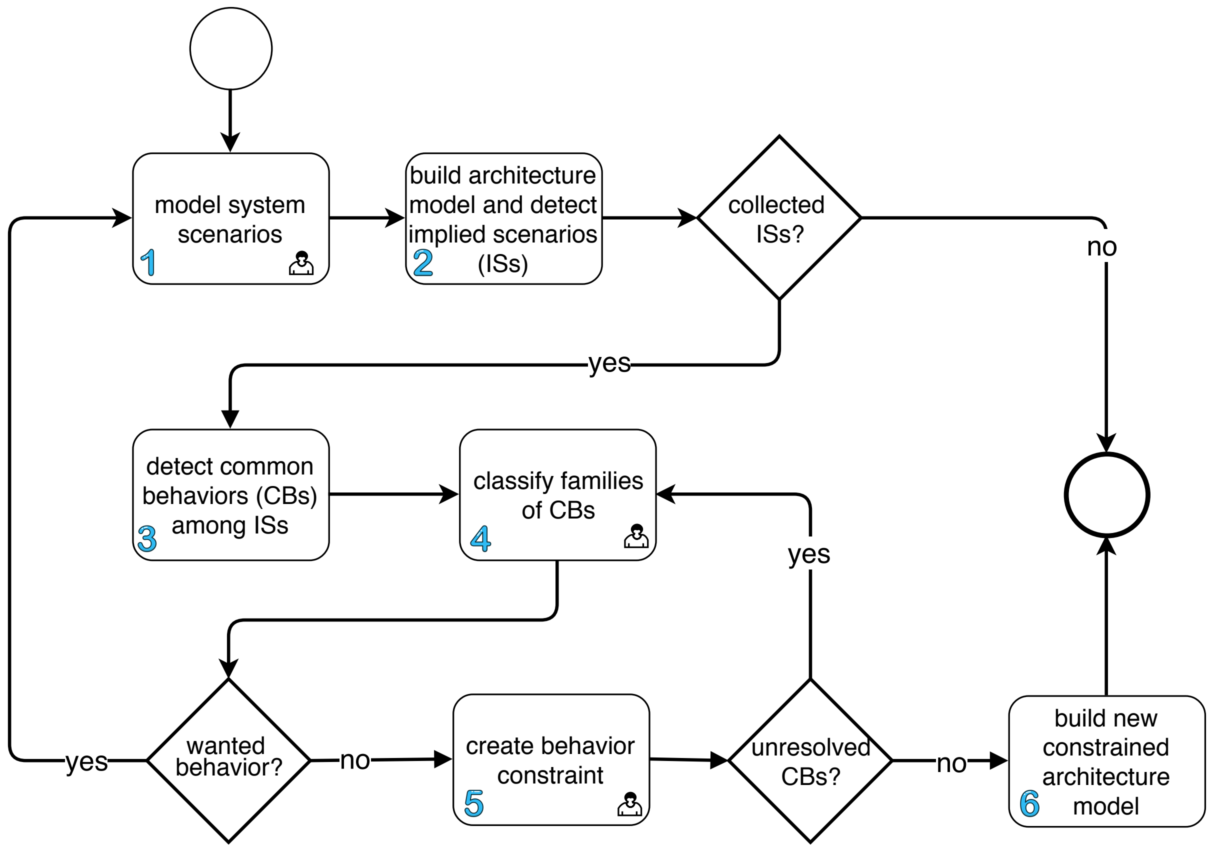

model system scenarios
1
build architecture model and detect implied scenarios (ISs)
2
collected ISs?
no
yes
detect common behaviors (CBs) among ISs
3
classify families of CBs
4
yes
wanted behavior?
yes
no
create behavior constraint
5
unresolved CBs?
no
build new constrained architecture model
6

Figure 4.1: Steps of the proposed methodology.

If there were collected ISs, in step 3 the CBs among the ISs are detected. The CBs are groups of ISs share common traces among them. Because the LTSA-MSC tool produces the implied scenarios in the form of error traces [5], that is, sequences of exchanged messages until an error occurs, the CBs are defined as shared sequences of messages among various ISs. This step is further explained in Section 4.2.1.

Next, step 4 first finds the similarities between the detected CBs. Because the CBs are sequences of messages, the SW algorithm will be used to find the most similar CBs. After finding the similarity, the CBs are hierarchically clustered to find groups of similar CBs; thus facilitating the manual user analysis. Following, the user manually analyses the results. The similarities and clustering of the CBs allow he/she to classify families of CBs that have the same cause and thus can be resolved together. Section 4.2.2 further explains the process of finding similarities among CBs and the classification of families of CBs.

After the user manages to classify a family of CBs, he/she needs to analyze if that family contains positive or negative CBs, that is, if the behavior the family represents is wanted or unwanted. If it is a positive behavior, then the ISs of the CBs were wanted scenarios that were overlooked [8] during system modeling. Thus, the user needs to go back to the modeling (step 1) and add new scenarios that represent the family of CBs. However, if the family represents a negative behavior, then he/she needs to remove the CBs from the system, which is achieved by creating constraints [6] in step 5, which are added to a list of constraints. This treatment process is repeated while there are CBs that have not been resolved. That is, there are CBs that have not been included in the specification nor prevented with constraints. The treatment of families of CBs is further explained in Section 4.2.3.

Finally, after all families of CBs have been dealt with, a new architectural model is generated in step 6, which no longer contains the detected families of CBs, by conducting a parallel composition of the architectural behavior LTS with the LTS for the constraints created. This results in a constrained architectural model, which does not allow for the previously collected ISs to happen.

4.2 Detailed Steps

In this section, the novel steps that require further explanations are presented.

4.2.1 Detecting Common Behaviors

The detection of common behaviors (CBs) is based on the hypothesis that whenever the same message is exchanged, the system reaches a same abstract state of correctness (a non-error state). In other words, if a message m is exchanged more than once, it did not take the system to different abstract states, unless it led the system to an error in one of its occurrences. Consequently, other messages that were exchanged between the different occurrences of m did not impact the system considerably, as the system was able to reach the same abstract state again.

These assumptions are made because LTSA-MSC produces the implied scenarios in the form of error traces [5]. That is, the tool used to collect ISs detects them as sequences of exchanged messages until an error occurs, which means that the last message caused the IS. This last message is called the *proscribed message* [29]. Therefore, the messages exchanged before the proscribed message do not lead the system to an error state, and thus their repetition is not relevant for finding the common behavior of an IS, because they keep the system in an abstract non-error state.

Hence, the messages that happen between repeated messages in the common behavior are removed, which results in the removal of loops of messages. That is, each message appears at most once in a common behavior. The single exception to this is the proscribed message, which is added without checking for repetitions. With that in mind, a common behavior is defined in Definition 11.

Definition 11 [Common Behavior] Given a set of Implied Scenarios S, if there is a minimal trace of execution (that includes the initial state of $L(Spec)$) c, where $\forall s \in S, c \supseteq s$ and $c \notin L(Spec)$, c is said to be the common behavior among elements of S.

The detection of CBs among ISs is performed by mean of Algorithm 1. The algorithm takes as input a list of ISs and outputs a list of CBs. In line 2, an empty list CBs is initialized, which will store the detected CBs. Line 3 starts a loop that will be executed for each IS in the list that was taken as input. Therefore, for each IS, an empty behavior *current_behavior* is created in line 4.

Next, as i is a sequence of messages, it starts a loop over the messages of IS in line 5. In line 6 it is checked if each message *message* has been already included in *current_behavior*. If it has not been included yet, that is, *message* is a new message, then it is added to the end of *current_behavior* in line 15. If *message* is already included in *current_behavior*, then in line 7 it is checked if *message* is the last message of the IS (i.e., the proscribed message). If *message* is the proscribed message, it is added to the end of *current_behavior* in line 8. However, if *message* has already been included in *current_behavior* and is not the proscribed message, the loop in lines 10-12 removes all messages between the current occurrence of *message* and the previous one, which is achieved by removing the last message of *current_behavior* until the last message is

ALGORITHM 1: Common behaviors detection process

```
1  findCBs ($ISs$)
       input  :  ISs – a list of implied scenarios
       output:  CBs – a list of common behaviors
2      $CBs = []$;
3      foreach $IS \in ISs$ do
4          $current\_behavior = []$;
5          foreach $message \in IS$ do
6              if $message \in current\_behavior$ then
7                  if $message = IS.last\_message()$ then
8                      $current\_behavior.append(message)$
9                  else
10                     while $current\_behavior.last() \neq message$ do
11                         $current\_behavior.remove\_last()$
12                     end
13                 end
14             else
15                 $current\_behavior.append(message)$
16             end
17         end
18         if $current\_behavior \notin CBs$ then
19             $CBs.append(current\_behavior)$
20         end
21     end
22     return $CBs$
```

message.

After the loop for each message in *IS* ends in line 17, *current_behavior* is the CB of *IS*. Therefore, in line 18 it is checked if *current_behavior* is already included in the list of common behaviors, and if it is not, *current_behavior* is added to *CBs*. Finally, after the loop for each IS ends in line 21, the algorithm returns the list *CBs*, which contains the unique CBs among the ISs in line 22. Additionally, an example of the application of this algorithm is provided in Section 4.3.2.

4.2.2 Classifying Families of Common Behaviors

After the common behaviors are filtered out among the collected ISs, it is necessary to check if any similar ones could be resolved together. In our proposal, hierarchical clustering is used to group detected common behaviors. By doing so, it helps the user to analyze which groups are similar, as hierarchical clustering shows the grouping order of the elements, and consequently, which common behaviors should be considered similar before analyzing other pairings.

However, it is first needed to define a similarity score between common behaviors. It is important to use a scoring method that is sensitive to the order of messages, as they represent a sequence of system states that lead to an error. Nevertheless, it is also essential to consider that due to the concurrent nature of the systems studied, the ordering of messages might be partially different in different executions of the same behavior. Therefore, the Smith-Waterman algorithm [17] is used, as it returns the best local alignment between two common behaviors and a score for that alignment, which illustrates what similarities they have while allowing the addition of *gaps* in the sequences, which helps to accommodate the matching of the same message in different orders. However, contrary to the metrics usually used in hierarchical clustering, the most similar the pair is, the higher the score will be. Thus, the dissimilarity function in Equation (4.1) is used.

$$dissimilarity(cb1, cb2) = \frac{1}{SW(cb1, cb2) + \varepsilon} \tag{4.1}$$

Because our dissimilarity function should return lower scores for the more similar pairs, and it is based on the Smith-Waterman algorithm score, which returns higher scores for the most similar pairs, the inverse of the SW score ($\frac{1}{SW}$) is used. The use of the inverse is possible because SW always returns an integer score ≥ 0, thus there are no negative values. Additionally, to avoid a possible division by zero, an insignificantly small non-zero value ε is added to the SW score.

The values 3, -3, and -2, were respectively defined for *MATCH*, *MISMATCH*, and *GAP* in an empirical manner. As "in a distributed system, it is sometimes impossible to say that one of two events occurred first" [35], it makes sense that a gap is penalized less than a mismatch in our domain. This happens because when analyzing sequences of exchanged messages, the same messages might appear out of order in different sequences, and a gap might indicate merely an out of order execution. Conversely, in bioinformatics mismatches are usually penalized less than gaps [36], as gaps are considered to be rarer than mismatches [37].

By using the dissimilarity function, a matrix of dissimilarities is then created containing the dissimilarities between all pairs of common behaviors detected in a system. This matrix is then used alongside Ward's method [21] to hierarchically cluster the common behaviors. Finally, a dendrogram showing the order of grouping is then exported, as well as the alignments found by the Smith-Waterman algorithm.

These results allow the user to manually identify clusters of common behaviors that are so similar that it is possible to resolve them together, which we call *families of common behaviors*. A family of CBs is formally defined in Definition 12. The treatment can be either an architectural refinement that includes CBs in $PSpec$ (e.g., the inclusion of new scenarios to the system specification), or a constraint that removes CBs from $PSpec$. The treatment will be further explained in Section 4.2.3. Finally, a treatment resolves a CB if after the treatment is applied to the system, the ISs that constitute the CB are not observed in the system.

Definition 12 [Family of Common Behaviors]
Given a cluster of Common Behaviors C, if there is a treatment t that $\forall c \in C$, t resolves c, then C is said to be a family of common behaviors.

4.2.3 Treating Families of Common Behaviors

After that the families of common behaviors are known, we need to deal with them in some way. According to [6], there are positive and negative implied scenarios. Positive ISs, are scenarios that were overlooked during the design of the system, that is, they are acceptable scenarios that were not included. This kind of IS can be treated by merely including the acceptable behaviors in the system's specification. On the other hand, negative ISs are unwanted behaviors. This kind of IS needs to be treated in a different manner, where guarantees that they will not happen are added to the specification.

Consequently, families of CBs can also be positive or negative, as they can be classified the same way as the ISs that constitute it. Therefore, if a family of CBs is positive, it can be resolved with an architectural refinement, which is the inclusion of the behavior the family describes in the original model of the system. However, if a family of CBs is negative, it has to be removed from the system model, which is achieved by creating LTS

ALGORITHM 2: Checks if a trace can be reached in an LTS.

```
1  trace_check ($\lambda, \iota$)
       input  : $\lambda$ – an LTS $(S, L, \triangle, q)$, $\iota$ – a trace
       output: reached – a boolean indicating if $\iota$ happens in $\lambda$
2      $reached = True$;
3      $cs = q$;
4      foreach $message \in \iota$ do
5          if $\exists ns | (cs, message, ns) \in \triangle$ then
6              $cs = ns$;
7          else
8              $reached = False$;
9              $break$;
10         end
11     end
12     return reached
```

constraints [29]. An LTS constraint is an LTS that when composed with the architectural model, removes unwanted behaviors.

In our methodology, although, we believe it would be possible to create the constraints automatically after the user has classified the families, the creation of such constraints has not been automated yet. Because the classification of families of CBs is a manual process, the user needs to use their domain knowledge to create the constraints. Therefore, the creation of the constraint is prone to human error. Hence, to make sure that the constraints indeed remove the collected ISs, a script that analyzes LTSs was developed. This script is shown in Algorithm 2, and tests whether a trace can happen in the constrained model.

Algorithm 2 receives an LTS λ and a trace ι as input, and checks if ι can happen in λ. Lines 2 and 3 are initializations, as in line 2 the return variable *reached* is initialized as $True$, and in line 3 the current state *cs* is initialized as the initial state q of λ. Next, it starts a loop (line 4) that goes through each message of ι.

For each message, it checks if a transition from the *cs* to a next state *ns* exists (line 5). If there is such a transition, it moves the current state to the next one (i.e., $cs = ns$) in line 6. If there is not, it sets *reached* to $False$ in line 8, and as there is no transition for the current message, the loop is broken, because there is no transition from *cs* labeled with $message$ (line 9).

After the loop is finished (line 11), it returns *reached* (line 12). *reached* is $True$ if there were always a next state *ns* for each message in ι, and thus ι can happen in λ, or $False$ if the ι cannot happen in λ.

By doing so, it is possible to verify that the traces of collected ISs have been removed from the system model, and thus the ISs will not happen at runtime. However, it is not enough to check that unwanted behaviors are removed, as it is also vital to verify that the expected behaviors are preserved. Therefore, it is essential to check if the traces of expected behaviors are also reached in all system models, which can also be achieved with Algorithm 2. These traces of expected behaviors are generated utilizing Algorithm 3.

Algorithm 3 receives a positive specification $PSpec$ and outputs a subset of all expected behaviors of the specification. However, because it can be impossible to list all behaviors due to the loops allowed, the only loops considered are the ones of the type $S \rightarrow S' \rightarrow S$, which are unrolled once. Firstly, the algorithm initializes the variables *expected*, which contains the behaviors, *next*, which contains the next nodes of H to be visited, and *visited*, which contains the nodes of H already visited.

Next, the main loop goes through all nodes of H while there are unvisited nodes, in line 5. It removes the next node to be visited and stores it in *current*, in line 6, includes the nodes that are reachable from *current* in *next*, in line 7, and finally includes *current* in the set of already visited nodes, in line 8. A loop goes through each simple path *sp* (i.e., a sequence of nodes without loops) that reaches the *current* node from the initial node s_0, in line 9. All paths in *expected* that are a prefix of *sp* are removed from *expected* in lines 10 and 11, and then *sp* is added to *expected* in line 14 if it is not a prefix of any other paths in *expected*. This makes sure that all paths included are maximal paths, as the non-maximal ones are extended with each new *sp* considered.

Lastly, the loops $S \rightarrow S' \rightarrow S$ are included in the behaviors encountered. A new set is created – *looped_behaviors* –, so that the new paths are not included in the set being analyzed, as that would generate an infinite loop, in line 18. This is achieved by going through each pair of nodes (s_i, s_j) that are connected with edges (s_i, s_j) and $(s_j, s_i) \in E$, in line 19. Next, another loop goes through each path *sp* that contains s_i, in line 20, and expands each position of *sp* that contains s_i to s_i,s_j,s_i and includes it in *looped_behaviors*, in lines 21 and 22. These new behaviors are included in *expected*, in line 25, which is then returned in line 26.

Finally, as a proof of concept, all implied scenarios collected throughout this dissertation are considered to be negative. This way, even though acceptable behaviors that might have been simply overlooked were removed, it shows that it is possible to resolve all detected unexpected behaviors applying the same treatment. Therefore, the only kind of treatment used were FSP constraints. There are examples of the creation of constraints further in Section 4.3 and Chapter 5.

ALGORITHM 3: Lists the expected traces of the expected behaviors of a system positive specification.

```
1 expected_behaviors ($PSpec$)
      input  : $PSpec$ – a positive specification $(B, H, f)$, where $B$ is a set of bMSCs,
               $H$ is an hMSC $(N, E, s_0)$, and $f$ is is a bijective function that maps
               hMSC nodes to bMSCs.
      output: $expected$ – a set containing the expected traces in $H$
2     $expected = \{s_0\}$;
3     $visited = \{s_0\}$;
4     $next = \{x | \exists(s_0, x) \in E,\ x \notin visited\}$;
5     while $next \neq \emptyset$ do
6         $current = next.pop(0)$;
7         $next.append(\{x | \exists(current, x) \in E,\ x \notin visited\})$;
8         $visited.append(current)$;
9         foreach $simple\_path\ sp = sc_0, sc_1, ..., sc_n \mid \forall\ 0 \leq i < n\ \exists(sc_i, sc_i + 1) \in E,$
           $\forall\ 0 \leq i < n\ \forall\ 0 \leq j < n\ sc_i = sc_j \leftrightarrow i = j,\ sc_0 = s_0,\ sc_n = current$ do
10            foreach $sp_j \in expected \mid \exists w = sc_k, sc_{k+1}, ...\ and\ sp_j.w = sp$ do
11                $expected.remove(sp_j)$;
12            end
13            if $\nexists\ sp_j \in expected,\ w = sc_k, sc_{k+1}, ... \mid sp.w = sp_j$ then
14                $expected.append(sp)$
15            end
16        end
17    end
18    $looped\_behaviors = \emptyset$;
19    foreach $(s_i, s_j) \mid \exists(s_i, s_j) \in E,\ \exists(s_j, s_i) \in E,\ s_i \neq s_j$ do
20        foreach $sp = sc_0, sc_1, ..., sc_n \in E \mid \exists\ 0 \leq k < n\ s_k = s_i$ do
21            $sp' = sc_0, sc_1, ..., sc_{k-1}, sc_i, sc_j, sc_k, sc_{k+1}, ..., sc_n$;
22            $looped\_behaviors.append(sp')$;
23        end
24    end
25    $expected.append(looped\_behaviors)$;
26    return $expected$
```

4.3 Example

To illustrate our approach, let us take the Boiler System as guiding example. The Boiler System model has been previously introduced, in Chapter 2. Hence, step 1 is skipped, as the model is already known.

4.3.1 Collecting Implied Scenarios

The LTSA-MSC tool is used to collect implied scenarios. Figure 4.2 shows the Boiler System opened in the tool, and the highlighted button on the upper right, opens the dialog to start collecting ISs, which is shown in Figure 4.3a. This window asks the user to input how many ISs he/she wishes to collect. In this example, the number of ISs to collect was set to 10.

After the 10 ISs have been collected (or the tool failed to collect more ISs), a window tells the user the collection process finished, how many ISs were collected, and how much time was spent. This latter window is shown in Figure 4.3b, which indicates that 10 ISs were indeed collected, and the collection process took 3.651s. The collected ISs are also exported to a text file, such as the one shown in Figure 4.4.

4.3.2 Detecting Common Behaviors

Next, after the IS collection, the common behaviors among the ISs are detected. For instance, take the first IS shown in Figure 4.4: *on,pressure,off,on,query*. It is also shown as MSC in Figure 4.5a. When Algorithm 1 is applied to this single IS, the two dashed *on* messages in Figure 4.5a will be detected as a loop. Thus, the messages that are between this repetition will be removed. After this removal, the detected CB for this IS is shown in Figure 4.5b.

This common behavior shows that the cause of the analyzed implied scenario is that *Control* is querying the last measured pressure, but *Sensor* has not registered anything since the system started running, which means that *Control* might decide to act based on old information that might not represent the current state of the system anymore, which correctly describes IS from Figure 4.5a.

Finally, after applying Algorithm 1 to all 10 collected ISs, only two common behaviors are detected. The ones shown in Figure 4.5b (CB0) and Figure 4.6 (CB1). Notice that a CB can be an IS, but that is not always true. For instance, the CB0 was not detected as an IS, while CB1 was the second IS collected.

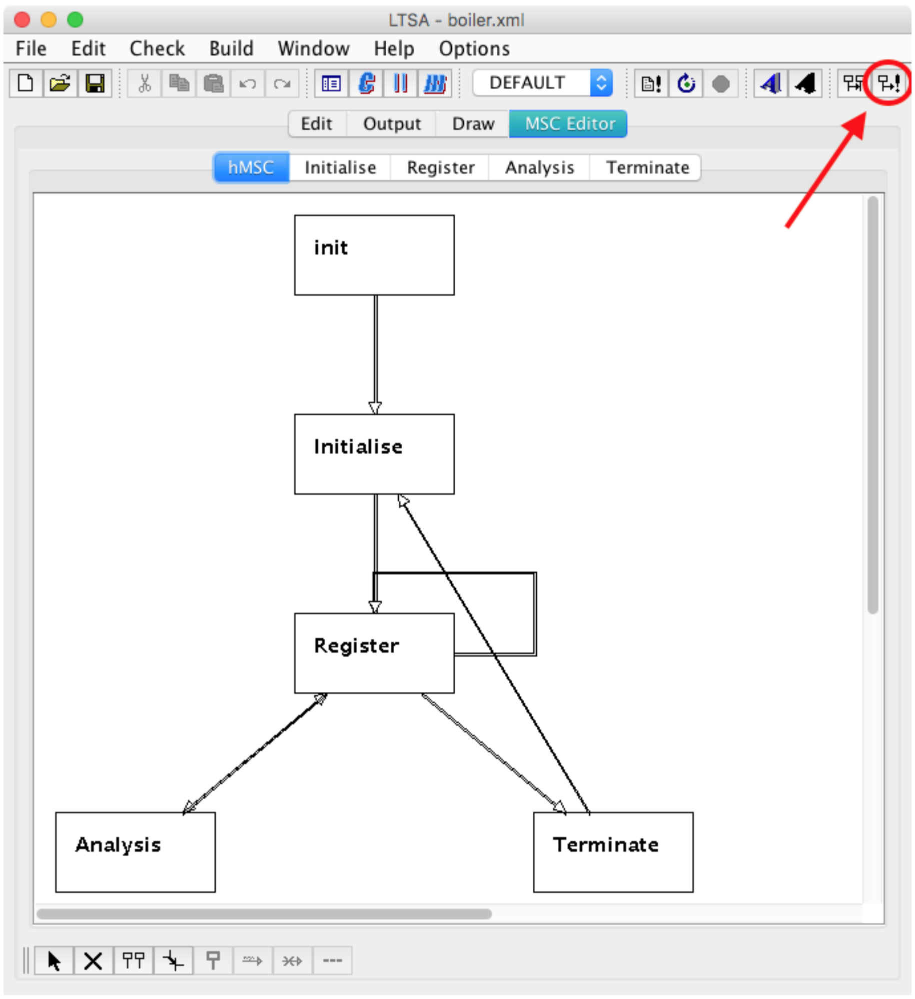
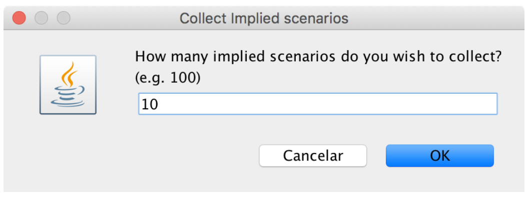
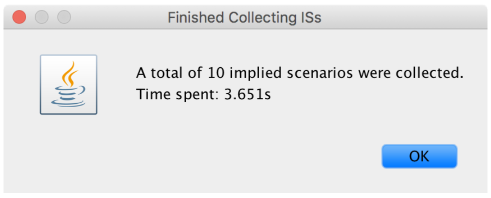

LTSA - boiler.xml
File Edit Check Build Window Help Options
DEFAULT
Edit Output Draw MSC Editor
hMSC Initialise Register Analysis Terminate
init
Initialise
Register
Analysis
Terminate

Figure 4.2: Boiler model opened in the LTSA-MSC tool.



How many implied scenarios do you wish to collect?
(e.g. 100)
10

(a) Start of IS collection.



A total of 10 implied scenarios were collected.
Time spent: 3.651s

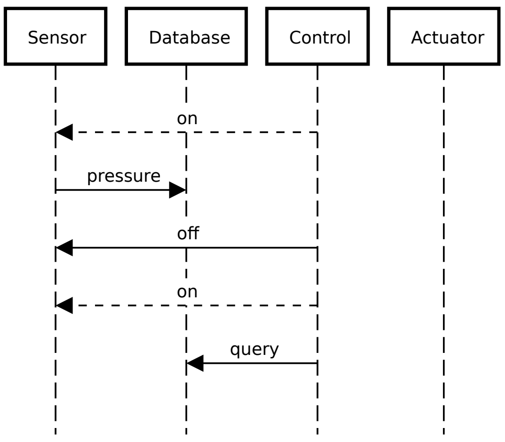
(b) End of IS collection.

Figure 4.3: Dialogs of start and finish of IS collection process.

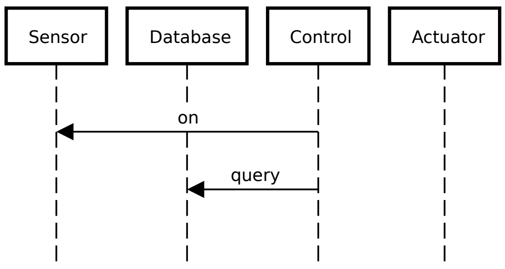
```
ID: [ordered messages]

0: [on, pressure, off, on, query]
1: [on, pressure, query, data, command, off]
2: [on, pressure, pressure, off, on, query]
3: [on, pressure, pressure, query, data, command, off]
4: [on, pressure, pressure, pressure, off, on, query]
5: [on, pressure, off, on, pressure, off, on, query]
6: [on, pressure, pressure, pressure, query, data, command, off]
7: [on, pressure, pressure, pressure, pressure, off, on, query]
8: [on, pressure, query, data, command, pressure, off, on, query]
9: [on, pressure, off, on, pressure, query, data, command, off]
```

Figure 4.4: The collected ISs for the Boiler example.



(a) First Boiler IS.



(b) Detected CB.

Figure 4.5: Example of a common behavior detection.

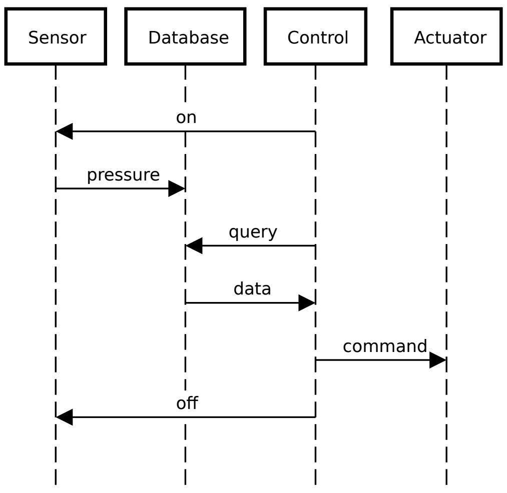
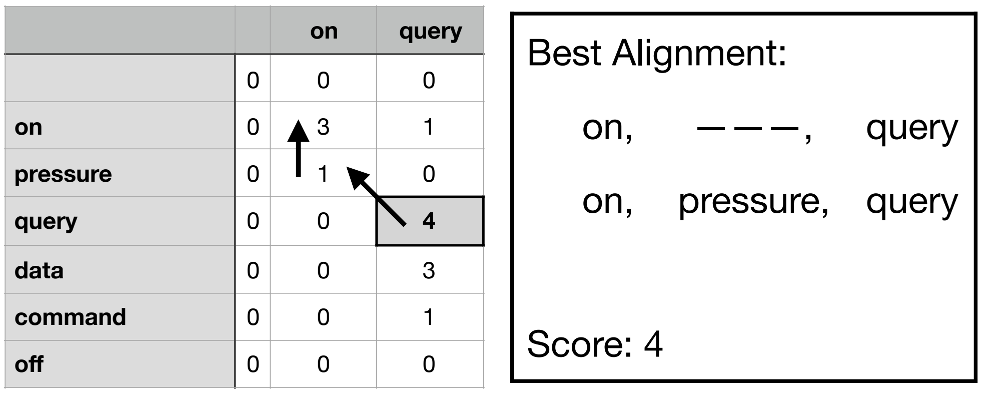


Figure 4.6: Second CB for the Boiler system.



		on	query
	0	0	0
on	0	3	1
pressure	0	1	0
query	0	0	4
data	0	0	3
command	0	0	1
off	0	0	0

(a) sw matrix

Best Alignment:

on, ———, query

on, pressure, query

Score: 4

(b) alignment and score

Figure 4.7: Smith-Waterman applied to Boiler's common behaviors.

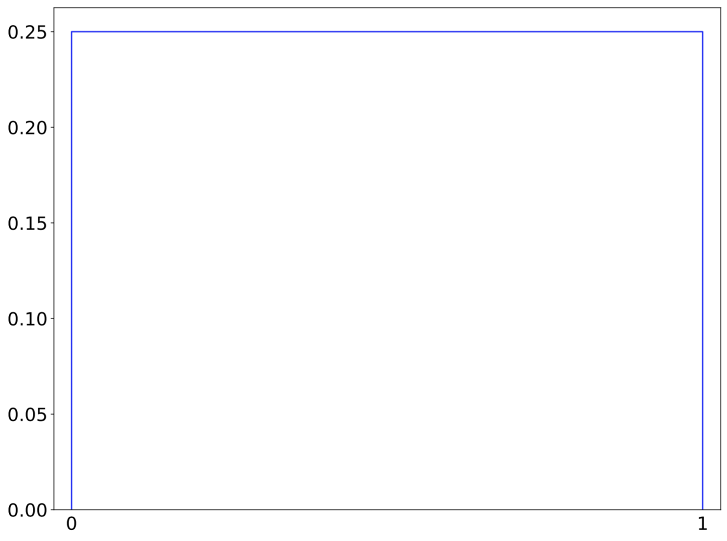


Figure 4.8: Dendrogram for the Boiler example.

4.3.3 Classifying Families of Common Behaviors

Because only two CBs were detected, it is only needed to apply the Smith-Waterman algorithm to this pair of common behaviors. The results are shown in Figure 4.7. In Figure 4.7a the result matrix of the Smith-Waterman algorithm is shown. By doing the traceback from the highest score, the alignment presented in Figure 4.7b is obtained. Finally, using Equation (4.1), the dissimilarity between the CBs is 0.25. Hence, the dendrogram shown in Figure 4.8 is obtained.

By analyzing the best alignment found for the pair of common behaviors, it is possible to see that they do not happen because of the same problem. For the first one (CB0), implied scenarios that share this behavior happen because of a *query* message before a new pressure is registered in the current run of the system, and thus the system might adjust to an outdated pressure.

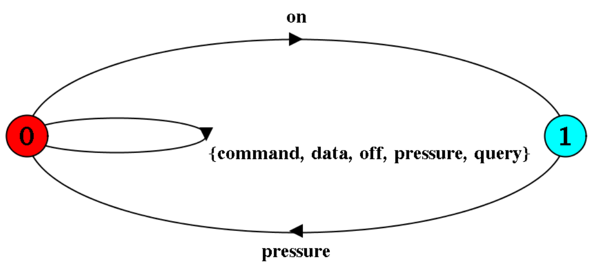


Figure 4.9: LTS of the constraint used to treat Boiler's first common behavior.

The result of the alignment for the second common behavior (CB1) however, has the *pressure* message before a query is carried out, thus, this erratic behavior would not be observed. Therefore, because these common behaviors do not happen because of the same problem, they are not in the same family, and thus, for the Boiler system, two families of common behaviors are defined, each with a single common behavior.

4.3.4 Treating a Family of CBs

As an example, we will focus on the family which contains CB0. Thus, a constraint was created to treat this family, and is shown in Figure 4.9. The LTS visually shows what this constraint guarantees. Its starting state is state 0, and this will be composed with the starting state of the LTS specification. It stays in state 0 for all messages in the specification alphabet but '*on*', that is, as long as the message being sent is not '*on*', this constraint will not interfere with the system execution. In other words, all messages other than '*on*' are ignored. However, whenever an '*on*' message is sent, the transition to state 1 occurs. From state 1, the only accepted transition is going back to state 0, via message '*pressure*'. This means that when an '*on*' message is sent, the following one has to be '*pressure*'. Otherwise, this constraint will not accept it.

This clearly resolves the issue of CB0, because '*pressure*' no longer appears after any '*on*'. Now that this common behavior has been treated, take a look at Figure 4.5. Consider that both sequences are a collected IS, and obviously would have the same common behavior, as they are simply partial steps of the detection process. This constraint prevents those two scenarios to happen, as they both have the same problem, a query of outdated information. That is, just as our hypothesis suggested, one treatment was able to resolve multiple ISs that share the same CB.

In addition, to make sure that the ISs from CB0 have been removed, a constrained architecture model is built by composing the constraint created with the original model. The constrained architecture model is shown in Figure 4.10. The only transition labeled with the message *on* is from state 0 to state 1, and the only transition outgoing from state 1 is to state 2 with *pressure*. Therefore, in the constrained architecture model the

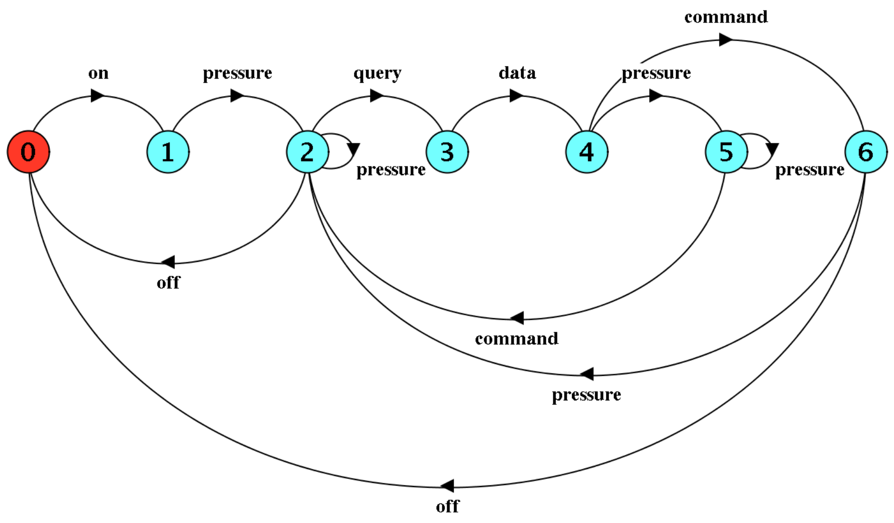


Figure 4.10: LTS of the Boiler constrained architecture model.

behavior *on, query* does not happen, as there are no three states s_0, s_1, s_2 that satisfies the transitions (s_0, on, s_1) and $(s_1, query, s_2)$ in the constrained model.

Finally, Algorithm 2 is used to verify that the collected ISs have been removed in the constrained architecture model. Figure 4.11 shows that all collected ISs are indeed reachable in the original architectural model, while Figure 4.12 shows which ISs are reachable in the constrained architecture model[2]. Furthermore, the two expected behaviors of the system generated with Algorithm 3 are preserved in the constrained model. These behaviors are:

- init, Initialise, Register, Terminate
- init, Initialise, Register, Analysis, Register, Terminate

which are then expanded with the messages of each scenario. Therefore, the traces reached by Algorithm 2 are:

- on,pressure,off
- on,pressure,query,data,command,pressure,off

As shown in the previous analysis of the LTS, all the ISs of CB0 have been removed. Therefore, the constraint introduced has resolved CB0. However, CB1 can still happen in the constrained model, which indicates that another constraint needs to be introduced to the system. This second constraint will be further detailed in Section 5.2.2.

[2]In the following uses of this approach, only the 'RESULTS SUMMARY' part will be shown, as it summarizes the relevant information regarding CBs

```
### Expected Behaviors

    0:on,pressure,off: reached
    1:on,pressure,query,data,command,pressure,off: reached

----------

### Common Behavior 0

    0:on,pressure,off,on,query: reached
    2:on,pressure,pressure,off,on,query: reached
    4:on,pressure,pressure,pressure,off,on,query: reached
    5:on,pressure,off,on,pressure,off,on,query: reached
    7:on,pressure,pressure,pressure,pressure,off,on,query: reached
    8:on,pressure,query,data,command,pressure,off,on,query: reached

----------

### Common Behavior 1

    1:on,pressure,query,data,command,off: reached
    3:on,pressure,pressure,query,data,command,off: reached
    6:on,pressure,pressure,pressure,query,data,command,off: reached
    9:on,pressure,off,on,pressure,query,data,command,off: reached

----------
----------

RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 6 out of 6
Common Behavior 1 -> traces reached: 4 out of 4

Total traces reached: 12 out of 12
```

Figure 4.11: Traces reached in the Boiler original model.

```
### Expected Behaviors

    0:on,pressure,off: reached
    1:on,pressure,query,data,command,pressure,off: reached

----------

### Common Behavior 0

    0:on,pressure,off,on,query: NOT reached
    2:on,pressure,pressure,off,on,query: NOT reached
    4:on,pressure,pressure,pressure,off,on,query: NOT reached
    5:on,pressure,off,on,pressure,off,on,query: NOT reached
    7:on,pressure,pressure,pressure,pressure,off,on,query: NOT reached
    8:on,pressure,query,data,command,pressure,off,on,query: NOT reached

----------

### Common Behavior 1

    1:on,pressure,query,data,command,off: reached
    3:on,pressure,pressure,query,data,command,off: reached
    6:on,pressure,pressure,pressure,query,data,command,off: reached
    9:on,pressure,off,on,pressure,query,data,command,off: reached

----------
----------

RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 0 out of 6
Common Behavior 1 -> traces reached: 4 out of 4

Total traces reached: 6 out of 12
```

Figure 4.12: Traces reached in the Boiler constrained model.

Chapter 5

Evaluation

In this chapter, a total of seven case studies are tested to validate the proposed methodology. These case studies were selected because they were already explored in the literature, and provide a range of complexity for the specifications. The most simple system specifications are the *A Passenger Transportation System* [6] and *Semantic Search Multi-Agent System* [16], where each system has only two scenarios and no loops, while the most complex ones are the *eB2B* and *Global System for Mobile Mobility Management System*, where each system has over 10 unique scenarios and multiple loops. The other three systems – (i) *Boiler System* [6], (ii) *Cruise Control System* [19], and (iii) *Distributed Smart Camera System* [2] – are a middle point between the two extremes, where (i) and (ii) contains four scenarios each and various loops, while (iii) contains five scenarios but no loops.

The rest of this chapter is structured as follows: firstly, Section 5.1 details the specifications of the equipment used for the tests; secondly, Section 5.2 shows a detailed analysis of the seven different system specifications analyzed; finally, Section 5.3 discusses the obtained results across all studied systems.

5.1 Setup

All experiments were executed in the same machine, running macOS 10.12.6, 16 GB of memory, and a 2.7 GHz Intel Core i7 processor. Additionally, Java heap space was set to 4GB for the LTSA-MSC tool to run.[1] For each system, the same analysis was repeated with up to 25, 50, 75, 100, 125, 150, and up to 500 collected implied scenarios. However, for some systems, it was not possible to repeat the analysis with a varying number of ISs collected. Thus, the collection process was repeated ten times with the same number of ISs. Finally, without loss of generality, all collected implied scenarios were considered to

[1]All files needed to replicate these can be found at https://github.com/cbdm/Implied_Scenarios.

be negative and thus were resolved with constraints. That is, we created constraints to avoid the faults of the observed behaviors. This is only to show that it is possible to treat multiple ISs at once.

5.2 Case Studies

5.2.1 Case Study 1: A Passenger Transportation System

System Description

A Passenger Transportation System (APTS), introduced by Uchitel et al. [6], consists of high-speed vehicles that transport one person at a time and Passengers can only embark at terminals where they select the destination terminal [6]. The system contains two scenarios represented by bMSCs (shown in Appendix A.1), and an hMSC that connects the two, which is depicted in Figure 5.1. The *VehicleAtTerminal* scenario describes what happens when the passenger requests for the vehicle while both are at the same terminal, and the *VehicleNotAtTerminal* scenario when the passenger is at one terminal while the vehicle is at a different one. Finally, the components of this system are two passengers, two terminals, one vehicle and a control center.

Analysis

Initially, by using the LTSA-MSC tool, a total of 25 implied scenarios were to be collected. However, the tool was only able to collect 9 ISs in the specification. That is, even though a larger collection of ISs was desired, the tool was only able to detect 9 ISs in the system specification, which prevented repeated analysis with other numbers of ISs. Therefore, the collection process was repeated to try to collect 25 ISs nine more times. However, the

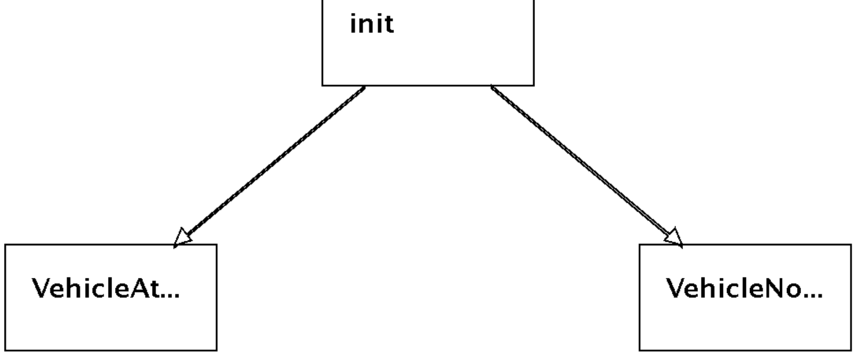


Figure 5.1: hMSC for APTS specification.

```
ID: [ordered messages]

0: [passenger2.terminal2.buyTicket, passenger1.terminal1.buyTicket]
1: [passenger1.terminal1.buyTicket, passenger2.terminal2.buyTicket]
2: [passenger2.terminal2.buyTicket, terminal2.passenger2.displayGate, terminal2.vehicle.setDestination]
3: [passenger2.terminal2.buyTicket, terminal2.passenger2.displayGate, passenger1.terminal1.buyTicket]
4: [passenger1.terminal1.buyTicket, terminal1.controlcentre.requestVehicle,
passenger2.terminal2.buyTicket]
5: [passenger2.terminal2.buyTicket, terminal2.passenger2.displayGate, passenger2.vehicle.enter,
passenger1.terminal1.buyTicket]
6: [passenger2.terminal2.buyTicket, terminal2.passenger2.displayGate, passenger2.vehicle.enter,
terminal2.vehicle.setDestination, passenger1.terminal1.buyTicket]
7: [passenger2.terminal2.buyTicket, terminal2.passenger2.displayGate, passenger2.vehicle.enter,
terminal2.vehicle.setDestination, vehicle.terminal2.departReq, passenger1.terminal1.buyTicket]
8: [passenger2.terminal2.buyTicket, terminal2.passenger2.displayGate, passenger2.vehicle.enter,
terminal2.vehicle.setDestination, vehicle.terminal2.departReq, terminal2.vehicle.departAck,
passenger1.terminal1.buyTicket]
```

Figure 5.2: ISs collected for APTS.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 1 out of 1
Common Behavior 1 -> traces reached: 1 out of 1
Common Behavior 2 -> traces reached: 1 out of 1
Common Behavior 3 -> traces reached: 1 out of 1
Common Behavior 4 -> traces reached: 1 out of 1
Common Behavior 5 -> traces reached: 1 out of 1
Common Behavior 6 -> traces reached: 1 out of 1
Common Behavior 7 -> traces reached: 1 out of 1
Common Behavior 8 -> traces reached: 1 out of 1

Total traces reached: 11 out of 11
```

Figure 5.3: Traces reached in APTS original model.

same 9 ISs were collected in every repetition, and they were also detected in the same order.

The collection process took on average 3.895 seconds, with a standard deviation of 0.108s, and the collected ISs are shown in Figure 5.2. Each message follows the structure: *component1.component2.message*, which means that *component1* sent *message* to *component2*. This notation is used to differentiate messages such as *passenger1.terminal1.buyTicket* and *passenger2.terminal2.buyTicket*. Finally, Figure 5.3 shows that all collected ISs can happen in the original model, as well as the expected behaviors listed by Algorithm 3.

After the collection of ISs, the process to detect common behaviors was done. However, each IS had a unique CB, and thus the number of elements was not reduced. Therefore, the CBs detected are equivalent to the ISs shown in Figure 5.2. With the CBs detected, the Smith-Waterman algorithm is applied to all possible pairs, and the dissimilarity is calculated. Next, the dendrogram in Figure 5.4. The process of detecting CBs, applying

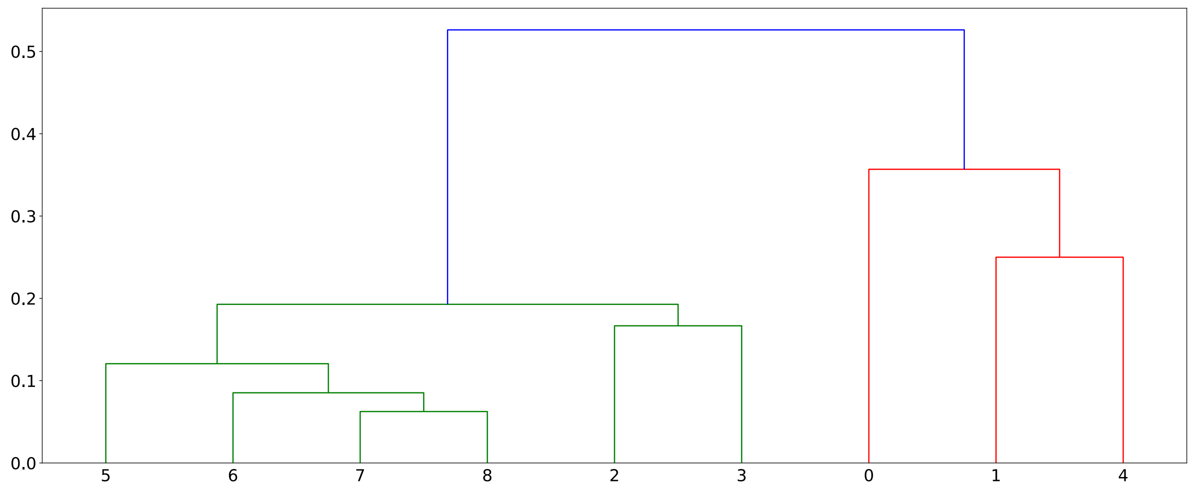


Figure 5.4: Dendrogram for APTS.

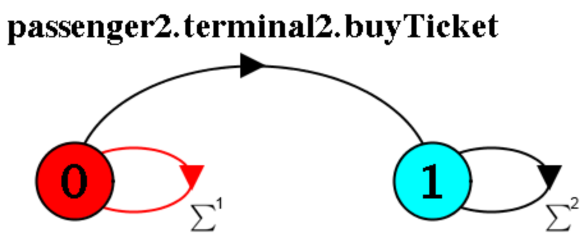


Figure 5.5: First constraint for APTS.

the SW algorithm to the pairs of CBs, clustering the CBs, and exporting the dendrogram took 0.33s.

Next, a manual analysis of the obtained results is required. The analysis starts from the less dissimilar pair (i.e., the lowest hight in the dendrogram), which is the pair of CBs 7 and 8. By looking at the sequence of messages of each CB (ISs 7 and 8 from Figure 5.2), it is noticeable that they share most of their massages. Also, their first message indicates that Passenger 2 bought a ticket, and the last message indicates that Passenger 1 bought a ticket. This behavior is unexpected, as in the system description only one passenger should be transported at a time. Therefore, the constraint in Figure 5.5 is created.

This constraint makes sure that after Passenger 2 buys a ticket, Passenger 1 is not allowed to buy a ticket. To reduce space, $\sum$ is used as the set of all messages in the model. $\sum^1$ and $\sum^2$ are then defined as, respectively, $\sum \setminus \{passenger2.terminal2.buyTicket\}$ and $\sum \setminus \{passenger1.terminal1.buyTicket\}$. This means that in state 0, the LTS allows for all messages to be exchanged, but when *passenger2.terminal2.buyTicket* is observed, the transition to state 1 occurs. In state 1, all the messages are allowed, except for *passenger1.terminal1.buyTicket*. In other words, the constraint allows everything until Passenger 2 buys a ticket, from then on it prohibits Passenger 1 from buying a ticket. Additionally,

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 0 out of 1
Common Behavior 1 -> traces reached: 1 out of 1
Common Behavior 2 -> traces reached: 1 out of 1
Common Behavior 3 -> traces reached: 0 out of 1
Common Behavior 4 -> traces reached: 1 out of 1
Common Behavior 5 -> traces reached: 0 out of 1
Common Behavior 6 -> traces reached: 0 out of 1
Common Behavior 7 -> traces reached: 0 out of 1
Common Behavior 8 -> traces reached: 0 out of 1

Total traces reached: 5 out of 11
```

Figure 5.6: Traces reached in APTS first constrained model.

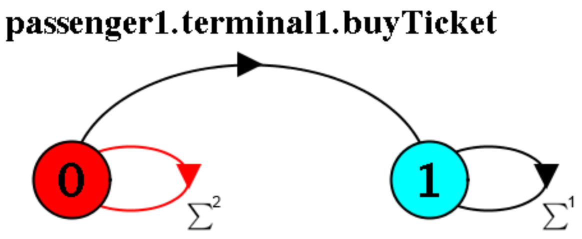


Figure 5.7: Second constraint for APTS.

Figure 5.6 shows that both CB 7 and 8 have been removed after this first constraint restricts the model. Alongside them, CBs 0, 3, 5, and 6 have also been removed, as they had the same cause (i.e., Passenger 1 buying a ticket after Passenger 2), which indicates that CBs 0,3,5,6,7,8 are a part of the same family.

Hence, only CBs 1, 2, and 4 remain untreated. Amongst these three CBs, the most similar pair is of CBs 1 and 4. Thus this is the next behavior analyzed. Through a similar analysis, it is noticeable that these two CBs exhibit a similar behavior to the first family, where both passengers buy a ticket. However, instead of Passenger 2 buying it first, now it is Passenger 1. Therefore, the constraint shown in Figure 5.7 is created.

Analogously to the first constraint, this constraint makes sure that after Passenger 1 buys a ticket, Passenger 2 is not allowed to buy a ticket. Similarly, $\sum^1$ and $\sum^2$ are the same as defined previously. This means that in state 0, the LTS allows for all messages to be exchanged, but when *passenger1.terminal1.buyTicket* is observed, the transition to state 1 occurs. In state 1, all the messages are allowed, except for *passenger2.terminal2.buyTicket*. In other words, the constraint allows everything until Passenger 1 buys a ticket, from then on it prohibits Passenger 2 from buying a ticket. Additionally, Figure 5.8 shows that both

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 0 out of 1
Common Behavior 1 -> traces reached: 0 out of 1
Common Behavior 2 -> traces reached: 1 out of 1
Common Behavior 3 -> traces reached: 0 out of 1
Common Behavior 4 -> traces reached: 0 out of 1
Common Behavior 5 -> traces reached: 0 out of 1
Common Behavior 6 -> traces reached: 0 out of 1
Common Behavior 7 -> traces reached: 0 out of 1
Common Behavior 8 -> traces reached: 0 out of 1

Total traces reached: 3 out of 11
```

Figure 5.8: Traces reached in APTS second constrained model.

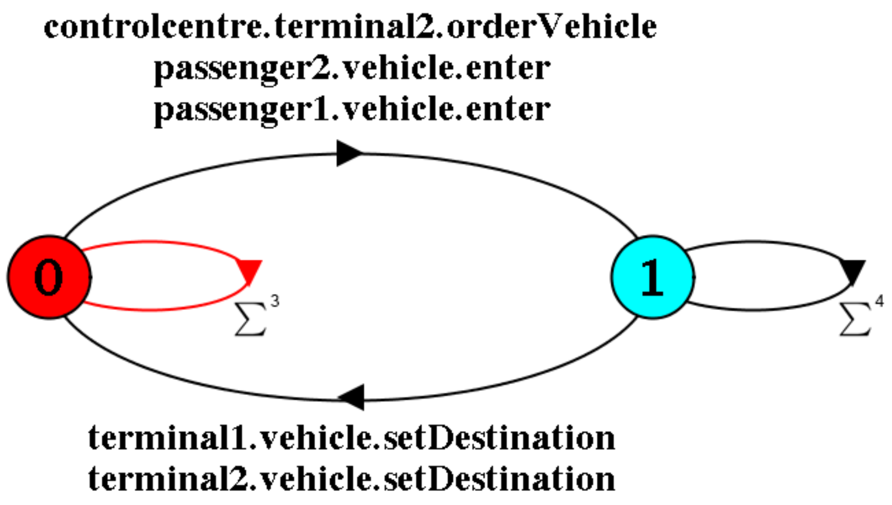

controlcentre.terminal2.orderVehicle
passenger2.vehicle.enter
passenger1.vehicle.enter
0
$\sum^3$
1
$\sum^4$
terminal1.vehicle.setDestination
terminal2.vehicle.setDestination

Figure 5.9: Third constraint for APTS.

CB 1 and 4 have been removed after this second constraint restricts the model, which shows that they are part of the same family.

Thus, only CB 2 is left untreated, whose unexpected behavior is that the destination is set before the passenger enters the vehicle. This is noticed by analyzing the bMSCs of the system, which are shown in Appendix A.1. Therefore, to prohibit this behavior of happening, a third constraint is created, which is shown in Figure 5.9.

This constraint makes sure that either a passenger enters the vehicle or a terminal orders the vehicle before a destination is set. To reduce space, $\sum^3$ and $\sum^4$ were respectively defined as Equation (5.1) and Equation (5.2), where *cc* is *controlcentre*, *p*1 is *passenger1*, *p*2 is *passenger2*, *t*1 is *terminal1*, *t*2 is *terminal2*, and *v* is *vehicle*. In state 0, the constraint allows all messages except for a *setDestination*. However, if either a passenger enter the

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 0 out of 1
Common Behavior 1 -> traces reached: 0 out of 1
Common Behavior 2 -> traces reached: 0 out of 1
Common Behavior 3 -> traces reached: 0 out of 1
Common Behavior 4 -> traces reached: 0 out of 1
Common Behavior 5 -> traces reached: 0 out of 1
Common Behavior 6 -> traces reached: 0 out of 1
Common Behavior 7 -> traces reached: 0 out of 1
Common Behavior 8 -> traces reached: 0 out of 1

Total traces reached: 2 out of 11
```

Figure 5.10: Traces reached in APTS third constrained model.

vehicle or a terminal orders the vehicle, the transition to state 1 occurs. In state 1, the constraint allows all messages, however, if a destination is set by either terminal, the transition to state 0 occurs, which prohibits another destination to be set until a new order or passenger enters the vehicle.

$$\sum^{3} = \sum^{4} \setminus \{cc.t2.orderVehicle, p1.v.enter, p2.v.enter\} \tag{5.1}$$

$$\sum^{4} = \sum \setminus \{t1.v.setDestination, t2.v.setDestination\} \tag{5.2}$$

Finally, Figure 5.10 shows that after the third constraint restricts the model, all detected CBs are removed, consequently, all collected ISs. Therefore, all 9 collected ISs were resolved with only 3 constraints. Furthermore, the expected behaviors were not prevented with the constraints.

Summary

The LTSA-MSC tool was only able to collect 9 ISs in the APTS specification, which makes it impossible to vary the number of ISs collected. Furthermore, the process of detecting CBs did not reduce the number of elements to be analyzed, as 9 CBs were detected for the 9 ISs. However, using the similarity between CBs, 3 families of CBs were defined, and a constraint for each one was created. By these means, all collected ISs were resolved with only three constraints. Thus, supporting the hypothesis that various ISs can be treated together.

Table 5.1: Time spent and # of CBs per ISs for Boiler

number of ISs	25	50	75	100	125	150	500
collection time (h:m:s)	0:00:20.315	0:00:46.597	0:01:39.417	0:03:05.82	0:05:39.565	0:09:18.535	7:35:51.831
number of CBs	2	2	2	2	2	2	2
clustering time (s)	0.318115	0.318115	0.318115	0.318115	0.318115	0.318115	0.318115

5.2.2 Case Study 2: Boiler System

System Description

The Boiler system has been previously presented in Chapter 2. It was introduced by Uchitel et al. [6], and describes a system that controls the temperature inside a boiler to keep the pressure inside thresholds. It is composed of 4 components: *actuator*, *control*, *database*, and *sensor*. It has four scenarios, which were previously shown in Figure 2.1.

Analysis

For the Boiler System, the LTSA-MSC tool was able to collect a variable number of ISs. Therefore, Table 5.1 shows how many ISs were collected in the first line, how long it took to collect the ISs in the second line, and how many CBs were defined with the ISs in the third line. Additionally, the time spent applying the SW algorithm, calculating the dissimilarity, clustering the CBs, and exporting the dendrogram was 0.318115s for all cases, and all the steps are independent of the number of ISs in this case, as they only take the CBs as input, and the number of CBs remained constant.

Although the number of ISs increased, the number of detected CBs remained constant among the collected ISs. Besides, the analysis started in Section 4.3 also detected the same 2 CBs using only 10 ISs. Therefore, it is possible to extend that analysis, which already showed that the two CBs are distinct, and thus there are two families of CBs, each one with a single CB. Therefore, the first constraint for the Boiler is the one created previously, which is shown in Figure 4.9. This constraint prevents that *query* is sent right after *on*, as it makes sure that there is a *pressure* between them, which is the unexpected behavior of the first CB.

Without loss of generality, the analysis will be continued using 500 collected ISs, as that was the most ISs collected, and to show that the methodology can scale to more elements. Additionally, Figure 5.11 shows that in the original model the expected behaviors and all 500 ISs can happen at runtime, while Figure 5.12 shows that after the first constraint restricts the model, all 308 ISs that share the first CB are removed. Thus, the constraint is correctly defined.

Therefore, only the second CB needs to be treated. The second CB consists of *on,pressure,query,data,command,off*. Through an analysis of this sequence of messages,

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 308 out of 308
Common Behavior 1 -> traces reached: 192 out of 192

Total traces reached: 502 out of 502
```

Figure 5.11: Traces reached in Boiler original model.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 0 out of 308
Common Behavior 1 -> traces reached: 192 out of 192

Total traces reached: 194 out of 502
```

Figure 5.12: Traces reached in Boiler first constrained model.

and the system scenarios, it is noticed that the unexpected behavior is for the system to turn off right after the *actuator* is told to control the temperature by the message *command*. Hence, a constraint is created to prevent that the message *off* is sent right after *command*, as in the original model at least one message *pressure* should happen between them. This constraint is shown in Figure 5.13.

The second constraint starts in state 0, where only *off, on, pressure,* and *query* are allowed, that is, all messages except *data* and *command*. This happens because *data* and *command* are only exchanged in the *Analysis* scenario, and that scenario starts with *query*. Therefore, they are prevented unless the scenario starts, which is perceived by observing the message *query*, which makes the transition to state 1 occurs.

From state 1 onwards, only the sequence of messages that are in the *Analysis* scenario are allowed, as the message *query* indicated that this scenario is executing. Therefore, the transitions until state 3 follow the order of messages of the scenario (i.e., *query, data, command*). Thus, state 3 represents the state when the system has finished executing the *Analysis* scenario. According to the Boiler hMSC, the following scenario should be *Register*, which contains only one message – *pressure* – therefore the message *pressure* takes the LTS back to the initial state – state 0.

Finally, to make sure that the second constraint has prevented the ISs that share the second CB, Figure 5.14 shows that all 192 ISs of the CB have been avoided in runtime after the second constraint restricts the model. Thus the constraint has been created correctly.

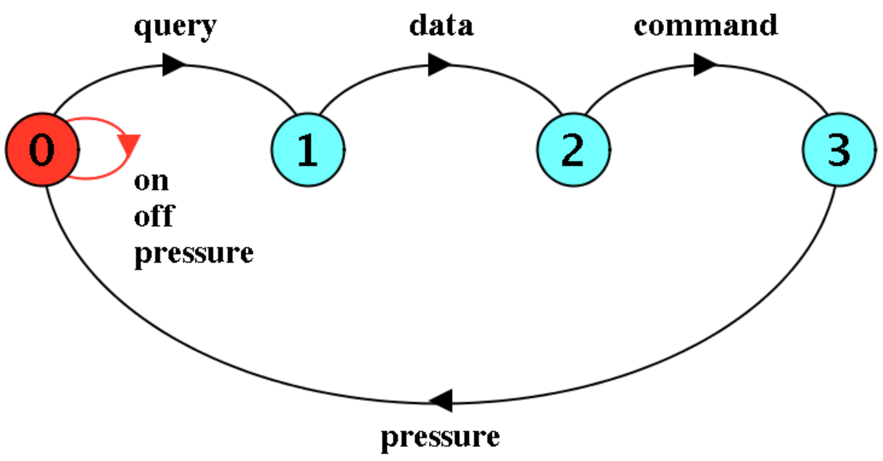


Figure 5.13: Second constraint for Boiler.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 0 out of 308
Common Behavior 1 -> traces reached: 0 out of 192

Total traces reached: 2 out of 502
```

Figure 5.14: Traces reached in Boiler second constrained model.

Summary

The LTSA-MSC tool was able to collect various numbers of ISs in the Boiler specification, which allowed the analysis of Table 5.1. It shows that even though more ISs were collected, the same CBs were detected. Because the manual analysis uses the CBs, detecting more ISs does not help in the analysis if the extra ISs do not have different CBs. Therefore, the 10 ISs collected in Section 4.3 would suffice for the analysis of this system. Even more so, the constraints created based on the CBs of those 10 ISs were able to remove all 500 ISs collected later, which shows that the detected CBs successfully describe the causes of unexpected behaviors in the system.

5.2.3 Case Study 3: Cruise Control System

System Description

The Cruise Control System (Cruiser) was introduced by Magee and Kramer [19], and is a system that can maintain a car in a constant speed. It allows the user to restart the system (*Scen1*), clear the speed (*Scen2*), decrease the set speed (*Scen3*), and increase the set speed (*Scen4*). The hMSC of the system is shown in Figure 5.15 and the individual

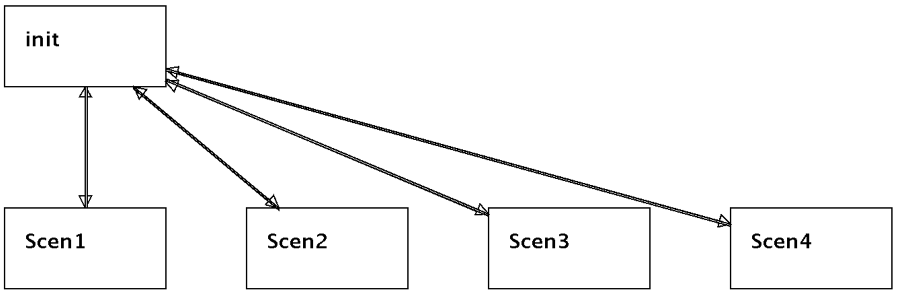


Figure 5.15: hMSC for Cruiser specification.

Table 5.2: Time spent and # of CBs per ISs for Cruiser

number of ISs	25	50	75	100	125	150	500
collection time (h:m:s)	0:00:26.612	0:01:07.325	0:02:38.328	0:05:10.123	0:07:41.147	0:12:19.951	9:26:55.046
number of CBs	18	18	18	18	18	18	18
clustering time (s)	0.497454	0.497454	0.497454	0.497454	0.497454	0.497454	0.497454

bMSCs are shown in Appendix A.3. The Cruiser hMSC allows any of the four scenarios to happen from init, and after the scenario has finished it goes back to init, which allows for any arrangement of scenarios to happen.

Analysis

The LTSA-MSC tool was able to collect a varying number of ISs in the Cruiser specification. Therefore, Table 5.2 shows how many ISs were collected in the first line, how long it took to collect the ISs in the second line, and how many CBs were defined with the ISs in the third line. Additionally, the time spent applying the SW algorithm, calculating the dissimilarity, clustering the CBs, and exporting the dendrogram was 0.497454s for all cases, which is shown in the fourth line. That happens because all these steps are independent of the number of ISs in this case, as they only take the CBs as input, and the number of CBs remained constant.

A total of 18 CBs were detected using the 500 collected ISs, and are shown in Figure 5.16. In addition, Figure 5.17 shows that all collected ISs and expected behaviors are reached in the original model. After using the Smith-Waterman algorithm and clustering the detected CBs, the dendrogram in Figure 5.18 is obtained. Next, the most similar pair of CBs is analyzed, which is the pair CB 12 and CB 13 with a dissimilarity of 0.047.

The CBs 12 and 13 share almost all of their messages, with the only difference being the last one, which tells us that the messages *brake* and *off* could not have been preceded by the previous messages. A further analysis of the scenarios shown in Appendix A.3 confirms such suspicion. The Cruiser bMSCs show that before *break* there should be a message *enableControl*, and before *off* there should be either *enableControl*, (*speed* $\rightarrow$

```
0: ['engineOn', 'clearSpeed', 'engineOff']

1: ['engineOn', 'speed', 'clearSpeed', 'speed']

2: ['engineOn', 'speed', 'clearSpeed', 'on']

3: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'off']

4: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'speed', 'enableControl', 'brake']

5: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'speed', 'enableControl', 'accelerator']

6: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'speed', 'off']

7: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'speed', 'brake']

8: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'brake', 'speed']

9: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'accelerator',
'disableControl']

10: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'speed', 'accelerator']

11: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'speed', 'brake']

12: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'speed', 'setThrottle',
'off']

13: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'speed', 'setThrottle',
'brake']

14: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'brake', 'disableControl',
'speed']

15: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'speed', 'setThrottle', 'brake']

16: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'speed', 'setThrottle', 'accelerator']

17: ['engineOn', 'clearSpeed', 'on', 'recordSpeed', 'enableControl', 'speed', 'accelerator',
'setThrottle', 'disableControl', 'speed']
```

Figure 5.16: CBs detected for Cruiser.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 12 out of 12
Common Behavior 0 -> traces reached: 46 out of 46
Common Behavior 1 -> traces reached: 39 out of 39
Common Behavior 2 -> traces reached: 39 out of 39
Common Behavior 3 -> traces reached: 29 out of 29
Common Behavior 4 -> traces reached: 24 out of 24
Common Behavior 5 -> traces reached: 24 out of 24
Common Behavior 6 -> traces reached: 24 out of 24
Common Behavior 7 -> traces reached: 24 out of 24
Common Behavior 8 -> traces reached: 24 out of 24
Common Behavior 9 -> traces reached: 24 out of 24
Common Behavior 10 -> traces reached: 35 out of 35
Common Behavior 11 -> traces reached: 35 out of 35
Common Behavior 12 -> traces reached: 20 out of 20
Common Behavior 13 -> traces reached: 20 out of 20
Common Behavior 14 -> traces reached: 20 out of 20
Common Behavior 15 -> traces reached: 29 out of 29
Common Behavior 16 -> traces reached: 29 out of 29
Common Behavior 17 -> traces reached: 15 out of 15

Total traces reached: 512 out of 512
```

Figure 5.17: Traces reached in Cruiser original model.

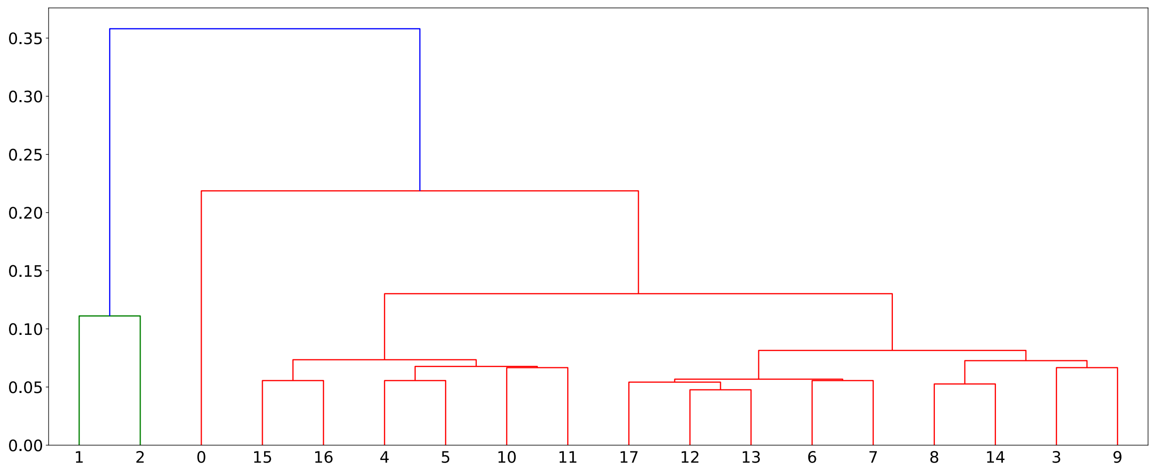


Figure 5.18: Dendrogram for Cruiser.

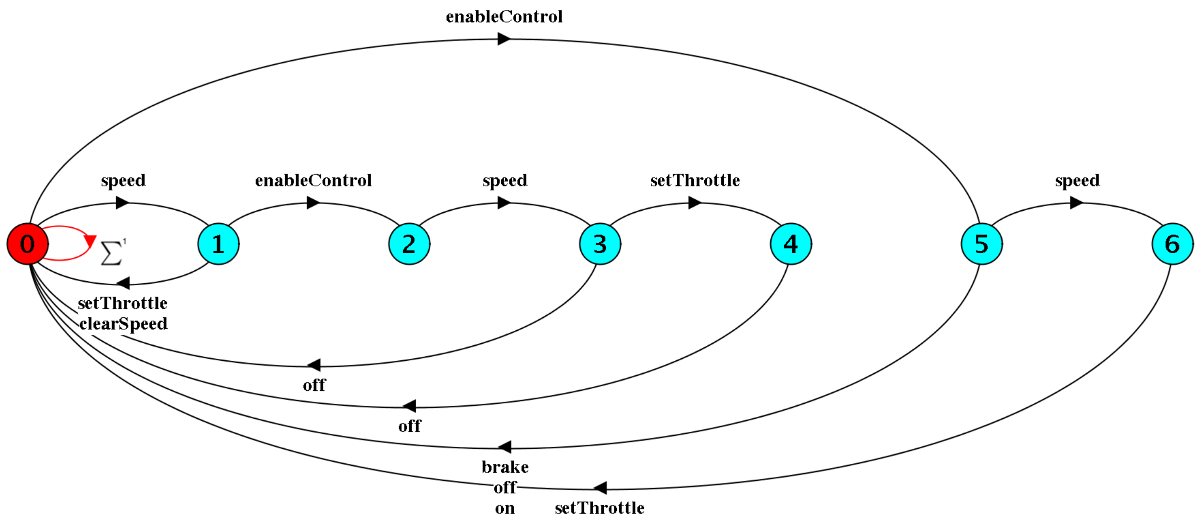


Figure 5.19: First constraint for Cruiser.

enableControl $\rightarrow$ *speed*), or (*speed* $\rightarrow$ *enableControl* $\rightarrow$ *speed* $\rightarrow$ *setThrottle*). Therefore, a constraint to make sure that *brake* and *off* appear only in the appropriate places is created, which is shown in Figure 5.19.

To reduce space, let $\sum^1 = \sum \setminus \{speed, enableControl\}$, where $\sum$ is the set of all messages in the specification. In state 0, the constraint allows all messages to be exchanged, but when either *speed* or *enableControl* are observed, respectively, a transition occurs to either state 1 or 5. State 1 checks if the message following speed is *enableControl*, that is, if the sequence of messages that allow *off* is observed, and thus the transition to state 2 occurs. If it is not, the other possible messages after *speed* according to the specification take the LTS back to state 0. In state 2 it is known that the precondition to *off* happened, thus the sequences (state 2 $\rightarrow$ state 3 $\rightarrow$ state 0) and (state 2 $\rightarrow$ state 3 $\rightarrow$ state 4 $\rightarrow$ state 0) allow the message *off* to happen in an appropriate place. In state 5, the possible continuations after *enableControl* in the specification take the LTS back to state 0. That is, from state 5, *brake*, *off*, or *on* make the transition to state 0 occurs, where *speed* makes the transition to state 6 occurs, where *setThrottle* takes the LTS back to state 0.

Figure 5.20 shows that CBs 4, 5, 6, 7, 9, 10, 11, 12, 13, 15, 16, and 17 have been completely removed. Interestingly, however, is the fact that some ISs from the remaining CBs have also been removed, but not all of them. Through further analysis of these ISs, it is possible to notice that the constraint restricted some behavior of the individual ISs, but not from their CBs.

Take the following IS for instance: *engineOn*, *clearSpeed*, *on*, *recordSpeed*, *enableControl*, *speed*, *accelerator*, *setThrottle*, *disableControl*, *engineOff*, *engineOn*, *clearSpeed*, *engineOff*. After removing the loops of messages, the detected behavior for this IS is CB 0 (i.e.,*engineOn*, *clearSpeed*, *engineOff*). Although the messages of the CB have not been restricted,

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 12 out of 12
Common Behavior 0 -> traces reached: 26 out of 46
Common Behavior 1 -> traces reached: 20 out of 39
Common Behavior 2 -> traces reached: 20 out of 39
Common Behavior 3 -> traces reached: 20 out of 29
Common Behavior 4 -> traces reached: 0 out of 24
Common Behavior 5 -> traces reached: 0 out of 24
Common Behavior 6 -> traces reached: 0 out of 24
Common Behavior 7 -> traces reached: 0 out of 24
Common Behavior 8 -> traces reached: 16 out of 24
Common Behavior 9 -> traces reached: 0 out of 24
Common Behavior 10 -> traces reached: 0 out of 35
Common Behavior 11 -> traces reached: 0 out of 35
Common Behavior 12 -> traces reached: 0 out of 20
Common Behavior 13 -> traces reached: 0 out of 20
Common Behavior 14 -> traces reached: 12 out of 20
Common Behavior 15 -> traces reached: 0 out of 29
Common Behavior 16 -> traces reached: 0 out of 29
Common Behavior 17 -> traces reached: 0 out of 15

Total traces reached: 126 out of 512
```

Figure 5.20: Traces reached in Cruiser first constrained model.

the IS itself had the messages *accelerator* and *setThrottle* out of order, according to *Scen4*. Thus, when the possible messages after *enableControl* were set to {*brake*, *off*, *on*, (*speed*→*setThrottle*)}, the sequence *speed* → *accelerator* → *setThrottle* was avoided after *enableControl*, which removed the IS, even though the detected CB was not affected.

Therefore, the CBs 0, 1, 2, 3, 8, and 14 remain unresolved and are the next to be treated. Out of these CBs, the most similar pair is CB 8 and CB 14, with a dissimilarity of 0.052. Analyzing their messages, it is noticeable that both share the same messages until *brake*. According to *Scen3*, which is the only scenario where *brake* happens, shows that it should be followed by *disableControl* → *engineOff*. Thus, a constraint is created to make force that ordering of message. The constraint is shown in Figure 5.21.

To reduce space, let $\sum^2 = \sum\setminus\{disableControl, enableControl\}$. State 0 allows all messages to be exchanged, while waiting for either *disableControl* or *enableControl* to trigger a transition. If the former is observed, the transition to state 1 occurs. In state 1, the only message after *disableControl* is allowed (i.e., *engineOff*), which takes the LTS back to state 0 and prohibits the cause of CB 14 to happen. If *enableControl* is observed in state 0, the transition to state 2 occurs. This state allows the possible sequences of messages after enable control to happen, which are: *speed*, (*on* → *recordSpeed* → *enableControl* → *off*), and (*brake* → *disableControl* →, *engineOff*). This last sequence

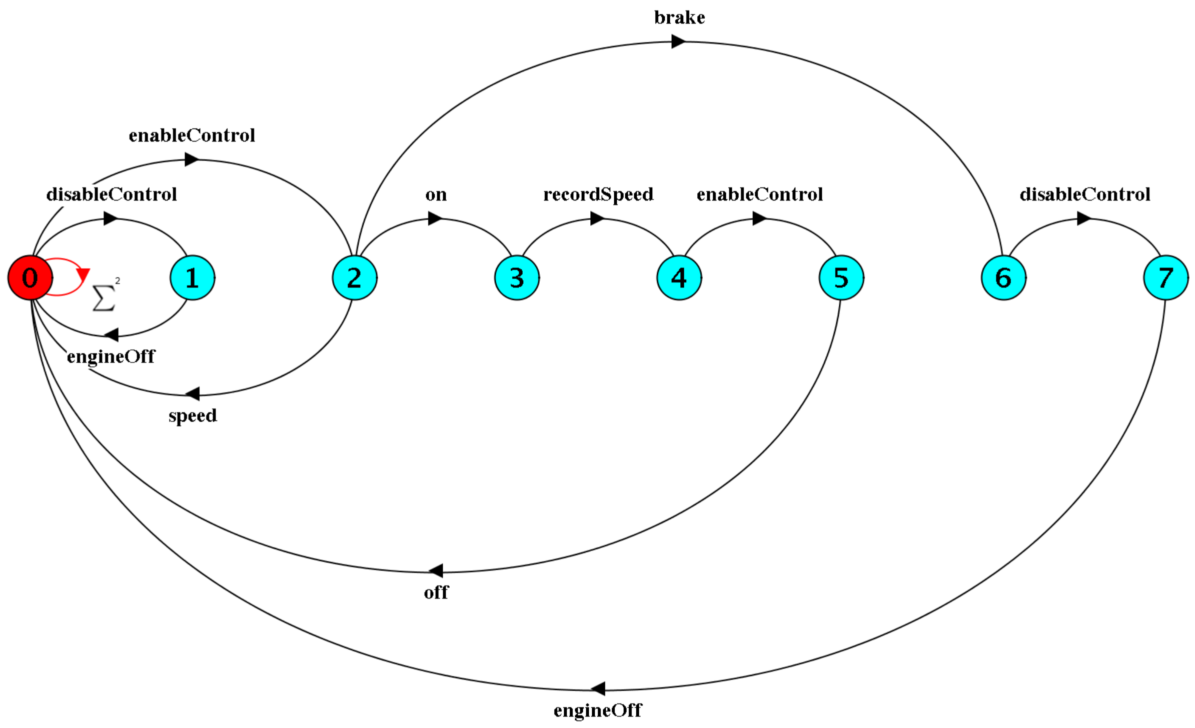


Figure 5.21: Second constraint for Cruiser.

restricts the occurrence of *speed* after *brake*, which was the cause of the CB 8.

Figure 5.22 shows that CBs 3, 8, and 14 have been removed. Thus, only CBs 0, 1, and 2 are unresolved. Through a comparison of all three CBs, it is noticeable that they have fewer messages than the other detected CBs. Because of their length, the messages they share, and the Cruiser scenarios, it is noticeable that they happen because of a wrong order in the start of the scenarios, as all scenarios start with *engineOn*. However, *Scen1* also contains a restart in the middle of the scenario, that is, it contains both *engineOff* and *engineOn* in a sequence. Therefore, a constraint to restrict the possible messages right after *engineOn* is created, considering the 'in-scenario' restart that happens in *Scen1*. The constraint is shown in Figure 5.23.

To reduce space, let $\sum^3 = \sum \setminus \{engineOn, setThrottle\}$, where $\sum$ is the set of all the messages in Cruiser specification. State 0 allows all messages while waiting for *engineOn* or *setThrottle* to trigger a transition. If *setThrottle* is observed, the transition to state 1 occurs. State 1 checks if the following message is *disableControl*, because this sequence ($setThrottle \rightarrow disableControl$) is the distinctive sequence that describes the 'in-scenario' restart that happens in *Scen1*. Therefore, if *disableControl* is observed in state 1, the sequence of states ($2 \rightarrow 3 \rightarrow 4 \rightarrow 0$) allow the correct messages to happen after the restart. If any of the other possible messages after *setThrottle* is observed, the transition back to state 0 occurs. Again from state 0, if *engineOn* is observed, the transition to state 7 occurs. State 7 allows either ($clearSpeed \rightarrow on$), which is the start of *Scen1, Scen3,* and

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 12 out of 12
Common Behavior 0 -> traces reached: 26 out of 46
Common Behavior 1 -> traces reached: 20 out of 39
Common Behavior 2 -> traces reached: 20 out of 39
Common Behavior 3 -> traces reached: 0 out of 29
Common Behavior 4 -> traces reached: 0 out of 24
Common Behavior 5 -> traces reached: 0 out of 24
Common Behavior 6 -> traces reached: 0 out of 24
Common Behavior 7 -> traces reached: 0 out of 24
Common Behavior 8 -> traces reached: 0 out of 24
Common Behavior 9 -> traces reached: 0 out of 24
Common Behavior 10 -> traces reached: 0 out of 35
Common Behavior 11 -> traces reached: 0 out of 35
Common Behavior 12 -> traces reached: 0 out of 20
Common Behavior 13 -> traces reached: 0 out of 20
Common Behavior 14 -> traces reached: 0 out of 20
Common Behavior 15 -> traces reached: 0 out of 29
Common Behavior 16 -> traces reached: 0 out of 29
Common Behavior 17 -> traces reached: 0 out of 15

Total traces reached: 78 out of 512
```

Figure 5.22: Traces reached in Cruiser second constrained model.

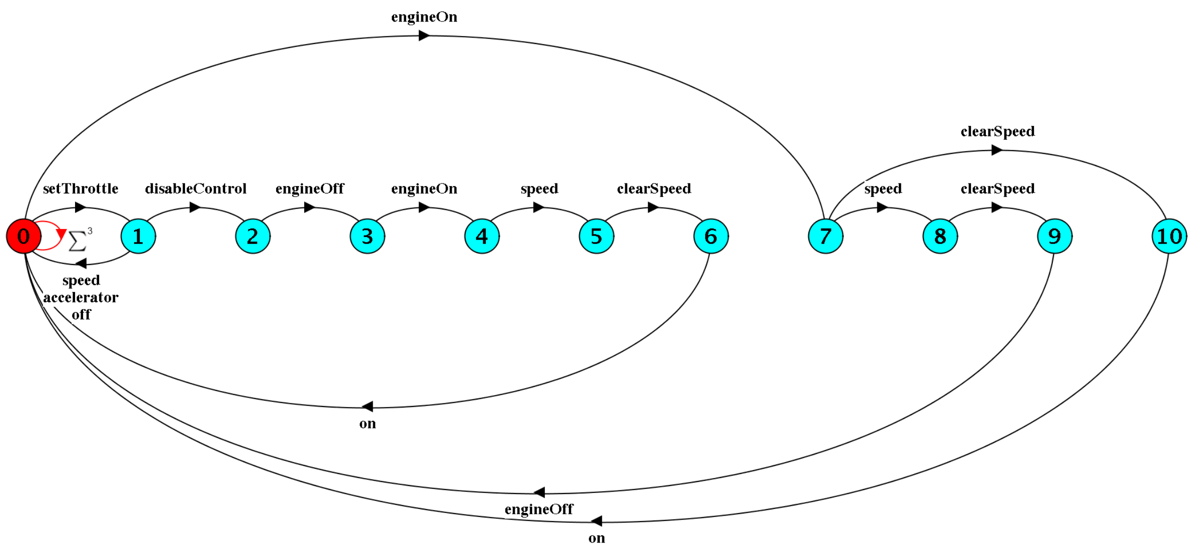


Figure 5.23: Third constraint for Cruiser.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 12 out of 12
Common Behavior 0 -> traces reached: 0 out of 46
Common Behavior 1 -> traces reached: 0 out of 39
Common Behavior 2 -> traces reached: 0 out of 39
Common Behavior 3 -> traces reached: 0 out of 29
Common Behavior 4 -> traces reached: 0 out of 24
Common Behavior 5 -> traces reached: 0 out of 24
Common Behavior 6 -> traces reached: 0 out of 24
Common Behavior 7 -> traces reached: 0 out of 24
Common Behavior 8 -> traces reached: 0 out of 24
Common Behavior 9 -> traces reached: 0 out of 24
Common Behavior 10 -> traces reached: 0 out of 35
Common Behavior 11 -> traces reached: 0 out of 35
Common Behavior 12 -> traces reached: 0 out of 20
Common Behavior 13 -> traces reached: 0 out of 20
Common Behavior 14 -> traces reached: 0 out of 20
Common Behavior 15 -> traces reached: 0 out of 29
Common Behavior 16 -> traces reached: 0 out of 29
Common Behavior 17 -> traces reached: 0 out of 15

Total traces reached: 12 out of 512
```

Figure 5.24: Traces reached in Cruiser third constrained model.

Scen4, or (*speed* → *clearSpeed* → *engineOff*), which is the complete *Scen2*. Finally, all CBs are successfully removed after the third constraint restricts the model, as shown in Figure 5.24.

Summary

The LTSA-MSC tool was able to collect various numbers of ISs in the Boiler specification, which allowed the analysis of Table 5.2. It shows that even though more ISs were collected, the same CBs were detected. Because the manual analysis uses the CBs, detecting more ISs does not help in the analysis if the extra ISs do not have different CBs. Furthermore, the SW algorithm and clustering made possible for the 18 detected CBs to be characterized in 3 families, which allowed the treatment of all collected ISs with only 3 constraints. Thus supporting the hypothesis that multiple ISs can be treated together.

However, unlike the previous case studies, the created constraints did not restrict general behaviors (such as querying the database after turning the system on, in Section 5.2.2), but instead restricted the order that the messages could appear to fit the modeled scenarios. Although it is possible that an architectural refinement would be better suited to remove the collected ISs, this analysis is not in the scope of this dissertation.

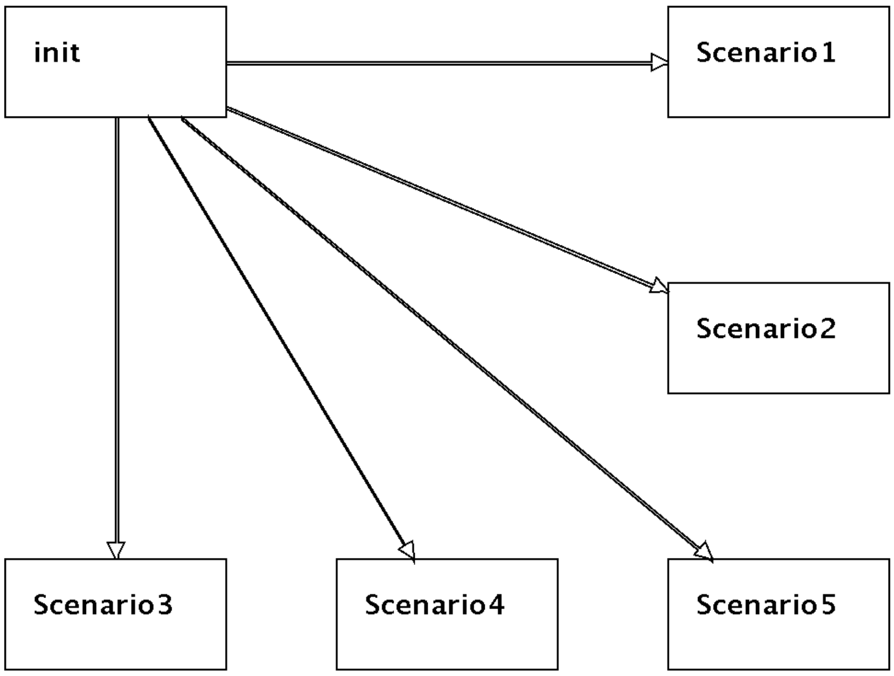


Figure 5.25: hMSC for SmartCam specification.

5.2.4 Case Study 4: Distributed Smart Camera System

System Description

The Distributed Smart Camera System (SmartCam) was first introduced by Esterle et al. [38], and describes the functionality of a system where multiple cameras interact so that an object is tracked by only one camera at a time while making sure that the camera tracking is the best one suited for the task. The cameras are arranged in two different manners, which are called *Architectures*. The model was later detailed by Al-Azzani [2], using four cameras and a single object, which are the five components of the system. The SmartCam model used for analysis is the Architecture 2 laid out by Al-Azzani [2], which is shown in Figure 5.25. The model consists of five scenarios, where each one describes the interactions between the cameras for a different trajectory of the object. The individual bMSCs and the trajectories they depict are shown in Appendix A.4, and were first presented in [34].

Analysis

A total of 9 ISs were collected in the SmartCam specification using the LTSA-MSC tool. That is, the tool was unable to collect more than the 9 ISs, which makes it impossible to vary the number of collected ISs. Therefore, the collection process was repeated to try to collect 25 ISs nine more times. However, unlike in the APTS case study, the ISs were not

```
ID: [ordered messages]

0: [camera2.object.sees_obj, camera4.object.sees_obj, camera2.camera1.ask_confidence, camera2.camera4.ask_confidence,
camera1.camera2.no_confidence]
1: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera3.object.loses_obj, camera1.object.sees_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera1.camera3.start_search]

2: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera1.object.sees_obj, camera3.object.loses_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera2.object.sees_obj]

3: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera1.object.sees_obj, camera3.object.loses_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera1.camera2.start_search]

4: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera3.object.loses_obj, camera1.object.sees_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera4.object.sees_obj, camera4.camera2.ask_confidence]

5: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera3.object.loses_obj, camera1.object.sees_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera4.object.sees_obj, camera1.camera3.start_search]

6: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera3.object.loses_obj, camera1.object.sees_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera4.object.sees_obj, camera2.object.sees_obj]

7: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera1.object.sees_obj, camera3.object.loses_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera1.camera3.start_search, camera2.object.sees_obj]

8: [camera3.object.sees_obj, camera3.camera1.ask_confidence, camera1.camera3.no_confidence, camera3.object.start_tracking,
camera3.object.loses_obj, camera1.object.sees_obj, camera3.camera1.start_search, camera1.camera3.found_obj,
camera3.camera1.ask_confidence, camera1.camera3.send_confidence, camera3.camera1.allow_tracking, camera1.object.start_tracking,
camera1.object.loses_obj, camera4.object.sees_obj, camera1.camera2.start_search, camera1.camera3.start_search,
camera2.object.sees_obj, camera2.camera1.found_obj, camera1.camera2.ask_confidence]
```

Figure 5.26: ISs collected for SmartCam.

always detected in the same order. In five of the ten tests, the ISs were detected in the same order as the one shown in Figure 5.26, while in the other five tests, the only change was that IS 3 was detected before IS 2. Nevertheless, the same 9 ISs were collected in all ten tests.

The collection process took on average 04.306 seconds, with a standard deviation of 0.068s. The collected ISs are shown in Figure 5.26, and each message follows the structure: *component1.component2.message*, which means that *component1* sent *message* to *component2*. This notation is used to differentiate messages such as *camera1.object.sees_obj* and *camera2.object.sees_obj*. Finally, Figure 5.27 shows that all collected ISs and expected behaviors can happen in the SmarCam original model.

The common behavior detection process detected 9 CBs, which indicates that each IS has its unique CB. Thus, there was not a reduction of the elements to be analyzed. However, by applying the Smith-Waterman algorithm to the pairs of CBs and clustering the CBs, the dendrogram shown in Figure 5.28 indicates that CB 0 (which contains only IS 0) is very dissimilar to the other elements. Furthermore, it shows that some pairs of CBs are quite similar. This process of detecting CBs, applying the SW algorithm to the pairs, clustering the CBs, and generating the dendrogram took 0.327333 seconds.

Therefore, the analysis starts with the most similar pair of CBs, which is the pair 5 and

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 5 out of 5
Common Behavior 0 -> traces reached: 1 out of 1
Common Behavior 1 -> traces reached: 1 out of 1
Common Behavior 2 -> traces reached: 1 out of 1
Common Behavior 3 -> traces reached: 1 out of 1
Common Behavior 4 -> traces reached: 1 out of 1
Common Behavior 5 -> traces reached: 1 out of 1
Common Behavior 6 -> traces reached: 1 out of 1
Common Behavior 7 -> traces reached: 1 out of 1
Common Behavior 8 -> traces reached: 1 out of 1

Total traces reached: 14 out of 14
```

Figure 5.27: Traces reached in the SmartCam original model.

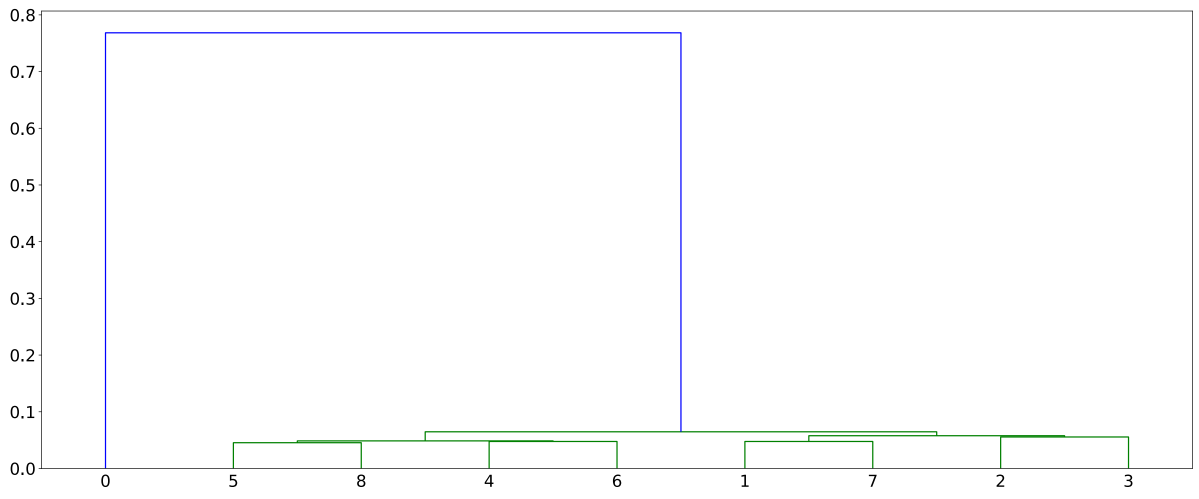


Figure 5.28: Dendrogram for SmartCam.

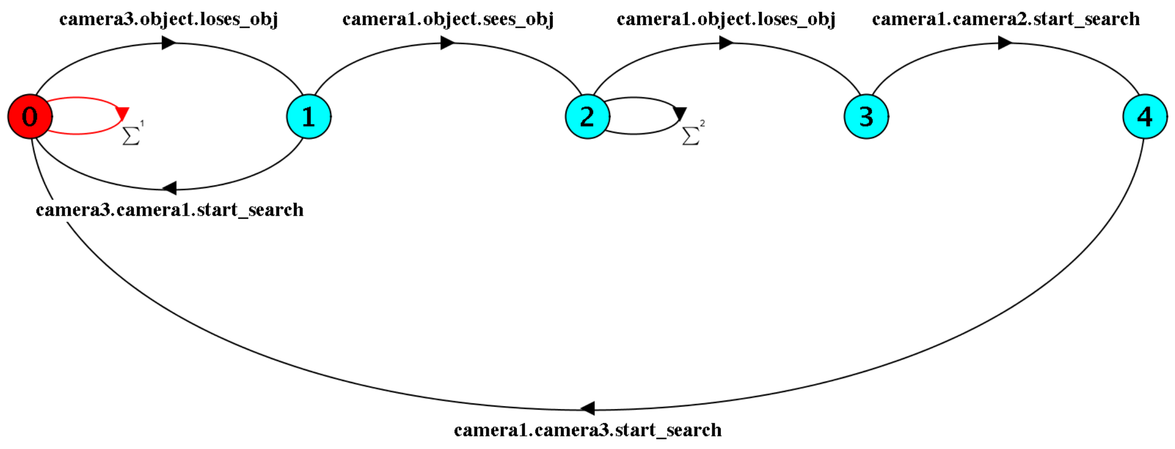

camera3.object.loses_obj
camera1.object.sees_obj
camera1.object.loses_obj
camera1.camera2.start_search
0
1
2
3
4
Σ¹
Σ²
camera3.camera1.start_search
camera1.camera3.start_search

Figure 5.29: First constraint for SmartCam.

8, with a dissimilarity of 0.045. Indeed, through manual analysis of the messages of ISs 5 and 8, it is possible to see that many messages are shared. In fact, when their sequences of messages are compared to the bMSCs, it is clear that they are following *Scenario5*. However, after *camera1.object.loses_obj*, *camera1* should ask *camera2* and *camera3* to start searching, strictly in that order according to *Scenario5*. Nonetheless, *camera1* asks the cameras in the wrong order. This happens because in *Scenario2* the order is inverted, and *camera1* does not have enough information to choose the right order.

This error could be easily fixed with an architectural refinement, by making *camera1* always send the messages in a standard order. However, as all collected ISs are considered to be negative in this dissertation, a constraint is created to prevent this behavior. This is achieved by analyzing the differences between *Scenario2* and *Scenario5*, which can be used to make the system aware of which order should be followed. The constraint is presented in Figure 5.29.

To reduce space, $\sum^1$ and $\sum^2$ are respectively defined as $\sum \setminus \{camera3.object.loses\_obj\}$ and $\sum \setminus \{camera1.object.loses\_obj\}$, where $\sum$ is the set of all messages of the specification. In state 0, the constraint allows all messages to be exchanged, however when *camera3.object.loses_obj* is observed the transition to state 1 occurs. State 1, checks which message follows *camera3.object.loses_obj*, as this message is different among *Scenario2* and *Scenario5*. If the message exchanged is *camera1.object.sees_obj*, then it is *Scenario5* and the transition to state 2 occurs. If it is not, then the transition to state 0 occurs. Similarly to state 0, state 2 allows all messages to happen, but observes for the message *camera1.object.loses_obj*. When this message is observed, the transition to state 3 occurs. States 3 and 4 merely force the order of messages to be *camera1.camera2.start_search*, *camera1.camera3.start_search*.

After this constraint restricts the original model, CBs 1, 4, 5, 6, and 8 are removed, as Figure 5.30 shows, which indicates that these five CBs had the same cause, which was that

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 5 out of 5
Common Behavior 0 -> traces reached: 1 out of 1
Common Behavior 1 -> traces reached: 0 out of 1
Common Behavior 2 -> traces reached: 1 out of 1
Common Behavior 3 -> traces reached: 1 out of 1
Common Behavior 4 -> traces reached: 0 out of 1
Common Behavior 5 -> traces reached: 0 out of 1
Common Behavior 6 -> traces reached: 0 out of 1
Common Behavior 7 -> traces reached: 1 out of 1
Common Behavior 8 -> traces reached: 0 out of 1

Total traces reached: 9 out of 14
```

Figure 5.30: Traces reached in SmartCam first constrained model.

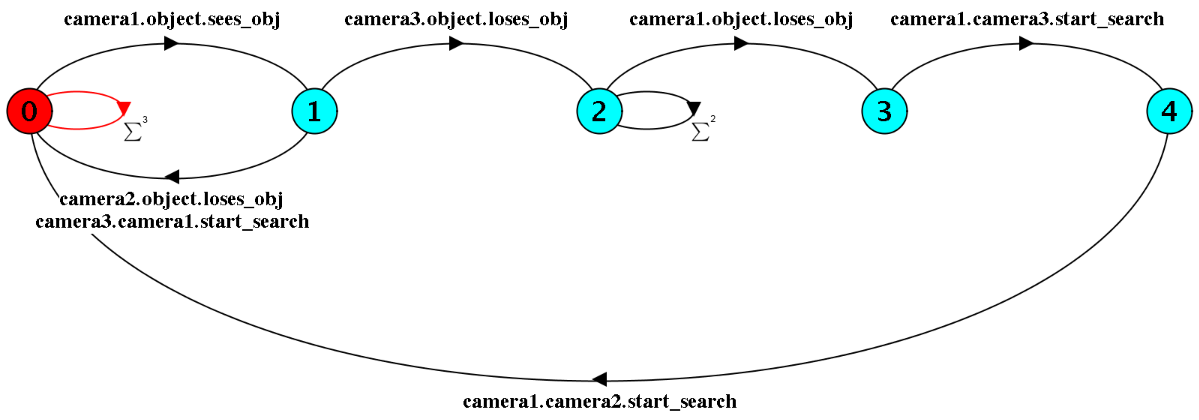


Figure 5.31: Second constraint for SmartCam.

camera1 was following *Scenario2* while the other components were following *Scenario5*. Therefore, only CBs 0, 2, 3, and 7 are left to be resolved.

Among the unresolved CBs, 1 and 7 are the most similar, with a dissimilarity of 0.047, and thus will be analyzed next. Through a manual analysis of the CBs 1 and 7, it is noticed that *camera1* gets the order of messages wrong, similarly to the previous case. However, in this case, *camera1* runs *Scenario5* while the other components run *Scenario2*. Therefore, *camera1* should ask *camera3* and *camera2* to start the search, in that order. Consequently, a second constraint is created, which is shown in Figure 5.31.

Analogously to the first created constraint, the different messages between *Scenario2* and *Scenario5* are used to check which order should be followed. Thus, both constraints are very similar. To reduce space $\sum^3$ is defined as $\sum \setminus \{camera3.object.loses\_obj\}$, where $\sum$ is the set of all messages in the specification, while $\sum^2$ is the same as previously defined for the first constraint. In state 0, the constraint allows all messages to be ex-

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 5 out of 5
Common Behavior 0 -> traces reached: 1 out of 1
Common Behavior 1 -> traces reached: 0 out of 1
Common Behavior 2 -> traces reached: 0 out of 1
Common Behavior 3 -> traces reached: 0 out of 1
Common Behavior 4 -> traces reached: 0 out of 1
Common Behavior 5 -> traces reached: 0 out of 1
Common Behavior 6 -> traces reached: 0 out of 1
Common Behavior 7 -> traces reached: 0 out of 1
Common Behavior 8 -> traces reached: 0 out of 1

Total traces reached: 6 out of 14
```

Figure 5.32: Traces reached in SmartCam second constrained model.

changed, however when *camera1.object.sees_obj* is observed the transition to state 1 occurs. State 1, checks which message follows *camera1.object.sees_obj*, as this message is different among *Scenario2* and the other scenarios. If the message exchanged is *camera3.object.loses_obj*, which indicates it is *Scenario2*, the transition to state 2 occurs. If it is not, then the transition to state 0 occurs. Similarly to state 0, state 2 allows all messages to happen, but observes for the message *camera1.object.loses_obj*, which makes the transition to state 3 to occur. States 3 and 4 merely force the order of messages to be *camera1.camera3.start_search*, *camera1.camera2.start_search*.

After this constraint restricts the model, CBs 2, 3, and 7 are removed, as Figure 5.32 shows, which indicates that these three CBs had the same cause, which was that *camera1* was following *Scenario5* while the other components were following *Scenario2*. Therefore, only CB 0 is left to be resolved.

Finally, CB 0 is analyzed, as it is the only unresolved CB. It is noticeable that IS 0 is the only one that starts with *camera2.object.sees_obj*, while the other ISs start with *camera3.object.sees_obj*. By further analyzing the sequence of messages of IS 0 and the bMSCs (shown in Appendix A.4), it is perceivable that the order of messages should follow *Scenario1*. However, in *Scenario1* *camera2* and *camera4* should ask the confidence of the other cameras before sending their confidence. This order is not observed in IS 0, as *camera1* sends a message to *camera2* before *camera4* sends an *ask_confidence* message.

Therefore, the third constraint is created to restrict this behavior, which is shown in Figure 5.33. To reduce space, let $\sum^4$ be $\sum \setminus \{$*camera2.object.sees_obj*$\}$, where $\sum$ is the set of all messages in the specification. In state 0, the constraint allows all messages to be exchanged, however when *camera2.object.sees_obj* is observed the transition to state 1 occurs. State 1, checks which message follows *camera2.object.sees_obj*, as this

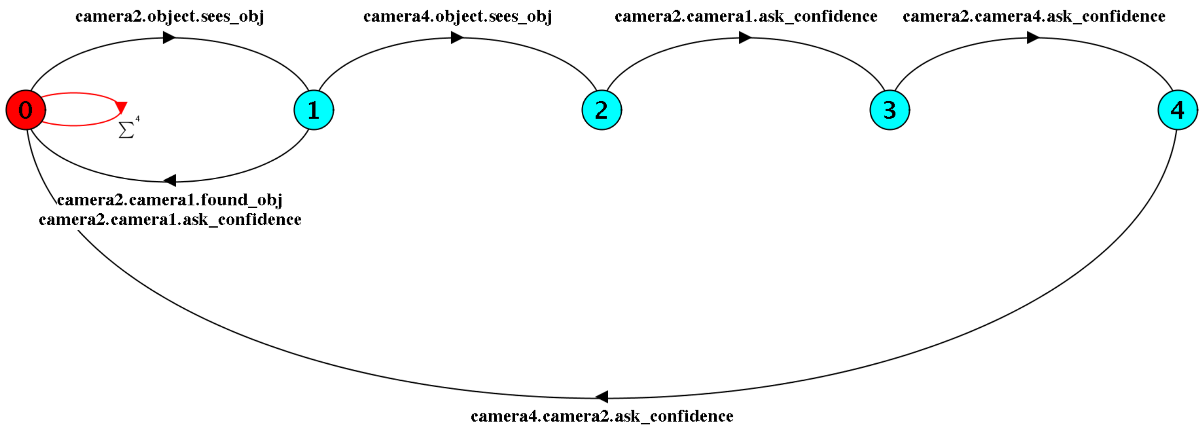

camera2.object.sees_obj
camera4.object.sees_obj
camera2.camera1.ask_confidence
camera2.camera4.ask_confidence
0
1
2
3
4
Σ^4
camera2.camera1.found_obj
camera2.camera1.ask_confidence
camera4.camera2.ask_confidence

Figure 5.33: Third constraint for SmartCam.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 5 out of 5
Common Behavior 0 -> traces reached: 0 out of 1
Common Behavior 1 -> traces reached: 0 out of 1
Common Behavior 2 -> traces reached: 0 out of 1
Common Behavior 3 -> traces reached: 0 out of 1
Common Behavior 4 -> traces reached: 0 out of 1
Common Behavior 5 -> traces reached: 0 out of 1
Common Behavior 6 -> traces reached: 0 out of 1
Common Behavior 7 -> traces reached: 0 out of 1
Common Behavior 8 -> traces reached: 0 out of 1

Total traces reached: 5 out of 14
```

Figure 5.34: Traces reached in SmartCam third constrained model.

message is different among *Scenario1* and the other scenarios. If the message exchanged is *camera4.object.sees_obj*, which indicates it is *Scenario1*, the transition to state 2 occurs. If it is not, then the transition to state 0 occurs. States 2, 3 and 4 merely force that all *ask_confidence* messages are sent before all other messages. Finally, after applying the third constraint to the model all CBs and ISs are removed, as shown in Figure 5.34.

Summary

The LTSA-MSC tool was only able to collect 9 ISs in the SmartCam specification, which makes it impossible to vary the number of ISs collected. Furthermore, the process of detecting CBs did not reduce the number of elements to be analyzed, as 9 CBs were detected for the 9 ISs. However, using the similarity between CBs in the manual analysis, it was possible to detect 3 families of CBs, and the creation of a constraint for each. By

Table 5.3: Time spent and # of CBs per ISs for eB2B

number of ISs	25	50	75	100	125	150	417
collection time (h:m:s)	0:00:27.804	0:01:32.477	0:03:50.922	0:09:01.721	0:14:45.148s	0:26:31.832	14:26:14.450
number of CBs	15	19	20	21	21	21	21
clustering time (s)	0.458030	0.542243	0.554717	0.562235	0.562235	0.562235	0.562235

these means, all collected ISs were resolved with only three constraints. Thus, supporting the hypothesis that various ISs can be treated together.

However, similarly to the previous case study (Section 5.2.3), the created constraints did not restrict general behaviors (such as querying the database after turning the system on, in Section 5.2.2), but instead restricted the order that the messages could appear to fit the modeled scenarios. Although it is possible that an architectural refinement would be better suited to remove the collected ISs, this analysis is not in the scope of this dissertation.

5.2.5 Case Study 5: eB2B System

System Description

The eB2B system *"is a system that allows access to an ERP (Enterprise Resource Planning) system, through a web interface"* [39], which allows the user to interact with the system through a web browser. The hMSC of the eB2B system is shown in Figure 5.35. The hMSC has a small adaptation from the original hMSC presented in [39], which is the addition of the *Logout* and *ShutDown* scenarios. The individual bMSCs are detailed in Appendix A.5.

The eB2B hMSC first starts up the system and then allows the user to login to the system. After the login succeeds, the user can either logout (which leads the system to shutdown) or enter the main loop of the system. In this loop, the user can use some criteria to find information and then view the details of the information.

Analysis

The LTSA-MSC tool was able to collect various numbers of ISs in the eB2B specification, which allowed the analysis of Table 5.3. It shows that even though more ISs were collected, the same CBs were detected starting at 100 ISs. However, the tool was unable to collect 500 ISs, as it ran out of memory when it searched for the 418th IS. That is, the trace of execution got so long without errors, that the tool could not extend it anymore with its available space. Therefore, the maximum number of ISs collected for eB2B is 417, which will be used for the analysis. Figure 5.36 shows that all 417 ISs and expected behaviors are reached in the LTS of the original model, and thus can happen at runtime.

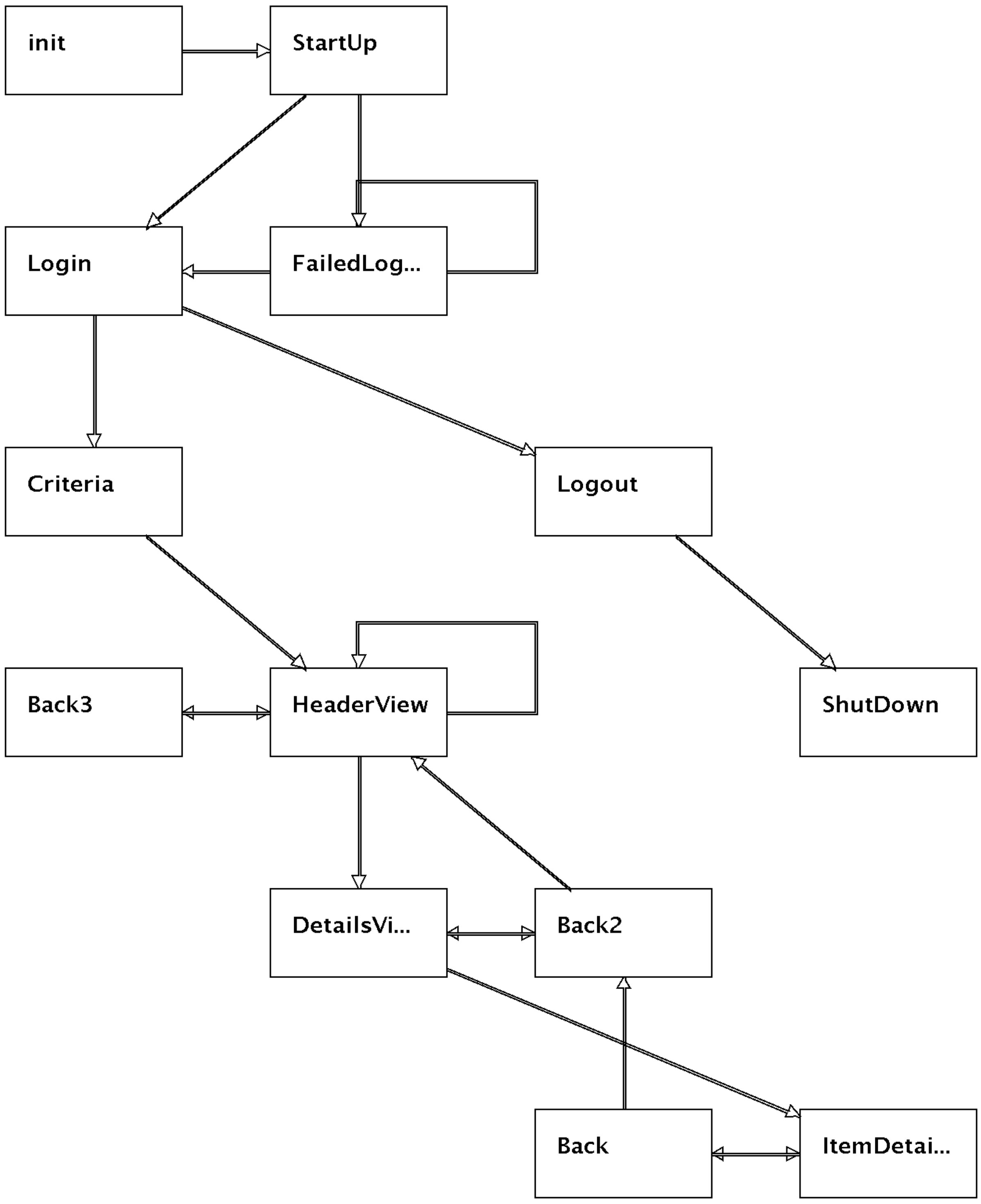


Figure 5.35: hMSC for eB2B specification.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 12 out of 12
Common Behavior 0 -> traces reached: 96 out of 96
Common Behavior 1 -> traces reached: 59 out of 59
Common Behavior 2 -> traces reached: 6 out of 6
Common Behavior 3 -> traces reached: 6 out of 6
Common Behavior 4 -> traces reached: 5 out of 5
Common Behavior 5 -> traces reached: 5 out of 5
Common Behavior 6 -> traces reached: 42 out of 42
Common Behavior 7 -> traces reached: 30 out of 30
Common Behavior 8 -> traces reached: 5 out of 5
Common Behavior 9 -> traces reached: 5 out of 5
Common Behavior 10 -> traces reached: 5 out of 5
Common Behavior 11 -> traces reached: 30 out of 30
Common Behavior 12 -> traces reached: 5 out of 5
Common Behavior 13 -> traces reached: 5 out of 5
Common Behavior 14 -> traces reached: 5 out of 5
Common Behavior 15 -> traces reached: 27 out of 27
Common Behavior 16 -> traces reached: 18 out of 18
Common Behavior 17 -> traces reached: 27 out of 27
Common Behavior 18 -> traces reached: 18 out of 18
Common Behavior 19 -> traces reached: 9 out of 9
Common Behavior 20 -> traces reached: 9 out of 9

Total traces reached: 429 out of 429
```

Figure 5.36: Traces reached in the eB2B original model.

```
0: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'shutdown']
1: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'shutdown']
2: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'failed', 'shutdown']
3: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'verified', 'shutdown']
4: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'failed', 'html', 'shutdown']
5: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'verified', 'html', 'shutdown']
6: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'shutdown']
7: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'orderHeader', 'shutdown']
8: ['run', 'enterPwd', 'shutdown']
9: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'verified', 'html', 'logout', 'shutdown']
10: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'verified', 'html', 'search', 'shutdown']
11: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'orderHeader', 'action', 'shutdown']
12: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'verified', 'html', 'search', 'action', 'shutdown']
13: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'verified', 'html', 'logout', 'action', 'shutdown']
14: ['run', 'enterPwd', 'authenticate', 'shutdown']
15: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'back', 'shutdown']
16: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'orderDetails', 'shutdown']
17: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'back', 'action', 'shutdown']
18: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'orderDetails', 'action', 'shutdown']
19: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'itemDetails', 'shutdown']
20: ['run', 'enterPwd', 'authenticate', 'query', 'data', 'xml', 'html', 'itemDetails', 'action', 'shutdown']
```

Figure 5.37: CBs detected for eB2B.

Using the 417 collected ISs, 21 CBs are detected, which are depicted in Figure 5.37. After using the SW algorithm and clustering the CBs, the dendrogram in Figure 5.38 is obtained. Next, the most similar pair of CBs is analyzed. However, there are six pairs of CBs tied as the most similar, all with a dissimilarity score of 0.04 ($\frac{1}{25}$). They are: CBs 7 and 11; CBs 9 and 13; CBs 10 and 12; CBs 15 and 17; CBs 16 and 18; CBs 19 and 20. Without loss of generality, the first pair, CBs 7 and 11, will be analyzed.

The CBs 7 and 11 share most of their messages, with the only difference being that CB 7 ends with *orderHeader* $\rightarrow$ *shutdown* while CB 11 has an extra message between those two, ending with *orderHeader* $\rightarrow$ *action* $\rightarrow$ *shutdown*. Furthermore, by analyzing the ending of all 21 detected CBs, it is noticeable that all of them end with *shutdown*, which is an indication that the cause of the CBs is related to the message *shutdown*.

According to the eB2B specification (Appendix A.5), the message *shutdown* only appears in the scenario *ShutDown*. The hMSC (Figure 5.35) specifies that this scenario can only happen after the *Logout* scenario. However, *logout* is the first message of the *Logout* scenario, which does not show up in any of the CBs. Therefore, a constraint can be created to prevent *shutdown* from happening before a *logout*.

Therefore, the constraint presented in Figure 5.39 is introduced. To reduce space, $\sum^1$ and $\sum^2$ were respectively used instead of $\sum \setminus \{$*logout*, *shutdown*$\}$ and $\sum \setminus \{$*html*, *shutdown*$\}$. In state 0, the constraint allows the system to exchange all messages, except for *shutdown*. If *logout* is observed, then the transition to state 1 occurs. In state 1, again, all messages except for *shutdown* are allowed, and when *html* is observed, the transition to state 2 occurs. In state 2, the only message allowed is *shutdown*, which starts the transition to state 0.

In other words, what the constraint does, is to restrict the occurrence of *shutdown* until the *Logout* scenario has ended. The *Logout* scenario starts with *logout* and ends

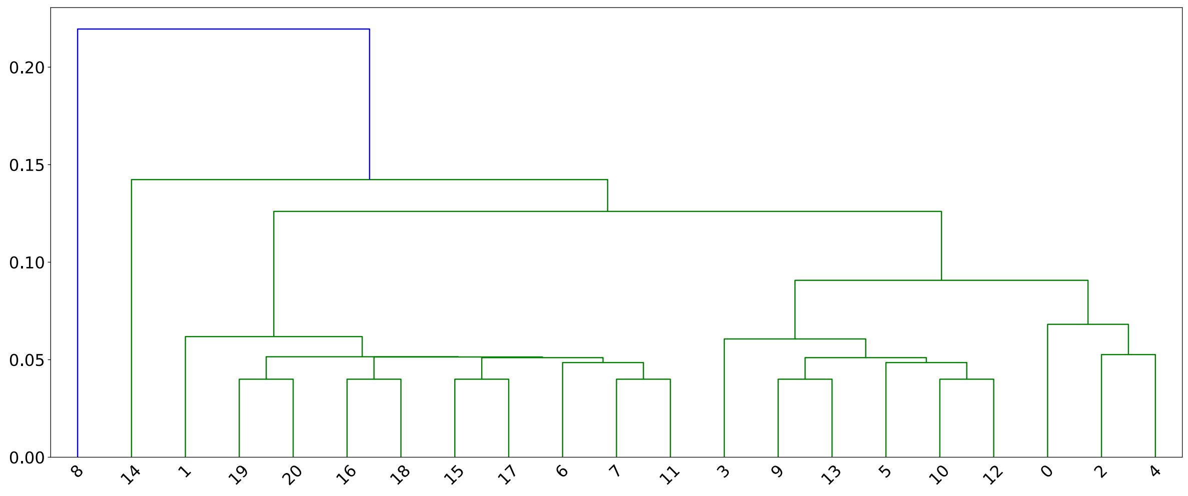


Figure 5.38: Dendrogram for eB2B.

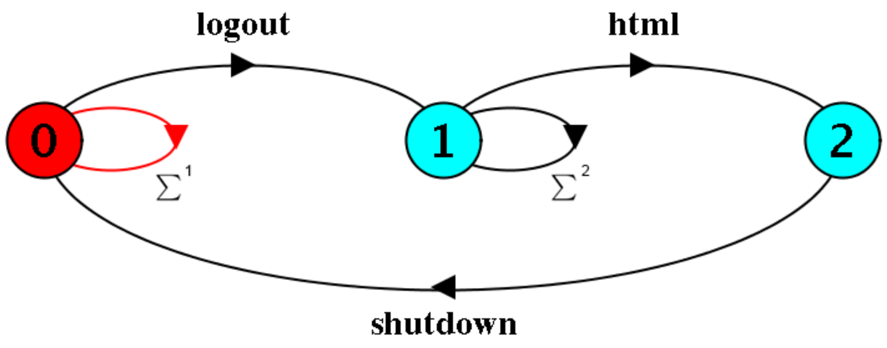


Figure 5.39: Constraint for eB2B.

when the user gets an html response (i.e., *html*). Therefore, when the LTS constraint reaches state 2, the *Logout* scenario has finished and thus *shutdown* can happen.

After applying the constraint to the original model, all 21 CBs are removed as shown in Figure 5.40, which shows that the constraint was correctly created. Also, this indicates that all detected CBs are part of the same family, as all of them are caused by the occurrence of *shutdown* out of place. Finally, note that in this case study, the same constraint could have been created if using only 15 CBs (i.e., 25 ISs) as the problem with the *shutdown* message would still be observable.

Summary

The LTSA-MSC tool was able to collect a varying number of ISs in the eB2B specification successfully, which allowed the analysis of Table 5.3. The table shows that, for the eB2B, even though the CBs increase at first, eventually there is a point where the number of detected CBs stabilizes. This indicates that it would not be necessary to detect more than ISs to realize the same analysis, as the same CBs were detected with 100, 125, 150,

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 12 out of 12
Common Behavior 0 -> traces reached: 0 out of 96
Common Behavior 1 -> traces reached: 0 out of 59
Common Behavior 2 -> traces reached: 0 out of 6
Common Behavior 3 -> traces reached: 0 out of 6
Common Behavior 4 -> traces reached: 0 out of 5
Common Behavior 5 -> traces reached: 0 out of 5
Common Behavior 6 -> traces reached: 0 out of 42
Common Behavior 7 -> traces reached: 0 out of 30
Common Behavior 8 -> traces reached: 0 out of 5
Common Behavior 9 -> traces reached: 0 out of 5
Common Behavior 10 -> traces reached: 0 out of 5
Common Behavior 11 -> traces reached: 0 out of 30
Common Behavior 12 -> traces reached: 0 out of 5
Common Behavior 13 -> traces reached: 0 out of 5
Common Behavior 14 -> traces reached: 0 out of 5
Common Behavior 15 -> traces reached: 0 out of 27
Common Behavior 16 -> traces reached: 0 out of 18
Common Behavior 17 -> traces reached: 0 out of 27
Common Behavior 18 -> traces reached: 0 out of 18
Common Behavior 19 -> traces reached: 0 out of 9
Common Behavior 20 -> traces reached: 0 out of 9

Total traces reached: 12 out of 429
```

Figure 5.40: Traces reached in eB2B constrained model.

and 417 ISs. Additionally, by a further analysis, it is possible that the same constraint that was created using the 21 CBs as a basis, could be achieved using only the 15 CBs detected with 25 ISs. By these means, the user would be able to save time and effort from collecting unnecessary extra ISs. Finally, through the analysis of the created constraint, it is clear that all 21 detected CBs are a part of the same family. Thus, all the collected ISs share the same cause.

5.2.6 Case Study 6: Global System for Mobile Mobility Management System

System Description

The Global System for Mobile Mobility Management System (GSM) was introduced by Leue et al. [40], specifies a system that keeps track of the location of GSM devices while allowing them to make and receive calls. It consists of four components, fourteen unique MSCs, and an hMSC that shows how they interact. However, in the description of the system specification, Leue et al. [40] describe some restrictions. For instance, if $CallSetupReq$ is executed on the second level of bMSCs from the top, then $MobileOrCS$ has to be chosen on the lower level. Thus, even though our model has the same fourteen unique bMSCs, some of them are repeated to accommodate these restrictions. Figure 5.41 shows our resulting hMSC, which has some repeated bMSCs (e.g., Accept1, Accept2, and Accept3). All individual bMSCs are included in the appendix of this dissertation (Appendix A.6). In this hMSC, three major loops can be identified from *ConnReq*:

1. *CallSetupReq* → *Identify/Authenticate* → *Accept/Reject* → *Encrypt/MobileORCS* → *MobileOrCR/MobileTrCR*;
2. *PagingResp* → *Identify/Authenticate* → *Accept/Reject* → *Encrypt/MobileTrCS* → *MobileOrCR/MobileTrCR*;
3. *LocUpdReq* → *Identify/Authenticate* → *Accept/Reject* → *Encrypt/LocationUpd*

The first describes the routine of the user initiating a call; the second describes the routine of the user receiving a call; and the third describes the routine of the network updating the location of the user. The two bottom scenarios (*MobileOrCR, MobileTrCR*) indicate that a call has been terminated (i.e., **C**all **R**elease), and thus return to the *ConnReq* so a new routine can start. *LocationUpd* also indicates the end of a routine, where the location has been successfully updated, and thus a new routine can start from *ConnReq*.

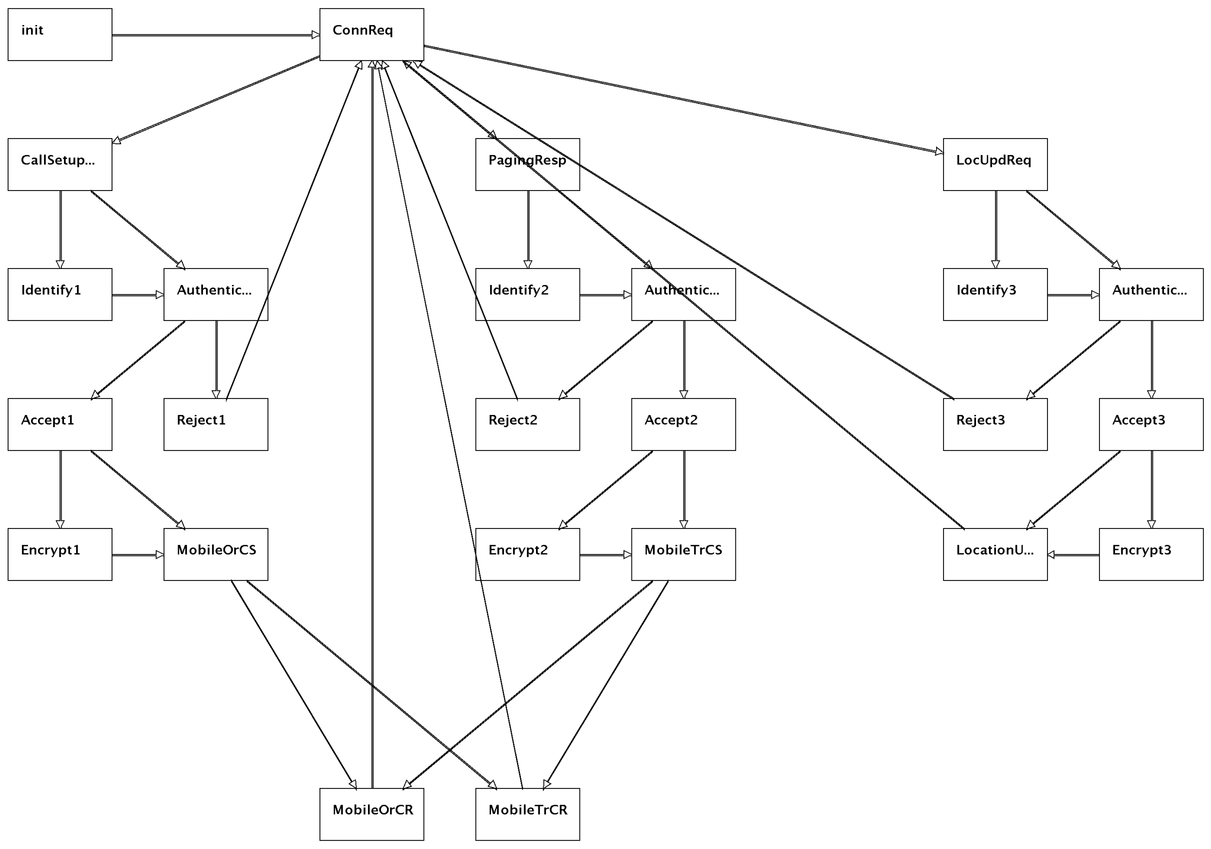


Figure 5.41: GSM hMSC.

Table 5.4: Time spent and # of CBs per ISs for GSM

number of ISs	25	50	75	100	125	150	357
collection time (h:m:s)	0:00:20.481	0:01:13.231	0:03:03.483	0:06:16.618	0:11:29.86	0:19:05.32	5:27:31.018
number of CBs	16	16	16	16	16	16	16
clustering time (s)	0.841828	0.841828	0.841828	0.841828	0.841828	0.841828	0.841828

Analysis

The LTSA-MSC tool was able to collect various numbers of ISs in the GSM specification, which allowed the analysis of Table 5.4. It shows that even though more ISs were collected, the same CBs were detected. However, the tool was unable to collect 500 ISs, as it ran out of memory when it searched for the 358th IS. That is, the trace of execution got so long without errors, that the tool could not extend it anymore with its available space. Therefore, the maximum number of ISs collected for GSM is 357, which will be used for the analysis. Figure 5.43 shows that all 357 ISs and expected behaviors are reached in the LTS of the original model, and thus can happen at runtime.

Therefore, it is needed to analyze the 16 detected CBs, which are shown in Figure 5.42. Now, the Smith-Waterman algorithm is applied to all possible pairs of the CBs detected. By using the dissimilarity function and Ward's method, the dendrogram in Figure 5.44 is obtained. The most similar pairs were CBs 14 and 15, with a dissimilarity of $\frac{1}{91}$.

Next, we can start looking at the most similar common behaviors, which are CBs 14 and 15, according to our dendrogram. By looking at their traces, it is possible to see that they share a very large prefix, until the last three messages:
channelReq, immAssign, pageResp, pageRspAck, pageRspReq, procAccessReq, provideImsi, identityReq, identityResp, imsiAck, authenticate, authenReq, authenResp, authenComplt, serviceAccept, setCipherMode, cipherModeCmnd, cipherMode, ciphModeCmplt, ciphCmplt, callSetup, callSUp, confirm, conf, addCmplt, alerting, alert, alrt, conct

Even more, they also have the same last three messages ($answer, discon, release$). However, the three last messages are in a different order. Their shared messages tell us that both CBs are executing the following bMSCs:

$$init, ConnReq, PagingResp, Identify2, Authenticate2, Accept2, Encrypt2, MobileTrCS$$

At the end of the receiving call routine, only one out of $MobileOrCR$ and $MobileTrCR$ should happen according to GSM's specification (Figure 5.41). In addition, *discon* is the first message of $MobileOrCR$, and *release* is the first message of $MobileTrCR$. This tells us that in both CBs, both scenarios are starting, which is not allowed in the original specification. Leue et al. [40], details these scenarios as call releases (CR), that is, the

```
0:channelReq,immAssign,callSetup,callSetupReq,callSetUp,authenticate,authenReq,authenResp,authenComplt,ser
viceAccept,assignComnd,assigned,assignCmplt,assignCmpt,initialMsg,addCmpltMsg,alerting,answerMsg,connect,c
onnectAck,discon,release

1:channelReq,immAssign,callSetup,callSetupReq,callSetUp,authenticate,authenReq,authenResp,authenComplt,ser
viceAccept,assignComnd,assigned,assignCmplt,assignCmpt,initialMsg,addCmpltMsg,alerting,answerMsg,connect,c
onnectAck,release,discon

2:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,authenticate,authenReq,authenResp,auth
enComplt,serviceAccept,callSetup,callSUp,confirm,conf,addCmplt,alerting,alert,alrt,conct,discon,answer,rel
ease

3:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,authenticate,authenReq,authenResp,auth
enComplt,serviceAccept,callSetup,callSUp,confirm,conf,addCmplt,alerting,alert,alrt,conct,answer,release,di
scon

4:channelReq,immAssign,callSetup,callSetupReq,callSetUp,provideImsi,identityReq,identityResp,imsiAck,authe
nticate,authenReq,authenResp,authenComplt,serviceAccept,assignComnd,assigned,assignCmplt,assignCmpt,initia
lMsg,addCmpltMsg,alerting,answerMsg,connect,connectAck,discon,release

5:channelReq,immAssign,callSetup,callSetupReq,callSetUp,provideImsi,identityReq,identityResp,imsiAck,authe
nticate,authenReq,authenResp,authenComplt,serviceAccept,assignComnd,assigned,assignCmplt,assignCmpt,initia
lMsg,addCmpltMsg,alerting,answerMsg,connect,connectAck,release,discon

6:channelReq,immAssign,callSetup,callSetupReq,callSetUp,authenticate,authenReq,authenResp,authenComplt,ser
viceAccept,setCipherMode,cipherModeCmnd,cipherMode,ciphModeCmplt,ciphCmplt,assignComnd,assigned,assignCmpl
t,assignCmpt,initialMsg,addCmpltMsg,alerting,answerMsg,connect,connectAck,discon,release

7:channelReq,immAssign,callSetup,callSetupReq,callSetUp,authenticate,authenReq,authenResp,authenComplt,ser
viceAccept,setCipherMode,cipherModeCmnd,cipherMode,ciphModeCmplt,ciphCmplt,assignComnd,assigned,assignCmpl
t,assignCmpt,initialMsg,addCmpltMsg,alerting,answerMsg,connect,connectAck,release,discon

8:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,provideImsi,identityReq,identityResp,i
msiAck,authenticate,authenReq,authenResp,authenComplt,serviceAccept,callSetup,callSUp,confirm,conf,addCmpl
t,alerting,alert,alrt,conct,discon,answer,release

9:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,provideImsi,identityReq,identityResp,i
msiAck,authenticate,authenReq,authenResp,authenComplt,serviceAccept,callSetup,callSUp,confirm,conf,addCmpl
t,alerting,alert,alrt,conct,answer,release,discon

10:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,authenticate,authenReq,authenResp,aut
henComplt,serviceAccept,setCipherMode,cipherModeCmnd,cipherMode,ciphModeCmplt,ciphCmplt,callSetup,callSUp,
confirm,conf,addCmplt,alerting,alert,alrt,conct,discon,answer,release

11:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,authenticate,authenReq,authenResp,aut
henComplt,serviceAccept,setCipherMode,cipherModeCmnd,cipherMode,ciphModeCmplt,ciphCmplt,callSetup,callSUp,
confirm,conf,addCmplt,alerting,alert,alrt,conct,answer,release,discon

12:channelReq,immAssign,callSetup,callSetupReq,callSetUp,provideImsi,identityReq,identityResp,imsiAck,auth
enticate,authenReq,authenResp,authenComplt,serviceAccept,setCipherMode,cipherModeCmnd,cipherMode,ciphModeC
mplt,ciphCmplt,assignComnd,assigned,assignCmplt,assignCmpt,initialMsg,addCmpltMsg,alerting,answerMsg,conne
ct,connectAck,discon,release

13:channelReq,immAssign,callSetup,callSetupReq,callSetUp,provideImsi,identityReq,identityResp,imsiAck,auth
enticate,authenReq,authenResp,authenComplt,serviceAccept,setCipherMode,cipherModeCmnd,cipherMode,ciphModeC
mplt,ciphCmplt,assignComnd,assigned,assignCmplt,assignCmpt,initialMsg,addCmpltMsg,alerting,answerMsg,conne
ct,connectAck,release,discon

14:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,provideImsi,identityReq,identityResp,
imsiAck,authenticate,authenReq,authenResp,authenComplt,serviceAccept,setCipherMode,cipherModeCmnd,cipherMo
de,ciphModeCmplt,ciphCmplt,callSetup,callSUp,confirm,conf,addCmplt,alerting,alert,alrt,conct,discon,answer
,release

15:channelReq,immAssign,pageResp,pageRspAck,pageRspReq,procAccessReq,provideImsi,identityReq,identityResp,
imsiAck,authenticate,authenReq,authenResp,authenComplt,serviceAccept,setCipherMode,cipherModeCmnd,cipherMo
de,ciphModeCmplt,ciphCmplt,callSetup,callSUp,confirm,conf,addCmplt,alerting,alert,alrt,conct,answer,releas
e,discon
```

Figure 5.42: CBs detected for GSM.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 26 out of 26
Common Behavior 0 -> traces reached: 46 out of 46
Common Behavior 1 -> traces reached: 46 out of 46
Common Behavior 2 -> traces reached: 44 out of 44
Common Behavior 3 -> traces reached: 43 out of 43
Common Behavior 4 -> traces reached: 21 out of 21
Common Behavior 5 -> traces reached: 20 out of 20
Common Behavior 6 -> traces reached: 19 out of 19
Common Behavior 7 -> traces reached: 18 out of 18
Common Behavior 8 -> traces reached: 18 out of 18
Common Behavior 9 -> traces reached: 18 out of 18
Common Behavior 10 -> traces reached: 14 out of 14
Common Behavior 11 -> traces reached: 14 out of 14
Common Behavior 12 -> traces reached: 10 out of 10
Common Behavior 13 -> traces reached: 10 out of 10
Common Behavior 14 -> traces reached: 8 out of 8
Common Behavior 15 -> traces reached: 8 out of 8

Total traces reached: 383 out of 383
```

Figure 5.43: Traces reached in GSM original model.

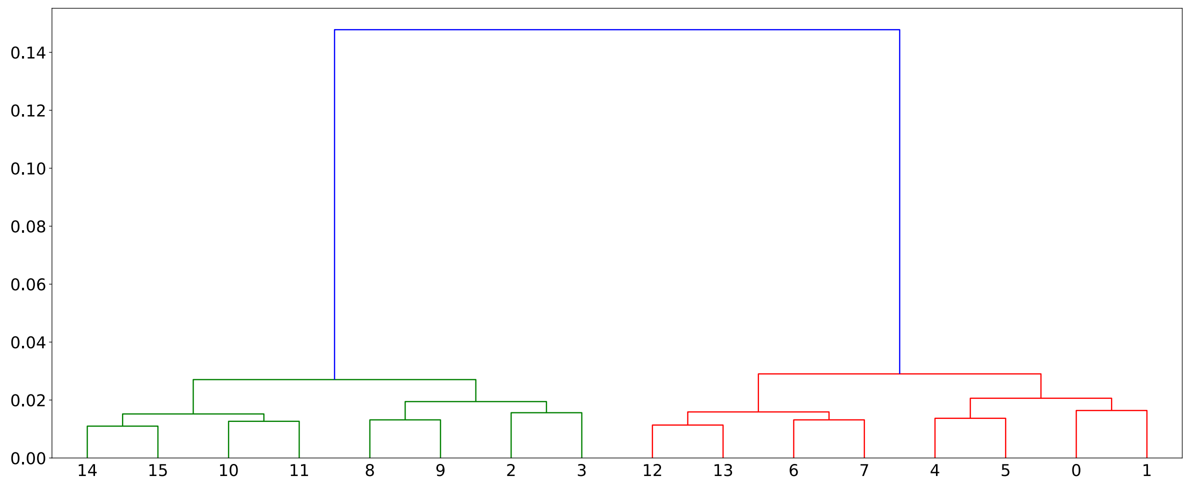


Figure 5.44: Dendrogram for GSM.

user's call has been finished. The difference between them is that one describes the scenario in which the user ends the call ($MobileOrCR$), and the other describes when the network ends the call ($MobileTrCR$).

Therefore, the start of both CR scenarios in CBs 14 and 15 tells us that either the user or the network is trying to end a call that has already been finished, thus resulting in unexpected behavior. This cannot happen according to the specification, as only one call release scenario can happen for each call. This way, it is possible to resolve both CBs at once by limiting the system to always execute only one of those two scenarios for each call. Additionally, through manual analysis of the 16 detected CBs for the GSM System, it is noticeable that all common behaviors share that same problem. That is, every IS detected for this system, happened because both Call Release scenarios were executing concurrently. Therefore, for GSM we have a single family of common behaviors, which contains all detected CBs, consequently, all collected implied scenarios.

From the manual analysis, it is known that all detected CBs are in the same family, and happen because both call release (CR) scenarios are occurring concurrently at runtime. Thus, we created a single constraint to prevent both CR scenarios from happening together. Figure 5.45 shows the constraint created to resolve this problem. To reduce space, let $\sum$ represent all messages in the system's alphabet, $\sum^1$ is used to represent all messages in the system's alphabet with the exception of $\{release, discon\}$ (i.e., $\sum^1 = \sum \setminus \{release, discon\}$). Analogously, $\sum^2 = \sum^1 \setminus \{channelReq\}$. This other message (*channelReq*) is removed from $\sum^2$ because it is used for the transition that goes from state 1 to state 0.

Succinctly, this constraint does not interfere with the system unless either call release scenarios start, which is detected by observing either *discon* or *release*. When either message is exchanged, the transition to state 1 occurs. In state 1 we prohibit the system to start another call release routine, that is, all messages are allowed but *discon* and *release* ($\sum^2$). When *channelReq* is observed, the system is getting ready to start a new routine (i.e., start a call, receive a call, or update location), and thus a call release scenario can happen in the future. Therefore, *channelReq* makes the transition to state 0 occurs.

Since only one CR scenario will happen when this constraint is applied to the original GSM specification, all CBs have been dealt with using this same constraint. Consequently, by constraining the model in such way, all ISs will also be prevented during runtime. Therefore, to make sure that the collected ISs have been avoided Algorithm 2 is used. Figure 5.46 shows that the collected ISs cannot happen at runtime after the model is constrained by the created constraint. Thus, the family of CBs was well defined, as all CBs were resolved with the same treatment, which was the creation of a constraint.

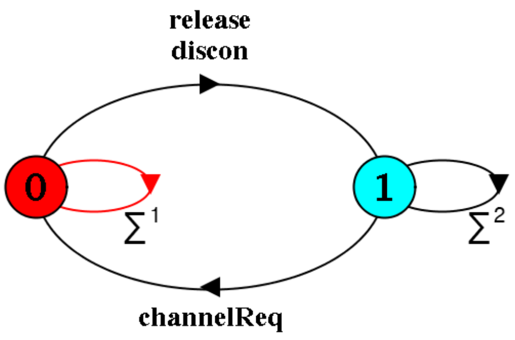


Figure 5.45: LTS of GSM's constraint.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 26 out of 26
Common Behavior 0 -> traces reached: 0 out of 46
Common Behavior 1 -> traces reached: 0 out of 46
Common Behavior 2 -> traces reached: 0 out of 44
Common Behavior 3 -> traces reached: 0 out of 43
Common Behavior 4 -> traces reached: 0 out of 21
Common Behavior 5 -> traces reached: 0 out of 20
Common Behavior 6 -> traces reached: 0 out of 19
Common Behavior 7 -> traces reached: 0 out of 18
Common Behavior 8 -> traces reached: 0 out of 18
Common Behavior 9 -> traces reached: 0 out of 18
Common Behavior 10 -> traces reached: 0 out of 14
Common Behavior 11 -> traces reached: 0 out of 14
Common Behavior 12 -> traces reached: 0 out of 10
Common Behavior 13 -> traces reached: 0 out of 10
Common Behavior 14 -> traces reached: 0 out of 8
Common Behavior 15 -> traces reached: 0 out of 8

Total traces reached: 26 out of 383
```

Figure 5.46: Traces reached in GSM constrained model.

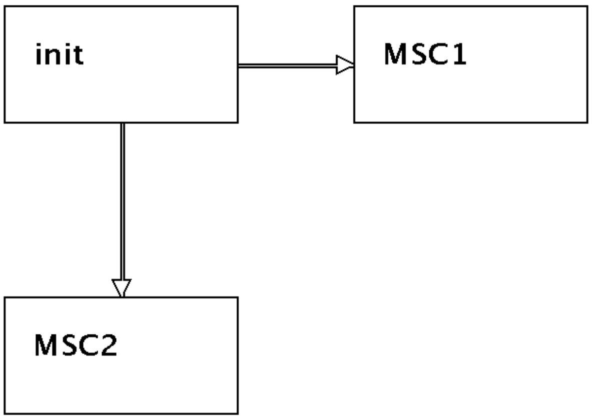


Figure 5.47: hMSC of S.S. MAS.

Summary

A varying number of ISs were collected in the GSM specification, which allowed the analysis in Table 5.4. The table shows that even though a large number ISs were collected, few CBs were detected. Because the manual analysis uses the CBs, detecting more ISs does not help in the analysis if the extra ISs do not have different CBs. It is shown that the constraint created with basis on the detected CBs is able to remove all collected ISs. That is, the family of CBs was correctly characterizing the cause of a multitude of emergent behaviors. This shows that is possible to remove multiple ISs when their common cause is correctly identified.

5.2.7 Case Study 7: Semantic Search Multi-Agent System

System Description

The Semantic Search Multi-Agent System (S.S. MAS) was introduced by Moshirpour et al. [16]. It describes a system which receives queries from the user, and if the system already knows the answer to the query it returns so (MSC1), if not, the system firstly learns the answer and then returns (MSC2). The hMSC is shown in Figure 5.47. The S.S. MAS hMSC merely allows for either scenario to happen, that is, it allows for the user to query the system, and then the system acts accordingly to whether or not it knows the answer to the query.

The system originally has 5 components: *User*, *Query Handler (QH)*, *Concept Learner (CL)*, *Repository (Rep)*, and *Peer*. However, because *QH* and *CL* send messages to themselves, which is not allowed in the LTSA-MSC tool, two auxiliary components were created: *QHaux* and *CLaux*. The bMSCs of these system are shown in Appendix A.7.

```
ID: [ordered messages]

0: [user.qh.enterQuery, qh.qhaux.extractConcepts,
qhaux.qh._extractConcepts, qh.cl.sendConcept, qh.rep.sendConcept,
rep.qh.returnResults, cl.claux.newConcepts]

1: [user.qh.enterQuery, qh.qhaux.extractConcepts,
qhaux.qh._extractConcepts, qh.cl.sendConcept, qh.rep.sendConcept,
cl.claux.newConcepts, rep.qh.returnResults]

2: [user.qh.enterQuery, qh.qhaux.extractConcepts,
qhaux.qh._extractConcepts, qh.cl.sendConcept, qh.rep.sendConcept,
cl.claux.newConcepts, claux.cl._newConcepts, rep.qh.returnResults]

3: [user.qh.enterQuery, qh.qhaux.extractConcepts,
qhaux.qh._extractConcepts, qh.cl.sendConcept, qh.rep.sendConcept,
cl.claux.newConcepts, claux.cl._newConcepts,
cl.qh.newConceptsDetected, rep.qh.returnResults]

4: [user.qh.enterQuery, qh.qhaux.extractConcepts,
qhaux.qh._extractConcepts, qh.cl.sendConcept, qh.rep.sendConcept,
cl.claux.newConcepts, claux.cl._newConcepts,
cl.qh.newConceptsDetected, cl.peer.startLearning,
rep.qh.returnResults]

5: [user.qh.enterQuery, qh.qhaux.extractConcepts,
qhaux.qh._extractConcepts, qh.cl.sendConcept, qh.rep.sendConcept,
cl.claux.newConcepts, claux.cl._newConcepts,
cl.qh.newConceptsDetected, cl.peer.startLearning,
peer.cl.learnConcepts, rep.qh.returnResults]
```

Figure 5.48: Collected ISs for S.S. MAS.

Analysis

The LTSA-MSC tool was only able to collect a total of 6 ISs for the S.S. MAS, which are shown in Figure 5.48. That is, even though a larger collection of ISs was desired, the tool was only able to detect 6 ISs in the system specification, which prevented repeated analysis with other numbers of ISs. Therefore, the collection process was repeated to try to collect 25 ISs nine more times. However, the same 6 ISs were collected in every repetition, and they were also detected in the same order.

The collection process took on average 3.213s, with an standard deviation of 0.118s, and the collected ISs are shown in Figure 5.48. Each message follows the structure: *component1.component2.message*, which means that *component1* sent *message* to *component2*. This notation is used to differentiate messages such as *qh.cl.sendConcept* and *qh.rep.sendConcept*. Finally, Figure 5.49 shows that all collected ISs and expected behaviors can happen in the original model.

After the collection of ISs, the process to detect common behaviors was done. However, like for APTS and SmartCam, each IS had a unique CB, and thus the number of elements was not reduced. Therefore, the CBs detected are equivalent to the ISs shown in Figure 5.48. With the CBs detected, the Smith-Waterman algorithm is applied to all

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 1 out of 1
Common Behavior 1 -> traces reached: 1 out of 1
Common Behavior 2 -> traces reached: 1 out of 1
Common Behavior 3 -> traces reached: 1 out of 1
Common Behavior 4 -> traces reached: 1 out of 1
Common Behavior 5 -> traces reached: 1 out of 1

Total traces reached: 8 out of 8
```

Figure 5.49: Traces reached in S.S. MAS original model.

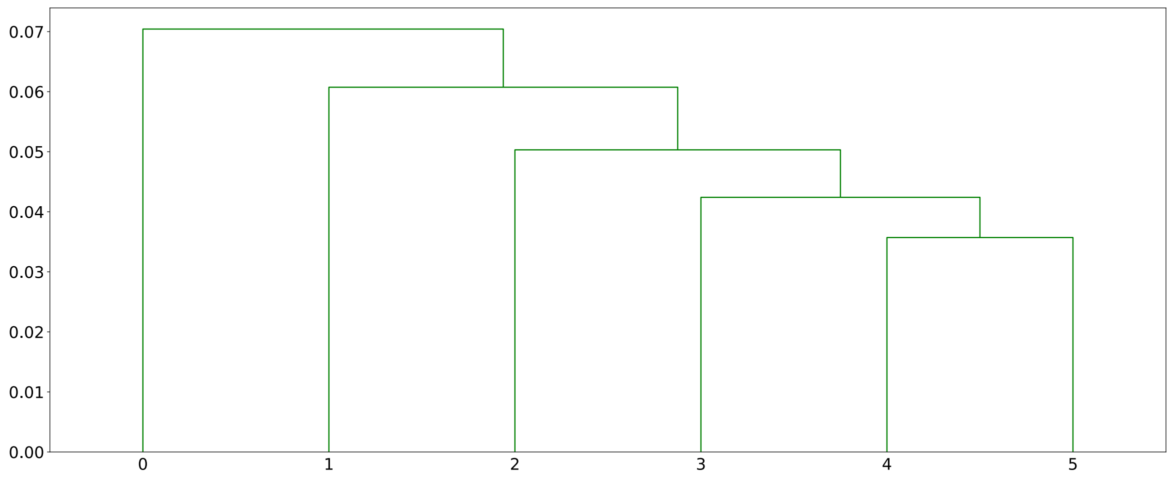


Figure 5.50: Dendrogram for S.S. MAS.

possible pairs, and the dissimilarity is calculated. Next, the dendrogram in Figure 5.50. The process of detecting CBs, applying the SW algorithm to the pairs of CBs, clustering the CBs, and exporting the dendrogram took 0.315s.

Next, a manual analysis of the obtained results is required. The analysis starts from the less dissimilar pair (i.e., the lowest hight in the dendrogram), which is the pair of CBs 4 and 5. By looking at the sequence of messages of each CB (ISs 4 and 5 from Figure 5.48), it is noticeable that they share most of their messages. Also, because they have the message *cl.claux.newConcepts*, they both indicate that a new concept was queried, and thus they should learn that concept and update the repository. However, before they return the results, the repository should be updated with this new concept. Therefore, the constraint shown in Figure 5.51 is created.

To preserve space, $\sum^1$ and $\sum^2$ were respectively used instead of $\sum \setminus \{cl.rep.updateRep$, $rep.qh.returnResults\}$ and $\sum \setminus \{rep.qh.returnResults\}$, where $\sum$ is the set of all messages in the S.S. MAS specification. The constraint starts in state 0 and allows all messages

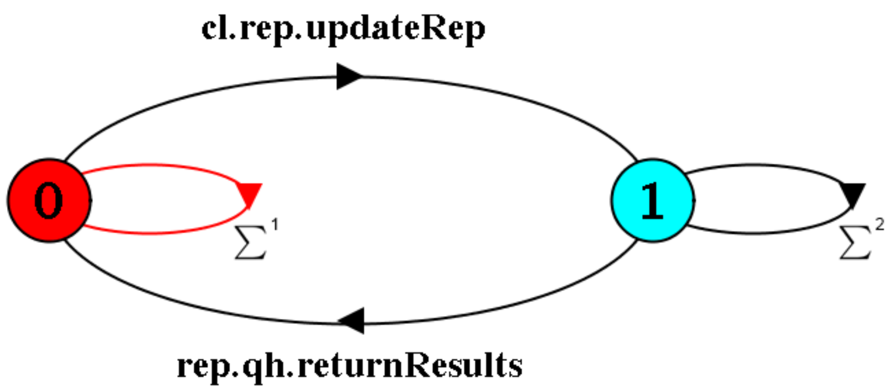


Figure 5.51: Initial constraint for S.S. MAS.

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 1 out of 2
Common Behavior 0 -> traces reached: 0 out of 1
Common Behavior 1 -> traces reached: 0 out of 1
Common Behavior 2 -> traces reached: 0 out of 1
Common Behavior 3 -> traces reached: 0 out of 1
Common Behavior 4 -> traces reached: 0 out of 1
Common Behavior 5 -> traces reached: 0 out of 1

Total traces reached: 1 out of 8
```

Figure 5.52: Traces reached in S.S. MAS first constrained model.

except for *returnResults*. This is done to prevent results from being returned before the repository is updated. Therefore, when the update is observed (*updateRep*), the transition to state 1 occurs. In state 1, all messages are allowed to be exchanged. However, when *returnResults* is observed, the transition to state 0 occurs, in order to require a new update before new results are returned.

This constraint removes the unwanted behaviors, as shown in Figure 5.52. In fact, all 6 detected CBs are removed by adding the required update before returning the results. However, unlike the constraints for the other systems, one wanted behavior is also restricted – as not always the repository will be updated before returning results. That happens because there are no required updates if there are no new concepts to be learned. In other words, the constraint in Figure 5.51 restricts the return of results only to when new concepts are learned, and thus the repository is updated.

Hence, an alteration is required in the first constraint. Because the behavior of waiting for the repository update is wanted, when there are new concepts, a new message is added

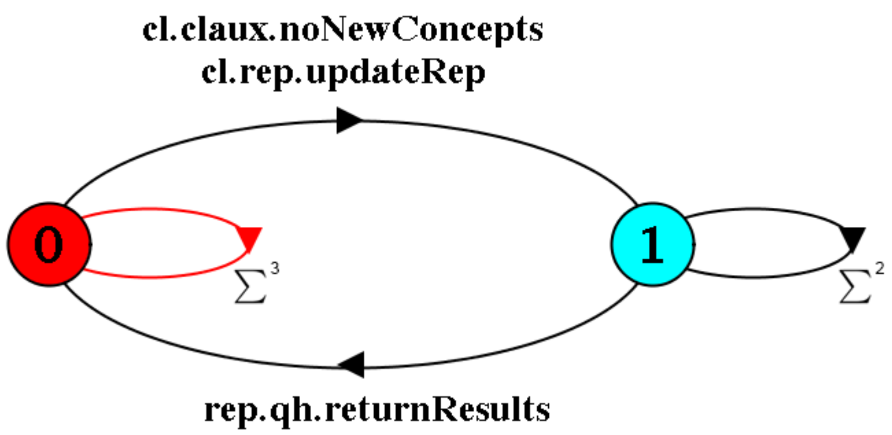


Figure 5.53: Altered constraint for S.S. MAS.

to the transition from state 0 to state 1 – *cl.claux.noNewConcepts*. This change is done to allow the return of results to happen after either the system detected that there is no need of learning new concepts or the system has already learned the new concepts and updated its repository. Therefore, the initial constraint from Figure 5.51 is altered to the constraint shown in Figure 5.53.

The only change made from the initial constraint is the removal of *cl.claux.noNewConcepts* from $\sum^1$, which makes $\sum^3 = \sum^1 \setminus \{cl.claux.noNewConcepts\}$, and the addition of the same message in the transition from state 0 to state 1. By these means, state 0 allows all messages but *return*Results, which can only be exchanged in state 1. To get to state 1, the system needs to exchange either *cl.claux.noNewConcepts* which indicates that it already knows all necessary concepts, or *cl.rep.updateRep*, which indicates that there were new concepts, but they have already been included in the repository.

Therefore, the fixed constraint preserves the wanted behaviors modeled in the system, as well as removes the unwanted ones, which is returning results before the system is ready. Finally, after constraining the original model with this new constraint, all CBs are removed as seen in Figure 5.54.

Summary

The LTSA-MSC tool was only able to collect only 6 ISs in the S.S. MAS specification, which makes it impossible to vary the number of ISs collected. Furthermore, the process of detecting CBs did not reduce the number of elements to be analyzed, as 6 CBs were detected for the 6 ISs. However, using the similarity between CBs and through further analysis, it became clear that all detected CBs shared the same cause, which was the

```
RESULTS SUMMARY:

Expected Behaviors -> traces reached: 2 out of 2
Common Behavior 0 -> traces reached: 0 out of 1
Common Behavior 1 -> traces reached: 0 out of 1
Common Behavior 2 -> traces reached: 0 out of 1
Common Behavior 3 -> traces reached: 0 out of 1
Common Behavior 4 -> traces reached: 0 out of 1
Common Behavior 5 -> traces reached: 0 out of 1

Total traces reached: 2 out of 8
```

Figure 5.54: Traces reached in S.S. MAS second constrained model.

incorrect return of results. Thus, a single family was defined. Therefore, it was ultimately possible to remove all collected ISs with a single constraint.

5.3 Discussion

Table 5.5 shows the summary of the obtained results for all case studies. For all systems, we tried to collect a varying number of ISs up to 500. Accordingly, in the second column, *# of ISs* of Table 5.5 the maximum number of implied scenarios that were actually collected is shown. Then, in *Collection*, the time spent collecting the ISs with the LTSA-MSC tool is presented. Next, in *Clustering* column accounts for the total time spent on: (1) detecting common behaviors, (2) calculating the distance for all possible pairs with the SW algorithm, (3) clustering them, and (4) exporting the dendrogram. Finally, in *CBs* column, the number of detected common behaviors for the system, and in *Constraints* column the number of constraints needed to resolve said common behaviors.

It is important to note that the time required to detect common behaviors, as well as to apply the Smith-Waterman algorithm, cluster all pairs, and export the dendrogram for all experiments never reached 1s. Because it is much smaller than times observed in the collection process with the LTSA tool, this part of the process does not contribute significantly to the time necessary for our methodology.

From the results presented in Table 5.5, for APTS, S.S. MAS, and SmartCam the tool could not collect more implied scenarios, that is, the tool collected all detectable ISs following the Uchitel et al. approach. When analyzing these systems, there are as many common behaviors as implied scenarios, which did not reduce the number of elements that had to be analyzed. However, the process of finding families of common behaviors did help, as only seven constraints were needed to resolve all 24 ISs across these systems.

On the other hand, for the other four systems,i.e. Boiler, Cruiser, eB2B, and GSM, it is possible that the tool would collect more implied scenarios than the ones detected. Even more, the time spent collecting them was rather large, and for eB2B and GSM systems, the tool ran out of memory and aborted the execution, collecting only 417 and

Table 5.5: Summary of results across all systems studied

System	# of ISs	Collection (h:m:s)	Clustering (s)	CBs	Constraints
APTS	9	0:00:03.895	0.331548	9	3
Boiler	500	7:35:51.831	0.318115	2	2
Cruiser	500	9:26:55.046	0.497454	18	3
eB2B	417	14:26:14.450	0.562235	21	1
GSM	357	5:27:31.018	0.841828	16	1
S.S. MAS	6	0:00:03.213	0.315708	6	1
SmartCam	9	0:00:04.306	0.327333	9	3
Total	1798	36:56:43.759	3.186659	81	14

Table 5.6: Number of CBs detected per number of ISs collected

System	up to 25 ISs	50 ISs	75 ISs	100 ISs	125 ISs	150 ISs	up to 500 ISs
APTS	9	-	-	-	-	-	-
Boiler	2	2	2	2	2	2	2
Cruiser	18	18	18	18	18	18	18
eB2B	15	19	20	21	21	21	21
GSM	16	16	16	16	16	16	16
S.S. MAS	6	-	-	-	-	-	-
SmartCam	9	-	-	-	-	-	-

Table 5.7: Time (h:m:s) to collect *n* implied scenarios

System	100 ISs	up to 500 ISs
Boiler	0:03:05.820	7:35:51.831
Cruiser	0:05:10.123	9:26:55.046
eB2B	0:09:01.721	14:26:14.450
GSM	0:06:16.618	5:27:31.018
Total	0:23:34.282	36:56:32.345

357 implied scenarios, respectively. However, IS collection could still be counting if the tool had more memory available.

Additionally, for these four systems, there is a significant decrease from the number of implied scenarios collected to the number of common behaviors detected, which facilitates the analysis process for the user. Furthermore, as Table 5.6 shows, the number of common behaviors detected remains the same starting at 100 implied scenarios collected for all four systems, that is, there is no point in detecting more than 100 ISs as the manual analysis and treatment take only common behaviors in consideration. In fact, only eB2B requires that many implied scenarios, as for Boiler, Cruiser, and GSM, 25 would suffice.

Thus, it would be possible to get the same results (i.e., CBs and constraints) by using only 25 ISs for the Boiler, Cruiser, and GSM systems, and only 100 ISs for the eB2B system, as the same common behaviors are detected by using either 25/100 or up to 500 ISs. Nevertheless, consider that for all these four systems 100 implied scenarios were collected. By doing so, the time spent by the collection process would drastically drop, as shown in Table 5.7, while the time required for analysis and treatment would not be affected, as for those processes only the detected CBs are used as input, and these would remain the same. This way, the other steps of the methodology would not be affected and only 1% of the original time would be required in the collection process.

Therefore, the obtained results support the validity of our proposal. Table 5.5 shows that our proposed methodology facilitates the analysis process for the user, as for the studied systems they would only need to analyze at most 4.5% of the cases (81 CBs instead of 1798 ISs). Even more so, since there are families with more than one CB, this number goes down even further to 14 families (0.7%). Also, by looking at the number of common behaviors detected, it would also allow the user to stop collecting implied scenarios when he/she notices a constant number of CBs, which would decrease the time spent collecting ISs. This way, our approach can decrease the time required to collect ISs, and, after the collection process, it facilitates the treatment process by reducing the number of elements analyzed.

Furthermore, except for the Boiler system, the analysis to define families of common behaviors allowed us to define fewer constraints than the actual number of common behaviors they resolved. For instance, a single constraint for the eB2B system resolved all 21 common behaviors, which consequently resolved all 417 collected implied scenarios in the system specification.

Nonetheless, the methodology still requires the user's input. For instance, notice that at first, it looks that there are two distinct groups of CBs for the GSM system. However the dissimilarity between these two groups is around 0.15, which is smaller than the dissimilarity in the Boiler example, which was 0.25. From this information, we conclude that all 16 CBs in the GSM system are more similar to each other than the only two CBs in the Boiler example. Further studies could be performed to explore means to automatically define families of CBs.

Additionally, we validated the choice for the Smith-Waterman algorithm by comparing the clustering results presented with the clustering results obtained by using the Levenshtein distance [18] as the distance metric. The two clustering results were compared to the groups obtained by the definition of families of CBs, using the Jaccard index [41]. The Jaccard index is a similarity metric that can be used to compare how close two clustering results are to each other, and it ranges from 0 to 1 with a higher index indicating more similar results. Therefore, we compare the clusterings obtained with the Smith-Waterman algorithm and the Levenshtein distance to the families of CBs obtained, to discover which similarity metric (Levenshtein or Smith-Waterman) gets a closer result to the families desired.

The results of the comparisons are shown in Table 5.8, where the second column shows the Jaccard index comparing the families of CBs to the clustering using Levenshtein distance, and the third column shows the Jaccard index comparing the families of CBs to the clustering using Smith-Waterman. The clustering with Smith-Waterman obtained a better result in four out of the seven case studies, while the clustering with Levenshtein was better in only one, and they were equal in two cases.

Furthermore, it is important to note that in the Boiler case study, the similarity

Table 5.8: Clustering results comparison

System	Levenshtein	Smith-Waterman
APTS	0.38	**0.50**
Boiler	0.00	0.00
Cruiser	**0.62**	0.53
eB2B	0.32	**0.54**
GSM	0.30	0.30
SmartCam	0.40	**0.41**
S.S.MAS	0.25	**1.00**

is 0 for both clustering results. Given that only two elements are clustered, further information is needed for the purpose of providing a more accurate clusterization outcome. However, the manual analysis of such elements shows that their behavior regard two distinct families. Even in this case, the Smith-Waterman algorithm helps the definition of families, as it explicitly shows which specific parts of the CBs are similar, as it was shown in Section 4.3.3.

Finally, it is noticeable in the case studies that the constraints for the Cruiser and SmartCam are created to make sure that a required ordering in the messages is followed. For instance, *camera1* sends *start_search* first to *camera3* in Figure 5.31 and to *camera2* in Figure 5.29, even though *camera1* sends *start_search* to both *camera2* and *camera3* in both constraints. On the other hand, for the other studied systems, the constraints are created based on the overall system behavior that should not happen (e.g., the call is being ended twice in GSM). Therefore, it indicates that an architectural refinement might be better suited than constraints for the Cruiser and SmartCam, as the behaviors might be positive and the different ordering of messages is only due to the concurrency nature of the system. Furthermore, the lack of modularity in the bMSCs (i.e., few bMSCs with lots of messages) makes it difficult for more effective constraints to be created. However, these considerations are out of the scope of this work.

5.3.1 Threats to Validity

The first threat that could be raised in our work may regard its performance. Some might worry that SW might slow down our proposal, since there is a lot of effort in bioinformatics research to speed up the Smith-Waterman algorithm. However, in the bioinformatics domain, the SW algorithm is applied to find local alignments between huge DNA sequences, usually sequences with millions of elements. In the scope of our work, we will unlikely face sequences of such magnitude. In fact, among our case studies the time to run the whole analysis script (which includes SW) did not reach 1s for none of the case studies.

Another drawback that is possible due to SW is that when lengthy common behaviors are compared, they might be found to be similar even though they may not share the same faulty behavior. This happens because they can have a considerable overlap of messages, possibly stemmed from poorly modularized hMSCs that could be split into more cohesive bMSCs. Nevertheless, this is a design issue and is out of scope of our work to deal with such drawback. Further studies could be performed to investigate the core causes of such issue.

Still on SW, as seen in the Boiler system analysis, the best alignment might provide an acceptable behavior, that is, when the given sequences are similar, they do not represent an error. Therefore, it is crucial that a domain expert manually analyze the outputted results.

Regarding the ability to generalize our methodology, it was not meant to deal with I/O implied scenarios, nor use a different communication scheme other than synchronous. For that purpose, it would be possible to use Song et al.'s [5] approach to detect implied scenarios instead of LTSA-MSC, which is used in our approach. In that case, it might require some tuning in the detection of common behaviors and how to find similarity between them.

Finally, LTSA-MSC cannot proceed on the ISs detection process of manually constrained FSP models. To turn around such issue, we have provided means to automatically compare the original architecture model and the constrained one to check if the latter had satisfactorily removed only the traces that could lead to the detected implied scenarios. In all case studies, it had been validated that the valid traces detected by Algorithm 3 had been preserved. We should note that, due to the NP-complete nature of implied scenario analysis, this might not always be possible since a feedback loop in the implied scenario detection process could still reveal the presence of other undetected implied scenarios.

Chapter 6

Conclusion & Future Work

We have proposed a methodology to deal with multiple implied scenarios (ISs) at large. Our approach facilitates the manual analysis required by the user, as he/she has to analyze a considerably smaller number of elements. This is achieved by detecting shared behaviors among the ISs, which we call Common Behaviors (CBs). Furthermore, we introduce a dissimilarity function that allows for the similar CBs to be grouped into families, further reducing the elements to be analyzed. By these means, our approach allows the user to investigate the cause of multiple ISs, which consequently allows he/she to deal with various ISs at once.

Additionally, we performed seven case studies with system specifications found in the literature. Throughout these experiments, we were able to collect 1798 ISs and remove all the ISs with only 14 constraints by using our approach. These results support the effectiveness of the proposed methodology, as it was possible to treat multiple ISs with the same fix. Furthermore, the experiments show that it might be possible to stop the collection of ISs at an earlier point, as after a certain point in all of our experiments, the collected ISs were merely a variation of the ones that had already been collected.

Our goal in this work has been to reach a point where the ISs analysis process can converge to a heuristic to guide the solution space. As our results suggest, there might be a point where new ISs detected are simply a variation of the ones already collected for a system. As a future work, one could use common behaviors to detect the point where no new information is encountered given a certain time threshold. For instance, as our results suggest, 100 ISs would have sufficed, and in every experiment that number was achieved under 10 minutes. Additionally, an extension to our work could provide an alternative way to generate the constraints for the whole family automatically. If that is possible, the stopping point for the detection process could become trivial, as after every CB detected, a new constraint could be generated and composed with the constrained model. Additionally, we should mention that other IS detection approaches found in the literature. Therefore, it would be also desirable to provide means to adapt our methodology to benefit from such approaches, especially the one proposed by Song et al. [5], given its comprehensiveness.

Furthermore, to leverage further analysis in an automated manner could take place by integrating our approach into the LTSA-MSC tool with support to other clustering and alignment algorithms other than those applied in this work. Also, it could be interesting to get the user input on whether each family of common behaviors is wanted or not, as this would eliminate the cost of restricting acceptable behaviors that we disregarded in this work.

Moreover, it would be interesting to check the impact of the clustering results in the manual analysis. In order to observe this impact, the analyses that were done for the case studies could be repeated, but instead of initiating the process with the most similar pair of CBs, the starting point could be randomized. By these means, it would be possible to check if the clustering result is a good reference for the manual analysis. As if better results are obtained by starting with the most similar pair, the clustering results have a significant impact on the manual analysis.

Additionally, we should note that there are many other alignment algorithms in the literature that we did not explore, such as Neighbor-joining [42] and BLAST [43], which could be viable options to find similarity among CBs. Therefore, even though, in our case studies, the Smith-Waterman performed better than the Levenshtein distance, there might be other alignment algorithms that can provide an even better dissimilarity measure than the Smith-Waterman. We believe, however, that local alignment algorithms should obtain better results than global ones, as the goal is to find a subsequence of messages causing the fault.

Particularly, a multiple alignment algorithm, such as Clustal W [44], could tell us the similarity between the identified CBs, instead of a pairwise approach. Finally, we envision that an extension of our approach could be to provide quantitative reliability analysis of the system followed by a sensitivity analysis of the ISs. This way, it would be possible to find which CBs have a higher impact on the system's overall reliability.

Bibliography

[1] Matteucci, Matteo: *A tutorial on clustering algorithms.* http://home.dei.polimi.it/matteucc/Clustering/tutorial_html, 2008. xi, 13

[2] Al-Azzani, Sarah: *Architecture-centric testing for security.* PhD Thesis, University of Birmingham, 2014. `http://etheses.bham.ac.uk/5206/`. xiii, 42, 60, 109, 111, 112, 114, 116

[3] ITU-TS, Recommendation Z.: *120: Message Sequence Chart (MSC) ITU-TS.* Geneva, 1999. 1, 6

[4] Uml, O. M. G.: *2.0 OCL Specification.* OMG Adopted Specification (ptc/03-10-14), 2003. 1, 21

[5] Song, In Gwon, Sang Uk Jeon, Ah Rim Han, and Doo Hwan Bae: *An approach to identifying causes of implied scenarios using unenforceable orders.* Special Section: Best papers from the APSEC, 53(6):666–681, June 2011, ISSN 0950-5849. `http://www.sciencedirect.com/science/article/pii/S0950584910002089`. 1, 2, 21, 22, 26, 27, 90, 92

[6] Uchitel, Sebastian, Jeff Kramer, and Jeff Magee: *Incremental elaboration of scenario-based specifications and behavior models using implied scenarios.* ACM Transactions on Software Engineering and Methodology, 13(1):37–85, January 2004, ISSN 1049331X. `http://portal.acm.org/citation.cfm?doid=1005561.1005563`, visited on 2017-05-30. 1, 2, 5, 6, 7, 8, 9, 11, 12, 20, 21, 24, 26, 30, 42, 43, 49

[7] Muccini, Henry: *Detecting Implied Scenarios Analyzing Non-local Branching Choices.* In Pezzè, Mauro (editor): *Fundamental Approaches to Software Engineering*, pages 372–386. Springer Berlin Heidelberg, 2003, ISBN 978-3-540-36578-5. 1, 2, 21

[8] Uchitel, Sebastian, Jeff Kramer, and Jeff Magee: *Implied Scenario Detection in the Presence of Behaviour Constraints1.* Electronic Notes in Theoretical Computer Science, 65(7):65–84, 2002. 1, 3, 20, 26

[9] Alur, Rajeev, Kousha Etessami, and Mihalis Yannakakis: *Realizability and Verification of MSC Graphs.* In Orejas, Fernando, Paul G. Spirakis, and Jan van Leeuwen (editors): *Automata, Languages and Programming*, pages 797–808. Springer Berlin Heidelberg, 2001, ISBN 978-3-540-48224-6. 1

[10] Rodrigues, Genaina Nunes, David Rosenblum, and Jonas Wolf: *Reliability Analysis of Concurrent Systems Using LTSA.* In *Companion to the proceedings of the 29th*

International Conference on Software Engineering, pages 63–64. IEEE Computer Society, 2007, ISBN 0-7695-2892-9. 1

[11] Roriz, A. V., G. N. Rodrigues, and L. A. Laranjeira: *Analysis of the impact of implied scenarios on the reliability of computational concurrent systems.* In *2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 105–114, Sept 2014. 1

[12] Al-Azzani, Sarah and Rami Bahsoon: *SecArch: Architecture-level Evaluation and Testing for Security.* In *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 51–60. IEEE Computer Society, 2012, ISBN 978-0-7695-4827-2. 2

[13] Chakraborty, Joy, Deepak D'Souza, and K. Narayan Kumar: *Analysing message sequence graph specifications.* In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 549–563. Springer, 2010. 2, 21

[14] R. Alur, K. Etessami, and M. Yannakakis: *Inference of message sequence charts.* IEEE Transactions on Software Engineering, 29(7):623–633, July 2003, ISSN 0098-5589. 2

[15] H. Dan, R. M. Hierons, and S. Counsell: *Non-local Choice and Implied Scenarios.* In *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 53–62, September 2010, ISBN 1551-0255. 2

[16] Moshirpour, Mohammad, Abdolmajid Mousavi, and Behrouz H. Far: *Model based detection of implied scenarios in multi agent systems.* pages 63–68. IEEE, August 2010, ISBN 978-1-4244-8097-5. `http://ieeexplore.ieee.org/document/5558962/`, visited on 2018-05-22. 2, 21, 22, 42, 80

[17] Smith, T. F. and M. S. Waterman: *Identification of common molecular subsequences.* Journal of Molecular Biology, 147(1):195–197, March 1981, ISSN 0022-2836. 3, 17, 29

[18] Levenshtein, Vladimir I: *Binary codes capable of correcting deletions, insertions, and reversals.* In *Soviet physics doklady*, volume 10, pages 707–710, 1966. 3, 88

[19] Magee, Jeff and Jeff Kramer: *Concurrency: state models & Java programs.* Wiley, Chichester, England ; Hoboken, NJ, 2006, ISBN 978-0-470-09355-9. OCLC: ocm64084212. 8, 42, 51

[20] Uchitel, Sebastian, Jeff Kramer, and Jeff Magee: *Detecting implied scenarios in message sequence chart specifications.* SIGSOFT Softw. Eng. Notes, 26(5):74–82, 2001. 11, 20

[21] Ward, Joe H.: *Hierarchical Grouping to Optimize an Objective Function.* Journal of the American Statistical Association, 58(301):236, March 1963, ISSN 01621459. `http://www.jstor.org/stable/2282967?origin=crossref`, visited on 2018-03-07. 14, 16, 30

[22] Johnson, Stephen C.: *Hierarchical clustering schemes.* Psychometrika, 32(3):241–254, September 1967, ISSN 0033-3123, 1860-0980. `http://link.springer.com/10.1007/BF02289588`, visited on 2018-03-07. 14

[23] Rokach, Lior and Oded Maimon: *Clustering methods.* In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005. 14

[24] Kaufman, Leonard and Peter J Rousseeuw: *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009. 14

[25] Cimiano, Philipp, Andreas Hotho, and Steffen Staab: *Comparing Conceptual, Divisive and Agglomerative Clustering for Learning Taxonomies from Text.* pages 435–439. Ramon López de Mántaras and Lorenza Saitta, 2004. 14

[26] Sneath, P. H. A. and Robert R. Sokal: *Numerical taxonomy: the principles and practice of numerical classification.* A Series of books in biology. W. H. Freeman, San Francisco, 1973, ISBN 978-0-7167-0697-7. 15, 16

[27] Alur, Rajeev, Gerard J. Holzmann, and Doron Peled: *An analyzer for message sequence charts.* In Margaria, Tiziana and Bernhard Steffen (editors): *Tools and Algorithms for the Construction and Analysis of Systems*, pages 35–48. Springer Berlin Heidelberg, 1996, ISBN 978-3-540-49874-2. 20

[28] Uchitel, Sebastian, Robert Chatley, Jeff Kramer, and Jeff Magee: *LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios.* In Goos, Gerhard, Juris Hartmanis, Jan van Leeuwen, Hubert Garavel, and John Hatcliff (editors): *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619, pages 597–601. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, ISBN 978-3-540-00898-9 978-3-540-36577-8. `http://link.springer.com/10.1007/3-540-36577-X_44`, visited on 2017-06-08. 20, 24

[29] Uchitel, Sebastian, Jeff Kramer, and Jeff Magee: *Negative scenarios for implied scenario elicitation.* In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 109–118, Charleston, South Carolina, USA, 2002. ACM, ISBN 1-58113-514-9. 20, 21, 27, 31

[30] E. Letier, J. Kramer, J. Magee, and S. Uchitel: *Monitoring and control in scenario-based requirements analysis.* In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 382–391, May 2005, ISBN 0270-5257. 20

[31] Castejon, Humberto Nicolas and Rolv Braek: *A collaboration-based approach to service specification and detection of implied scenarios.* In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 37–43, Shanghai, China, 2006. ACM, ISBN 1-59593-394-8. 21

[32] Moshirpour, Mohammad, Abdolmajid Mousavi, and Behrouz H Far: *Detecting emergent behavior in distributed systems using scenario-based specifications.* International Journal of Software Engineering and Knowledge Engineering, 22(06):729–746, 2012. 21, 22

[33] F. H. Fard and B. H. Far: *Detection of implied scenarios in multiagent systems with clustering agents' communications.* In *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*, pages 237–244, August 2014. 22

[34] Reis, Thiago P.: *Uma Abordagem para o Uso de Cenários Implícitos na Geração de Casos de Teste para Sistemas Concorrentes.* Dissertação (Mestrado), Universidade de Brasília, 2015. 22, 60

[35] Lamport, Leslie: *Time, clocks, and the ordering of events in a distributed system.* Commun. ACM, 21(7):558–565, July 1978, ISSN 0001-0782. `http://doi.acm.org/10.1145/359545.359563`. 30

[36] Jones, Neil C, Pavel A Pevzner, and Pavel Pevzner: *An introduction to bioinformatics algorithms.* MIT press, 2004. 30

[37] Fitch, Walter M and Temple F Smith: *Optimal sequence alignments.* Proceedings of the National Academy of Sciences, 80(5):1382–1386, 1983. 30

[38] Esterle, Lukas, Peter R Lewis, Xin Yao, and Bernhard Rinner: *Socio-economic vision graph generation and handover in distributed smart camera networks.* ACM Transactions on Sensor Networks (TOSN), 10(2):20, 2014. 60

[39] Chatley, Robert, Jeff Kramer, Jeff Magee, and Sebastian Uchitel: *Model-based Simulation of Web Applications for Usability Assessment.* In *Proceedings of ICSE 2003 Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction, May 3-4, 2003, Portland, Oregon, USA*, pages 5–11, 2003. `http://chatley.com/articles/icse03.pdf`. 67

[40] Leue, Stefan, Lars Mehrmann, and Mohammad Rezai: *Synthesizing ROOM Models from Message Sequence Chart Specifications.* Technical Report 98-06, 1998. 73, 75

[41] Jaccard, Paul: *Nouvelles recherches sur la distribution florale.* Bull. Soc. Vaud. Sci. Nat., 44:223–270, 1908. 88

[42] Saitou, N and M Nei: *The neighbor-joining method: a new method for reconstructing phylogenetic trees.* Molecular Biology and Evolution, 4(4):406–425, 1987. `http://dx.doi.org/10.1093/oxfordjournals.molbev.a040454`. 92

[43] Altschul, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215(3):403 – 410, 1990, ISSN 0022-2836. `http://www.sciencedirect.com/science/article/pii/S0022283605803602`. 92

[44] Larkin, Mark A, Gordon Blackshields, NP Brown, R Chenna, Paul A McGettigan, Hamish McWilliam, Franck Valentin, Iain M Wallace, Andreas Wilm, Rodrigo Lopez, *et al.*: *Clustal w and clustal x version 2.0.* bioinformatics, 23(21):2947–2948, 2007. 92

Appendix A

Systems Scenarios

A.1 A Passenger Transportation System (APTS)

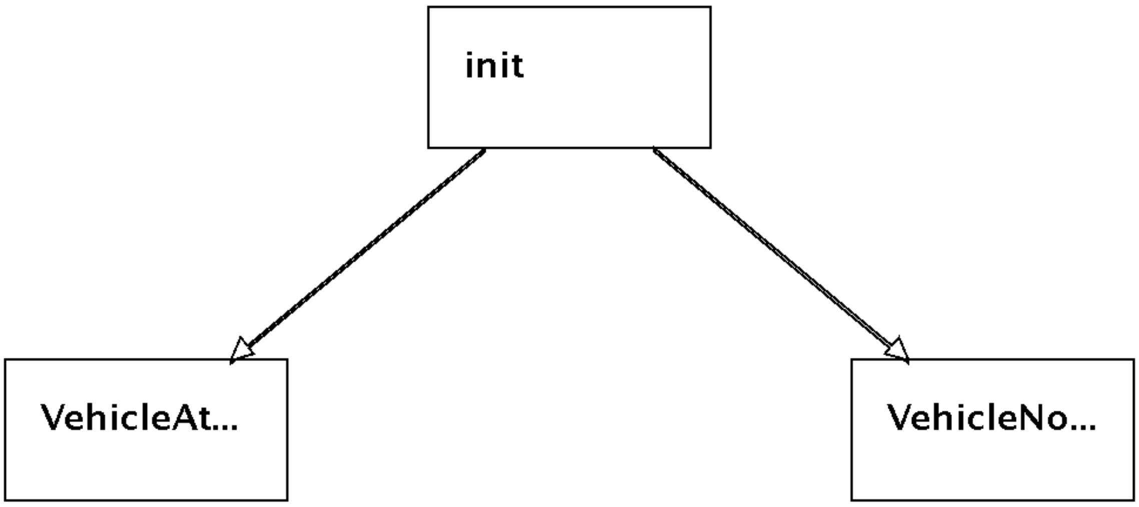


Figure A.1: APTS's hMSC.

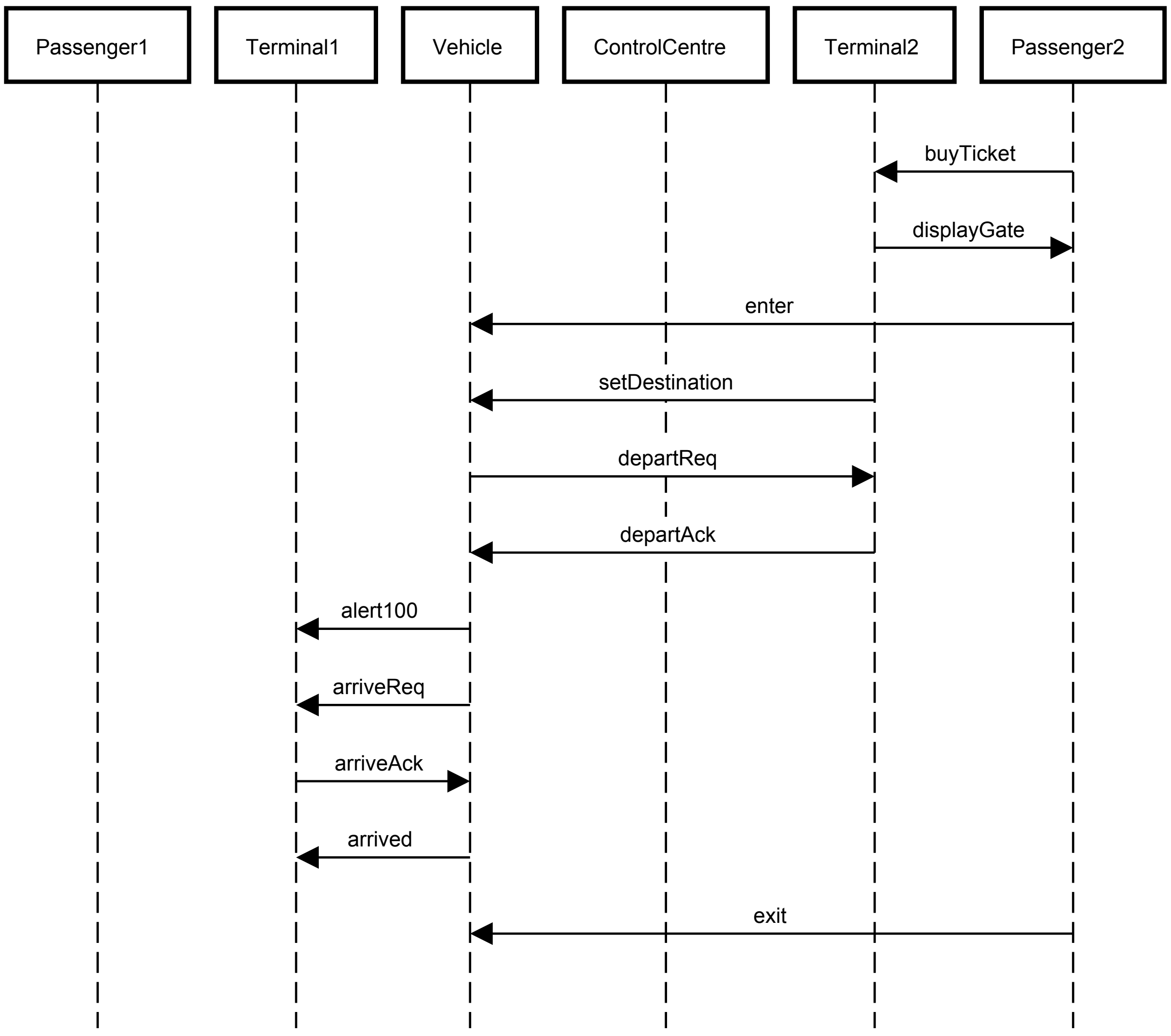


Figure A.2: bMSC for APTS's VehicleAtTerminal scenario.

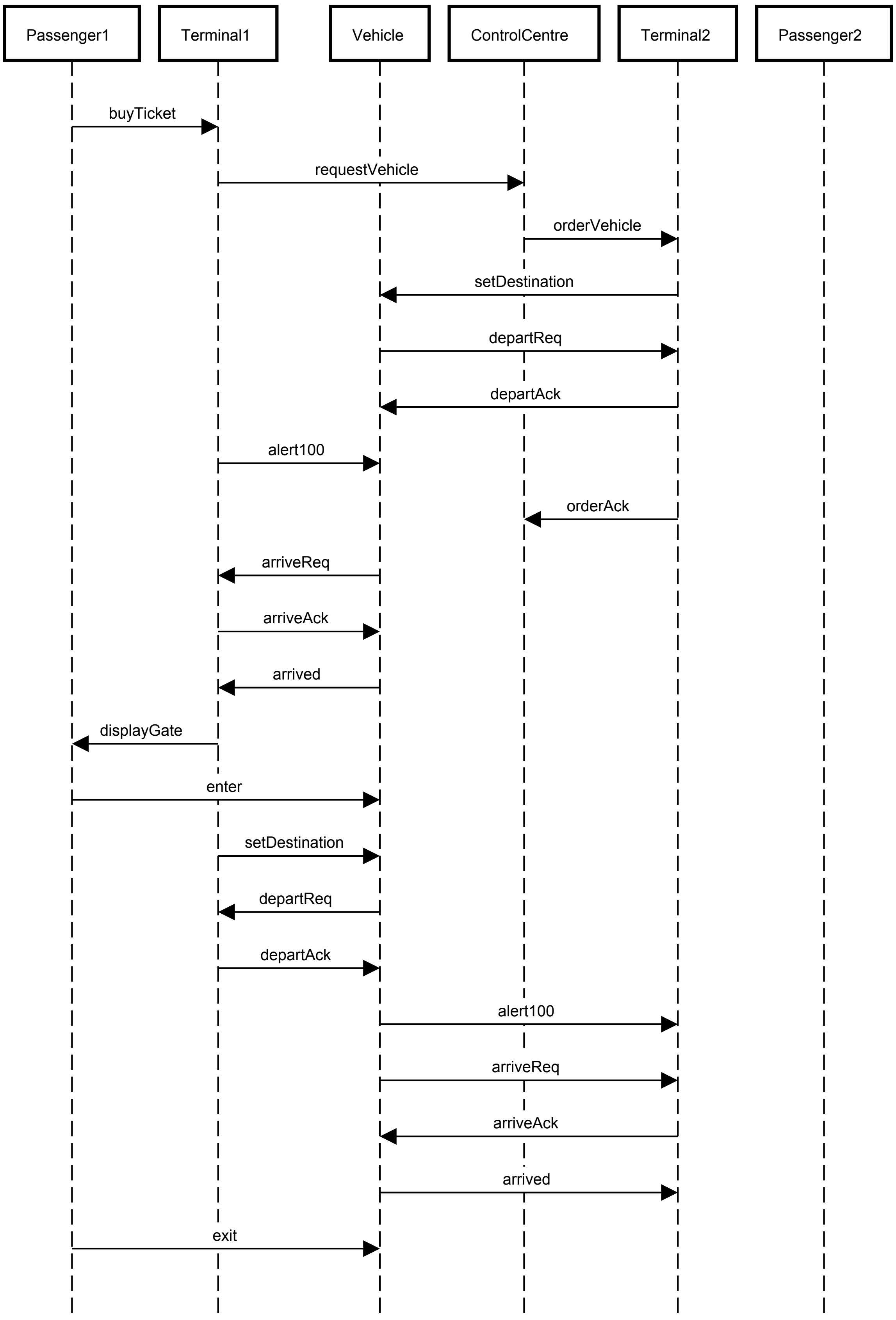


Figure A.3: bMSC for APTS's VehicleNotAtTerminal scenario.

A.2 Boiler

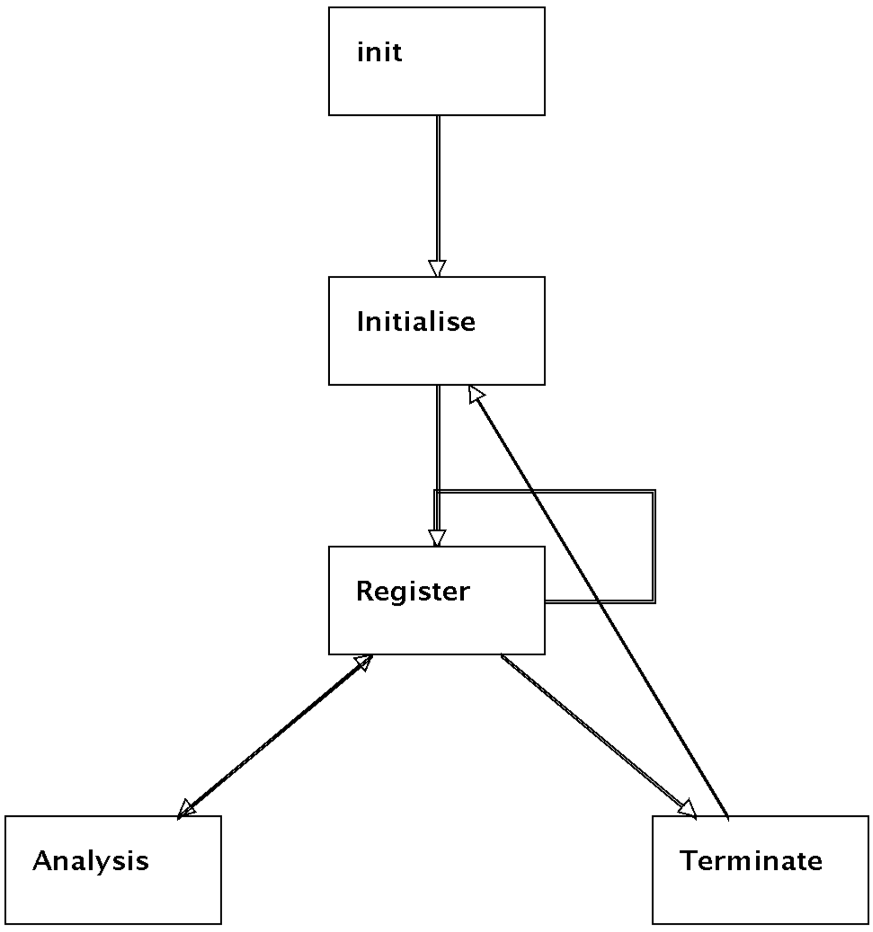


Figure A.4: Boiler's hMSC.

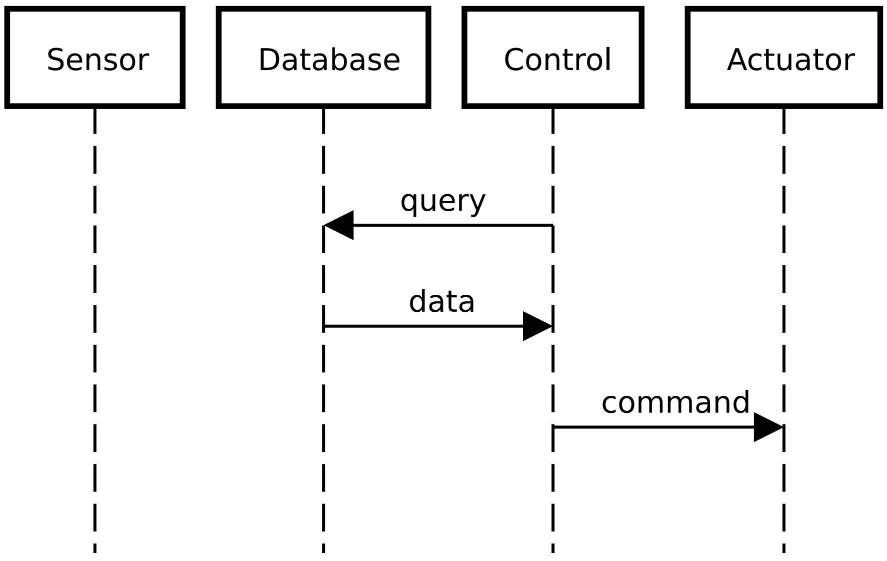


Figure A.5: bMSC for Boiler's Analysis scenario.

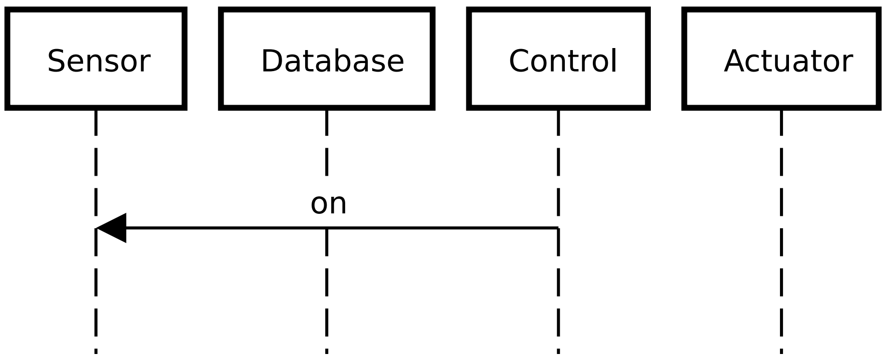


Figure A.6: bMSC for Boiler's Initialise scenario.

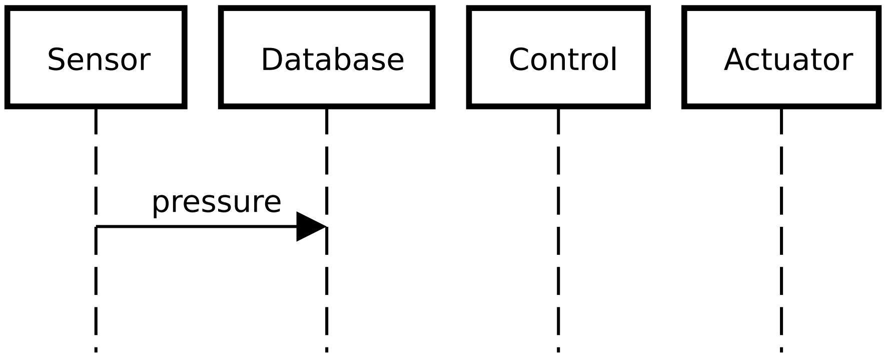


Figure A.7: bMSC for Boiler's Register scenario.

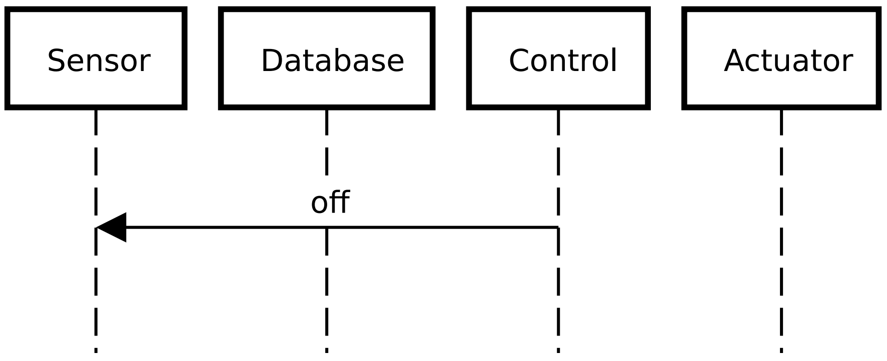


Figure A.8: bMSC for Boiler's Terminate scenario.

A.3 Cruise Control (Cruiser)

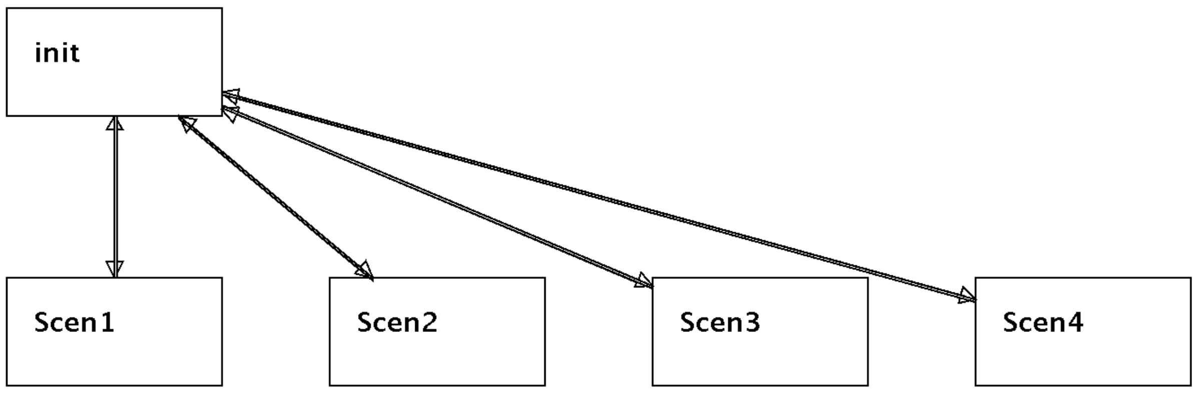


Figure A.9: Cruiser's hMSC.

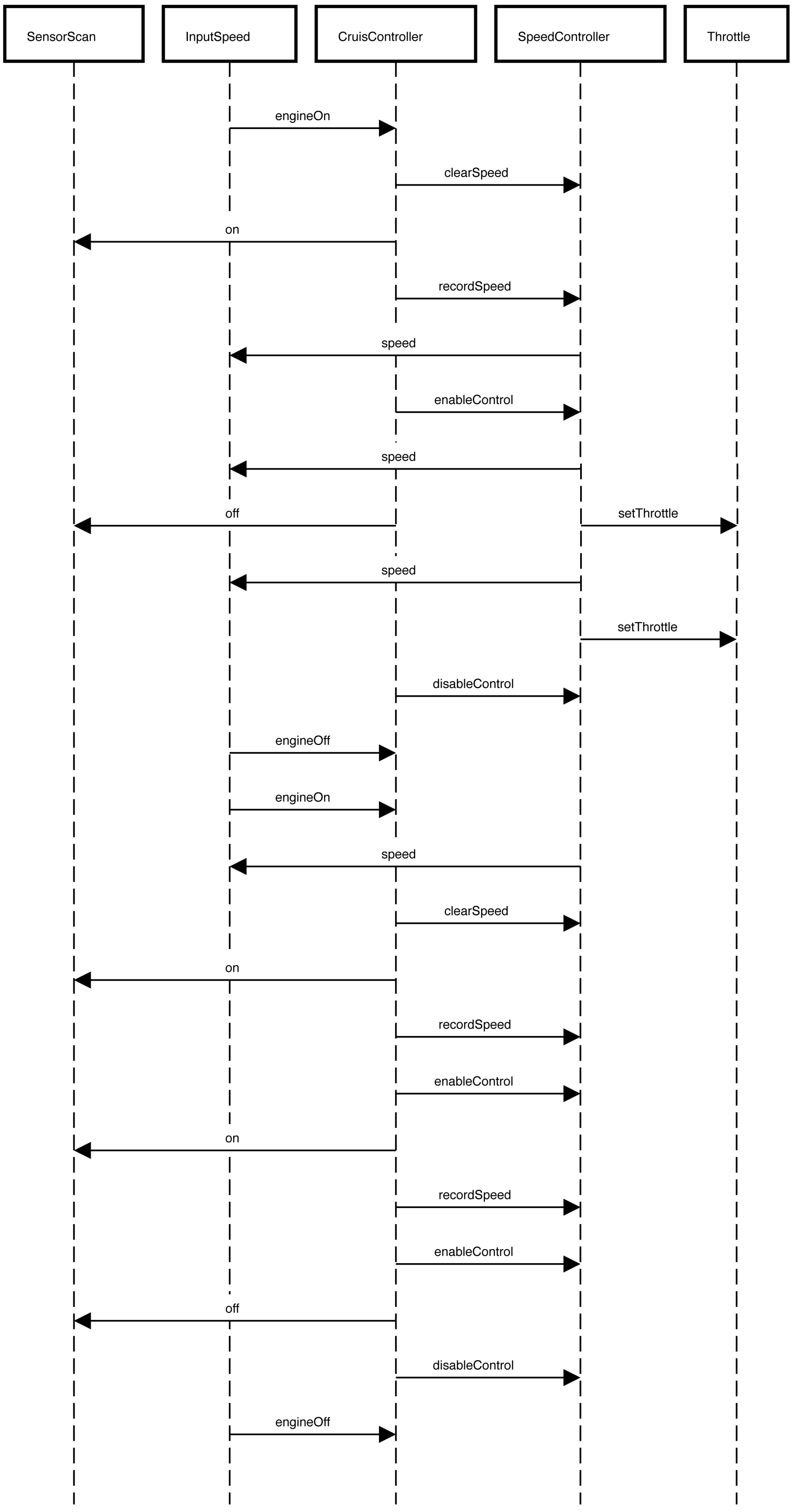


Figure A.10: bMSC for Cruiser's Scen1.

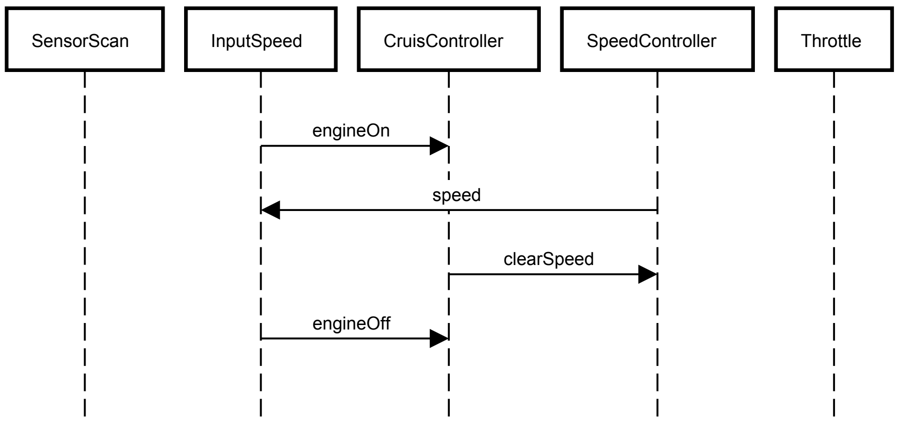


Figure A.11: bMSC for Cruiser's Scen2.

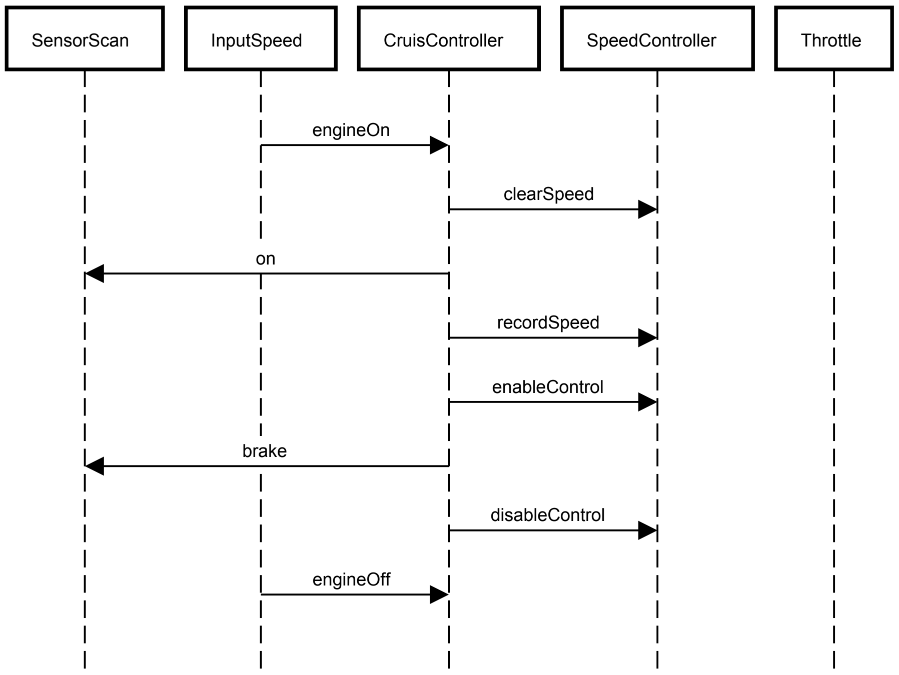


Figure A.12: bMSC for Cruiser's Scen3.

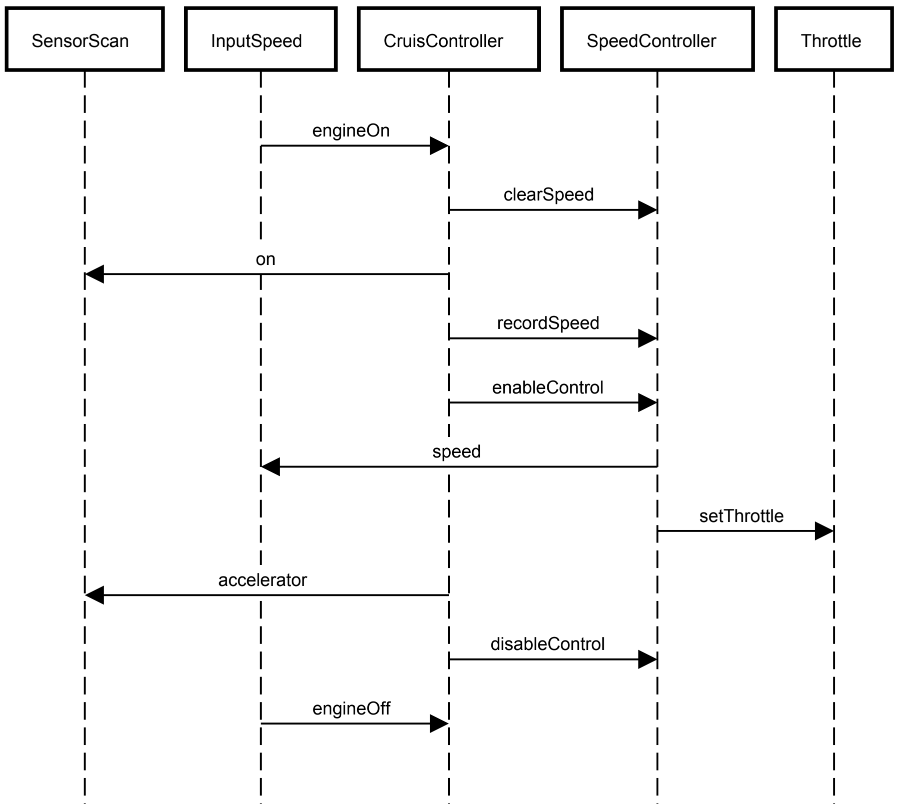


Figure A.13: bMSC for Cruiser's Scen4.

A.4 Distributed Smart Camera (SmartCam)

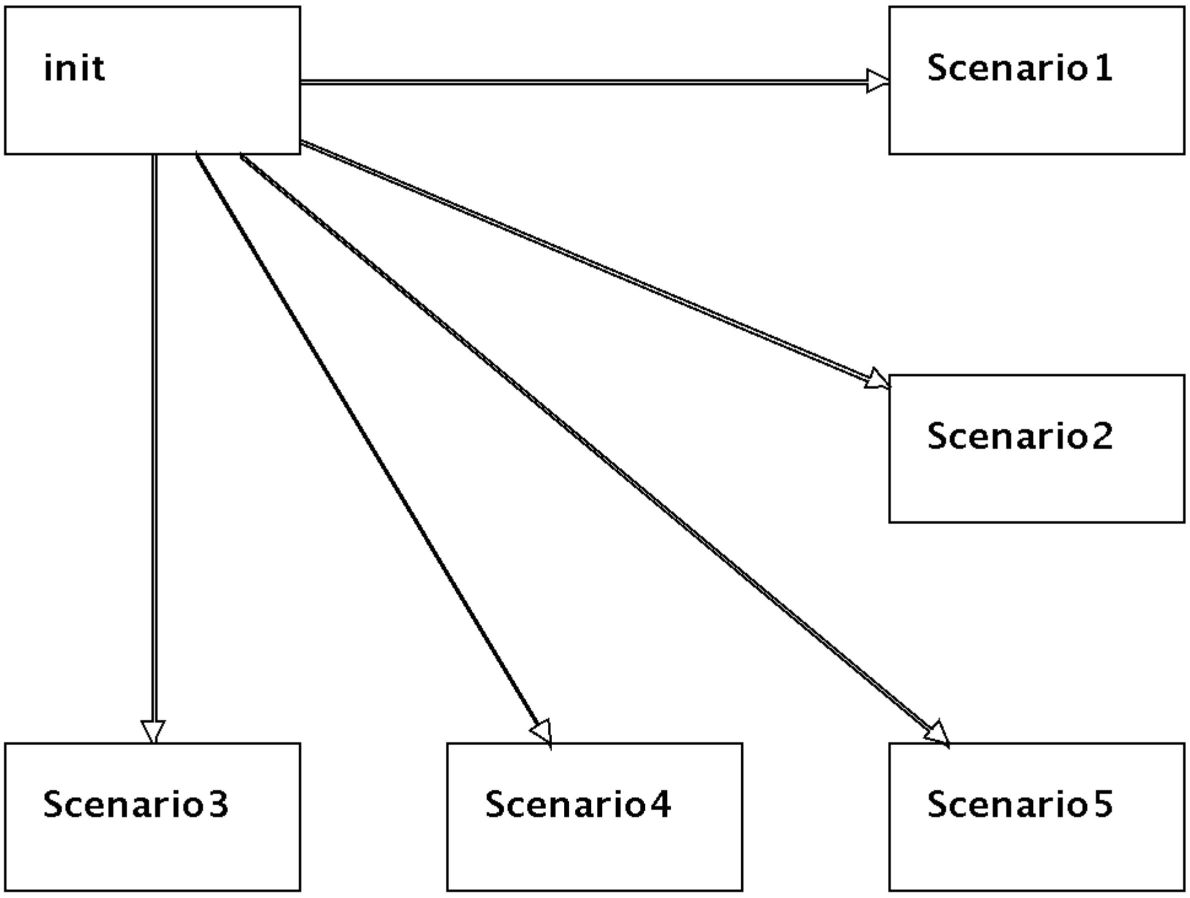


Figure A.14: SmartCam's hMSC.

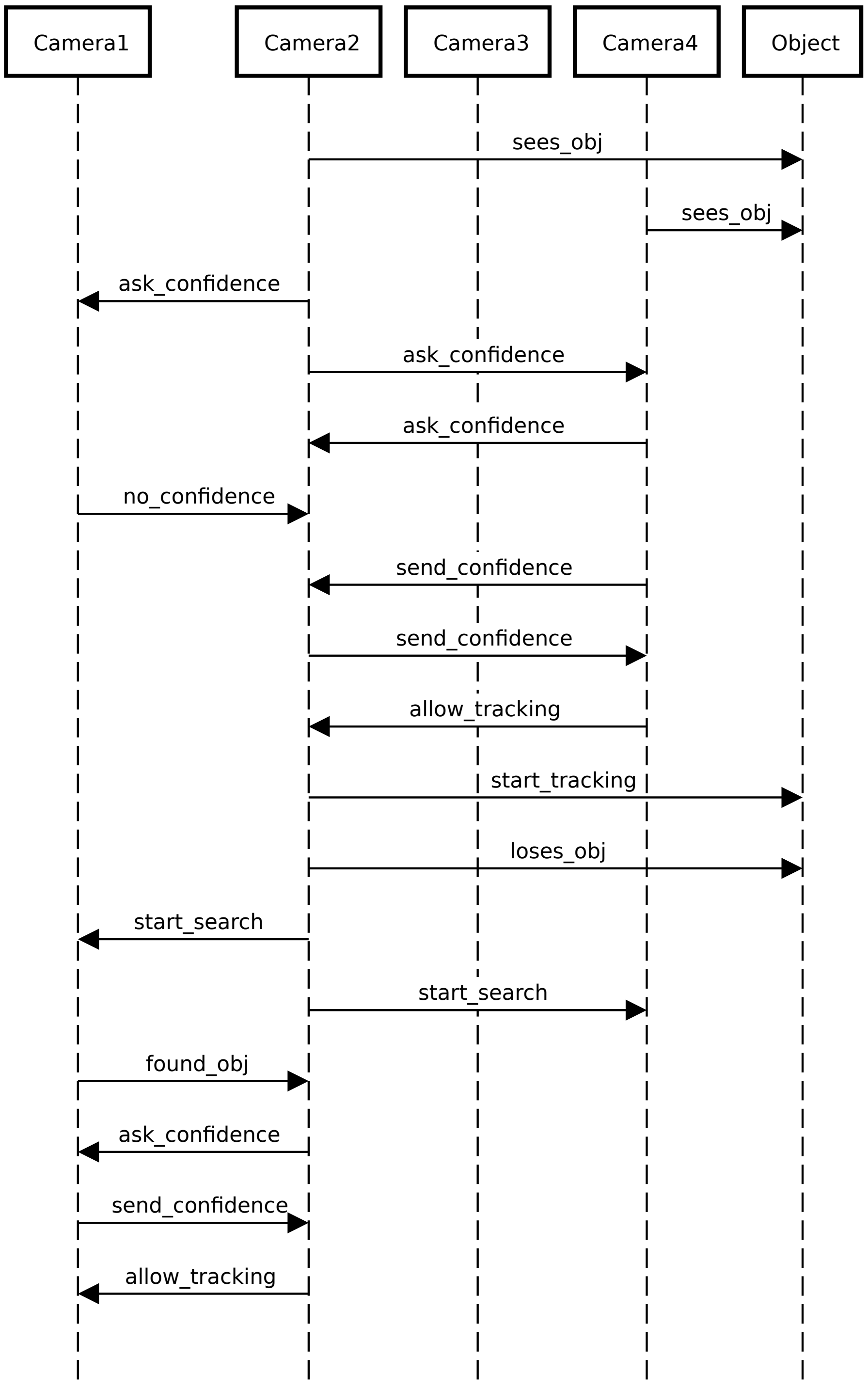


Figure A.15: bMSC for SmartCam's Scenario1.

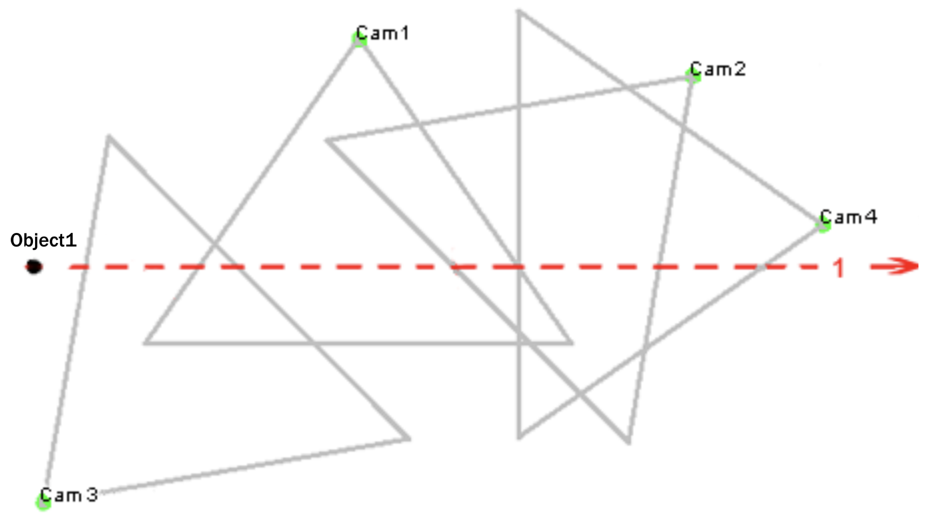


Figure A.16: Trajectory depicted in SmartCam's Scenario1, taken from [2].

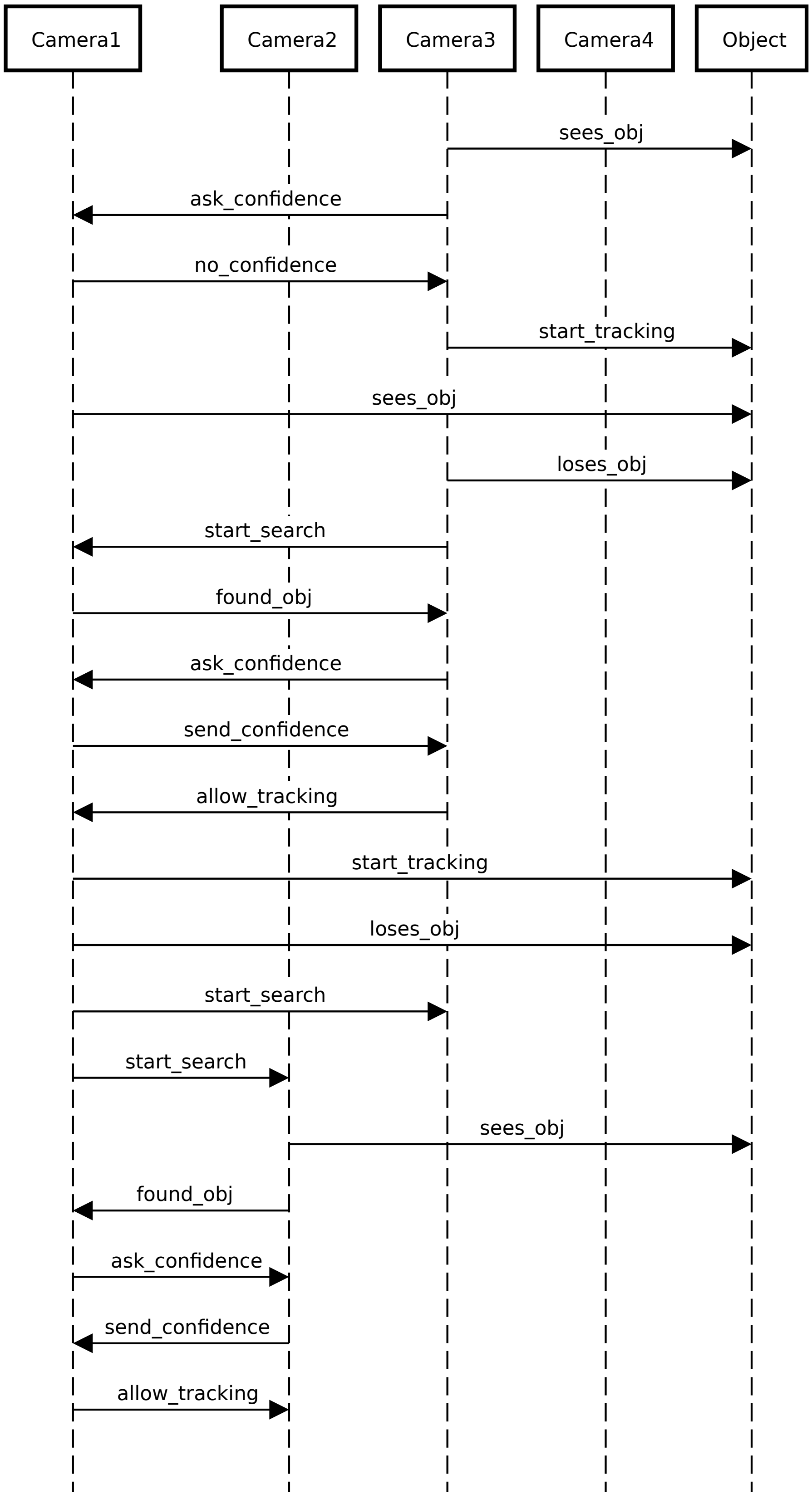


Figure A.17: bMSC for SmartCam's Scenario2.

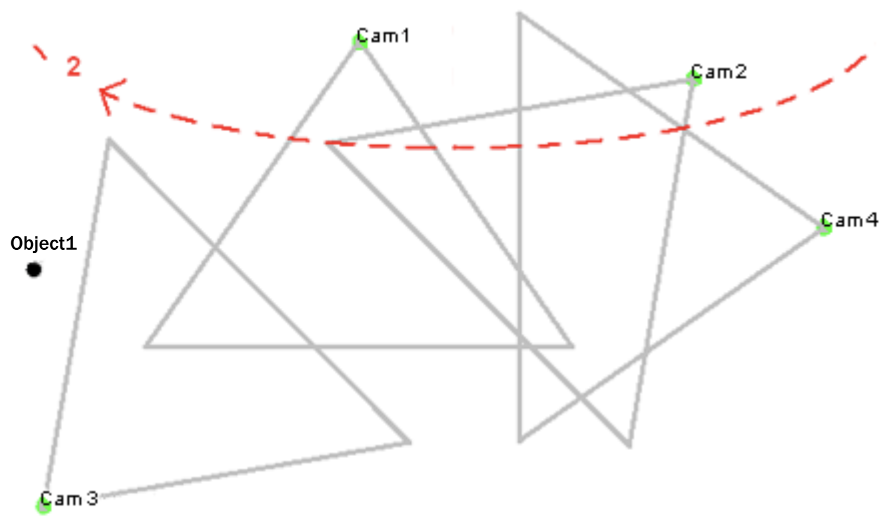


Figure A.18: Trajectory depicted in SmartCam's Scenario2, taken from [2].

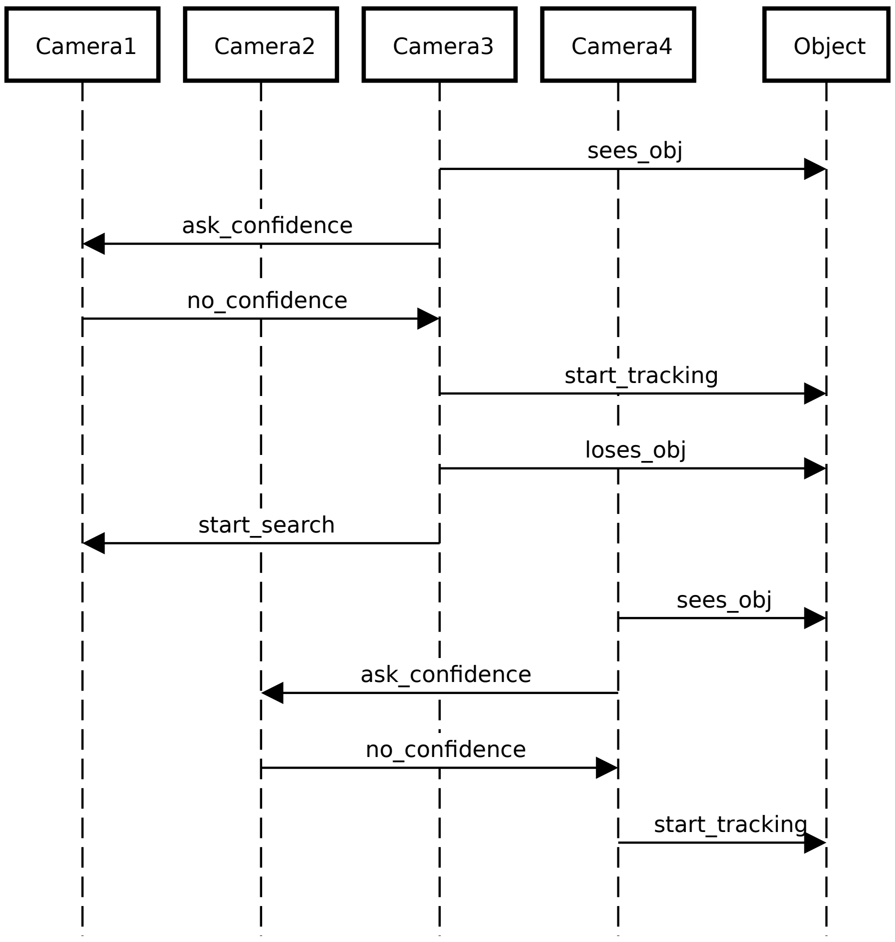


Figure A.19: bMSC for SmartCam's Scenario3.

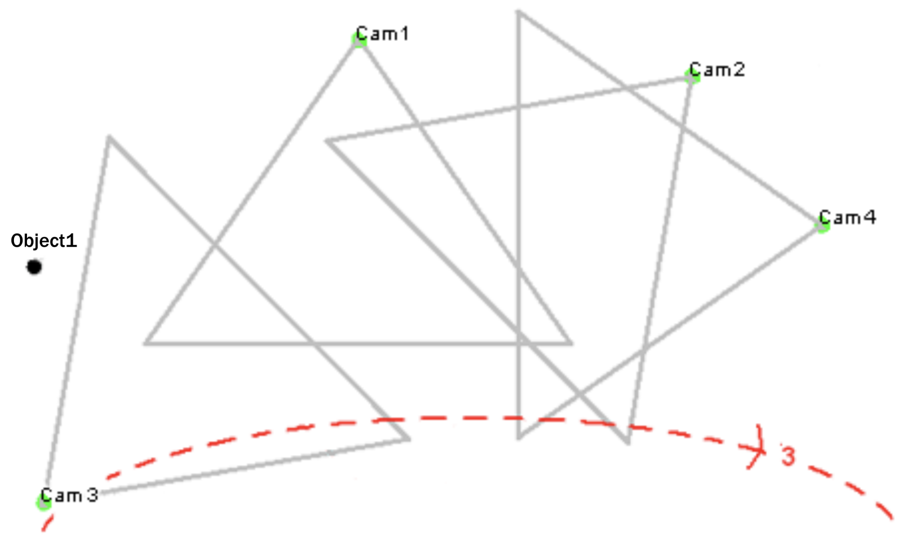


Figure A.20: Trajectory depicted in SmartCam's Scenario3, taken from [2].

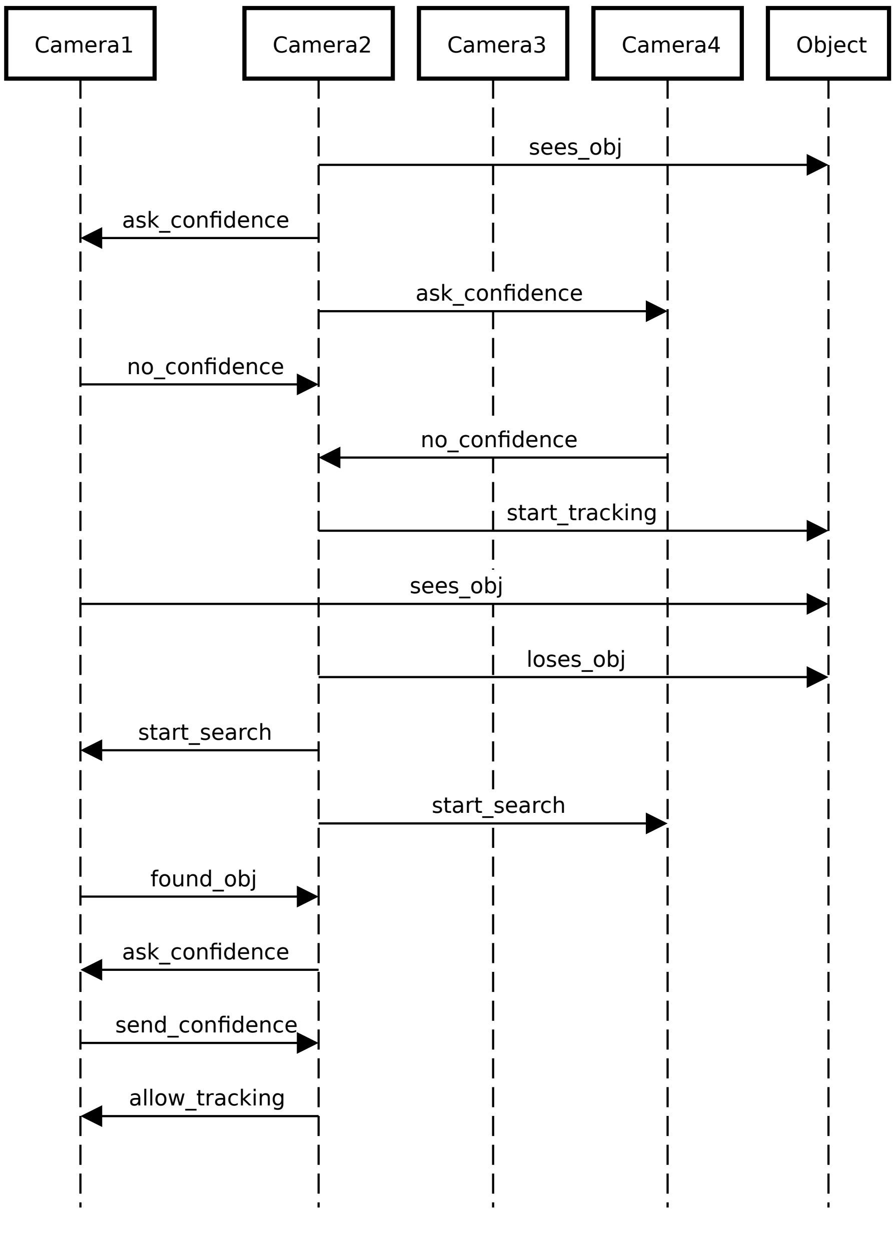


Figure A.21: bMSC for SmartCam's Scenario4.

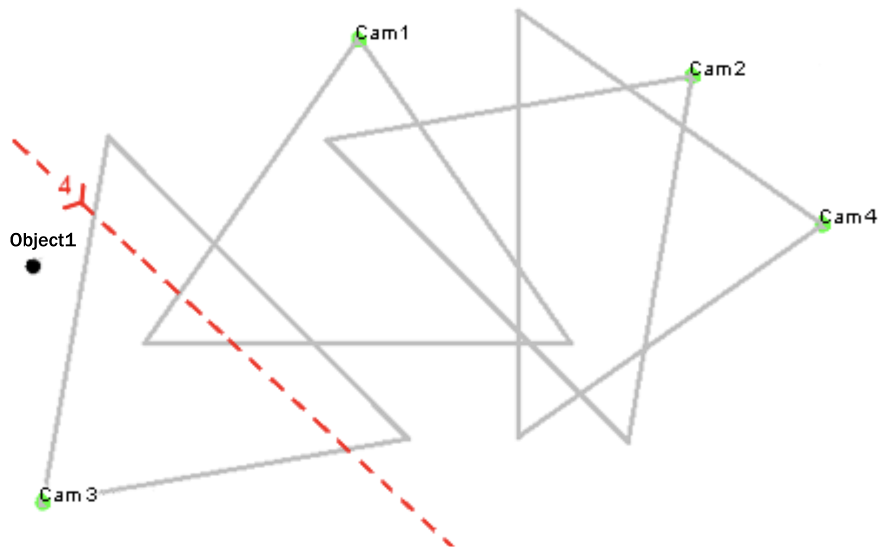


Figure A.22: Trajectory depicted in SmartCam's Scenario4, taken from [2].

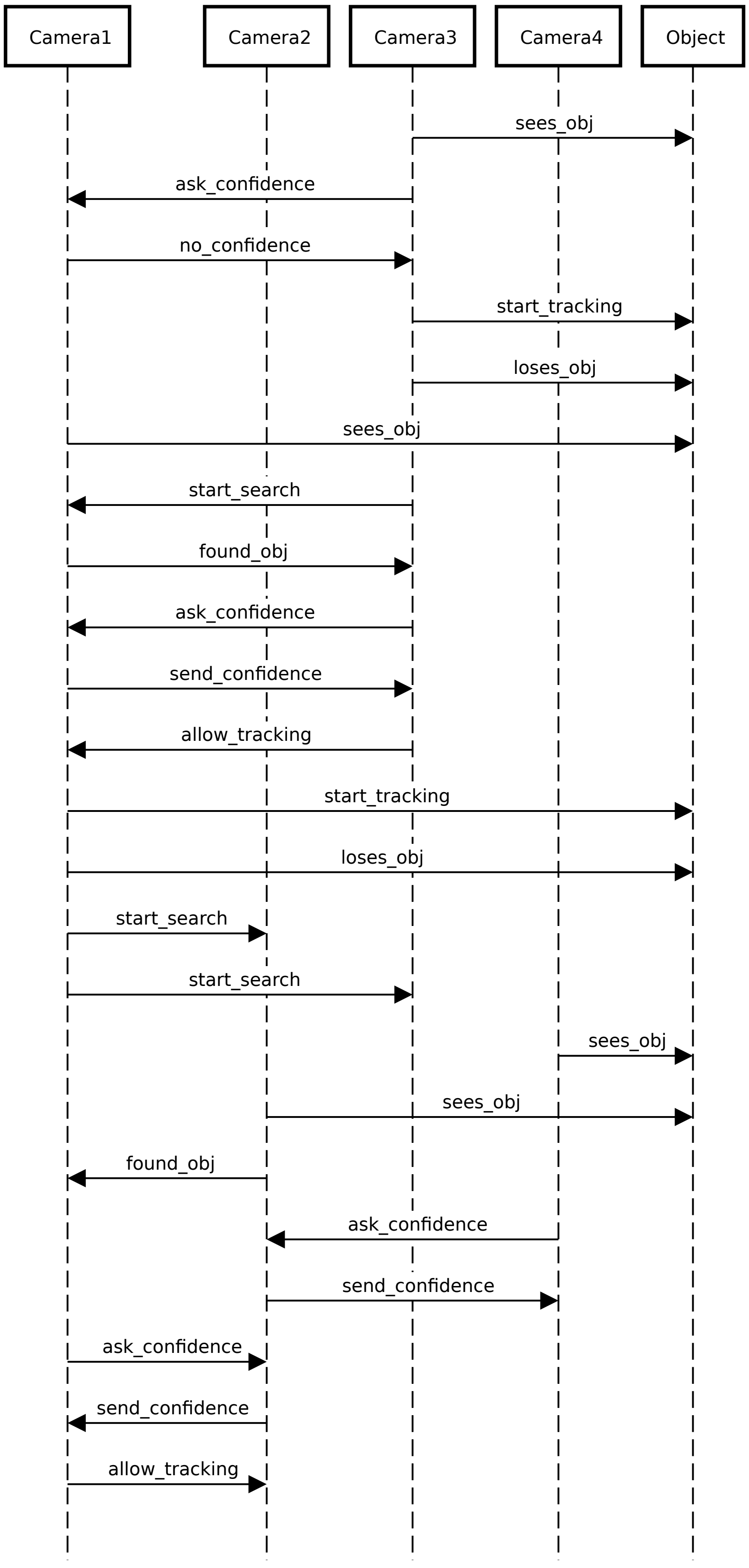


Figure A.23: bMSC for SmartCam's Scenario5.

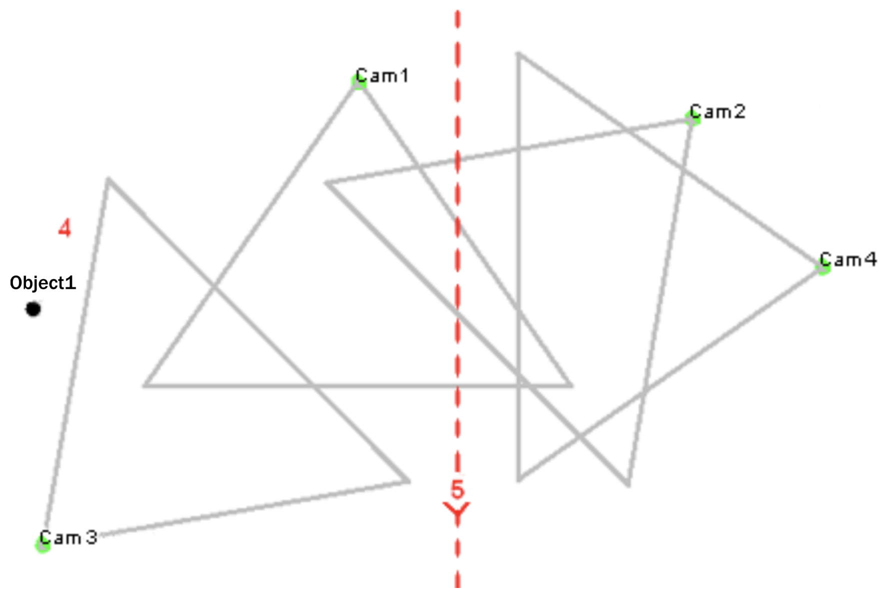


Figure A.24: Trajectory depicted in SmartCam's Scenario5, taken from [2].

A.5 eB2B

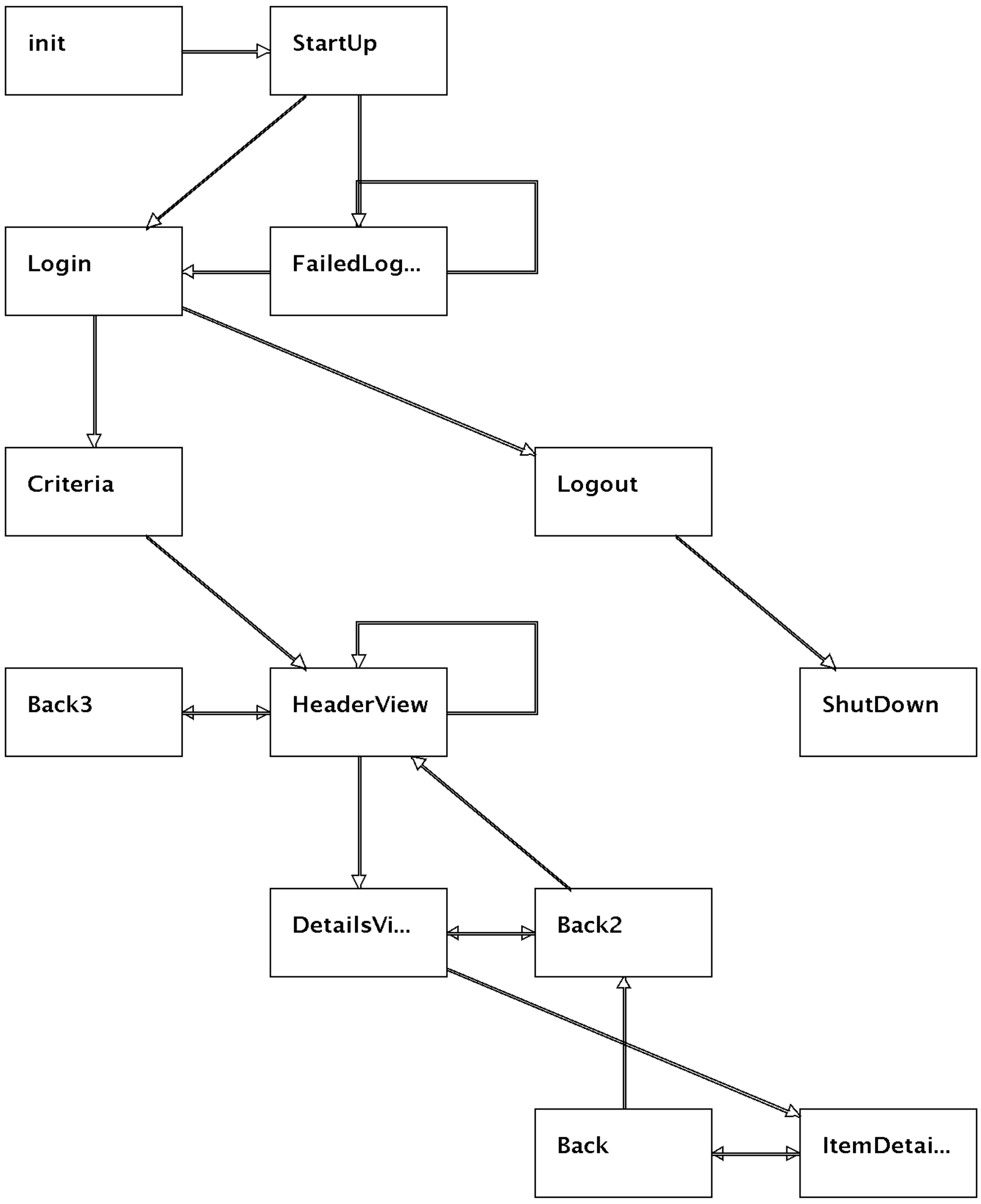


Figure A.25: eB2B's hMSC.

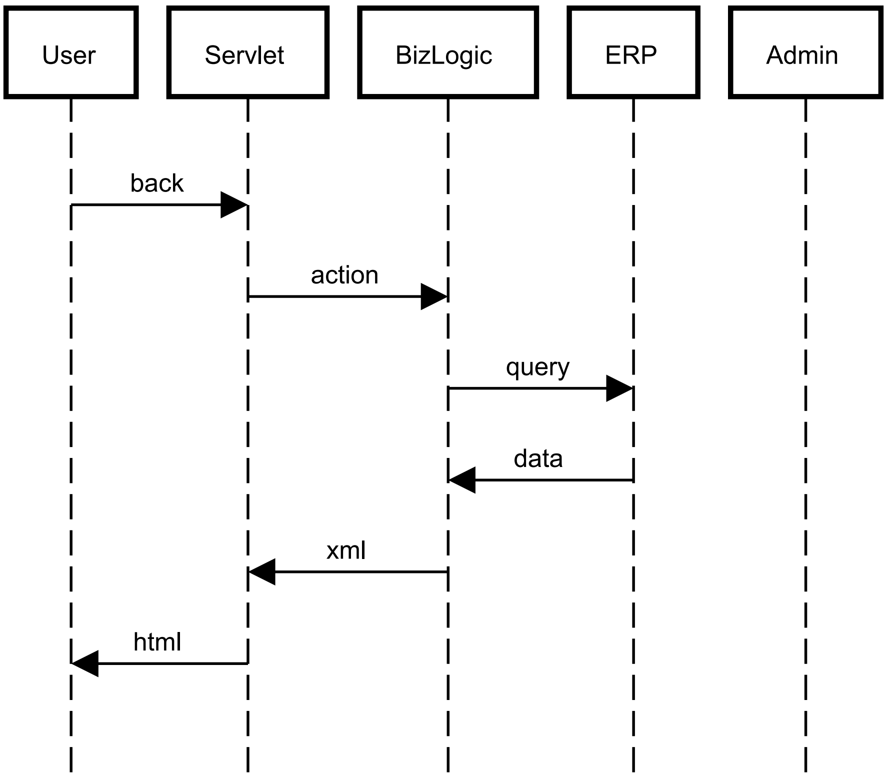


Figure A.26: bMSC for eB2B's Back1, Back2, and Back3 scenarios.

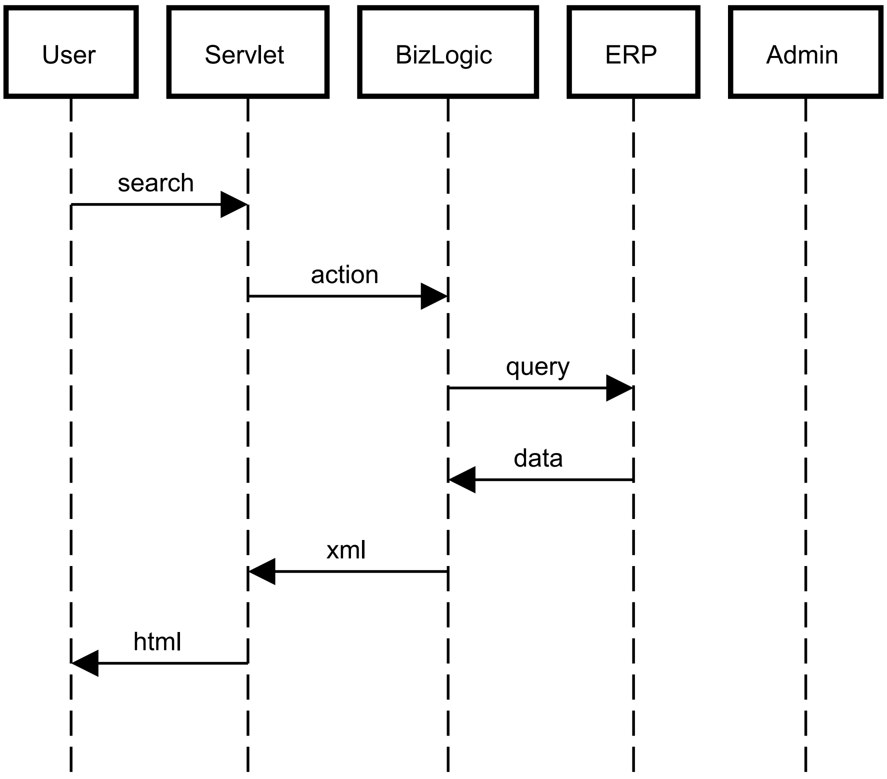


Figure A.27: bMSC for eB2B's Criteria scenario.

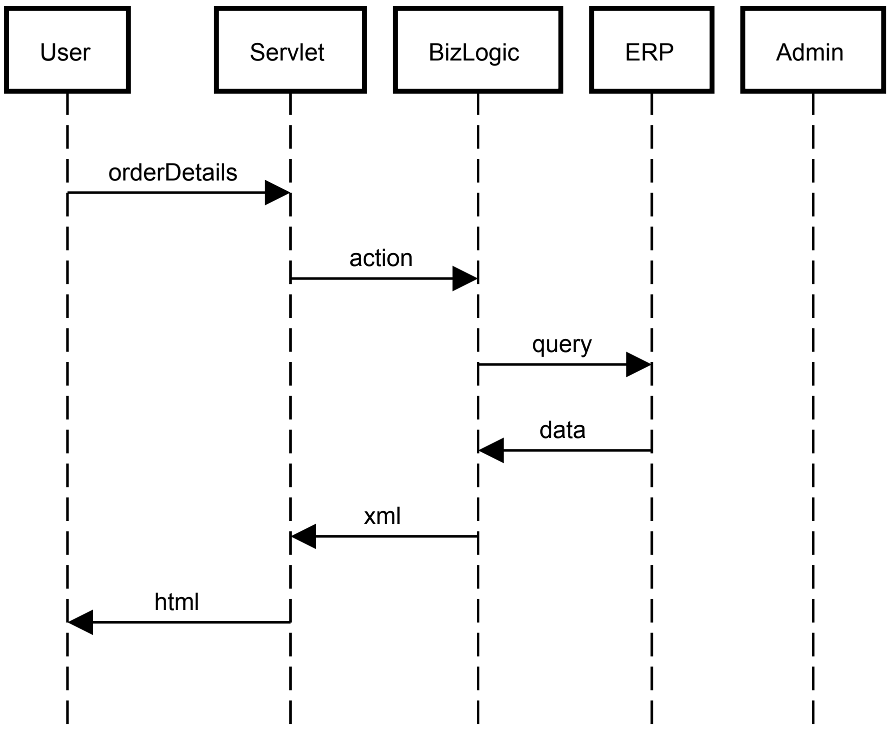


Figure A.28: bMSC for eB2B's DetailsView scenario.

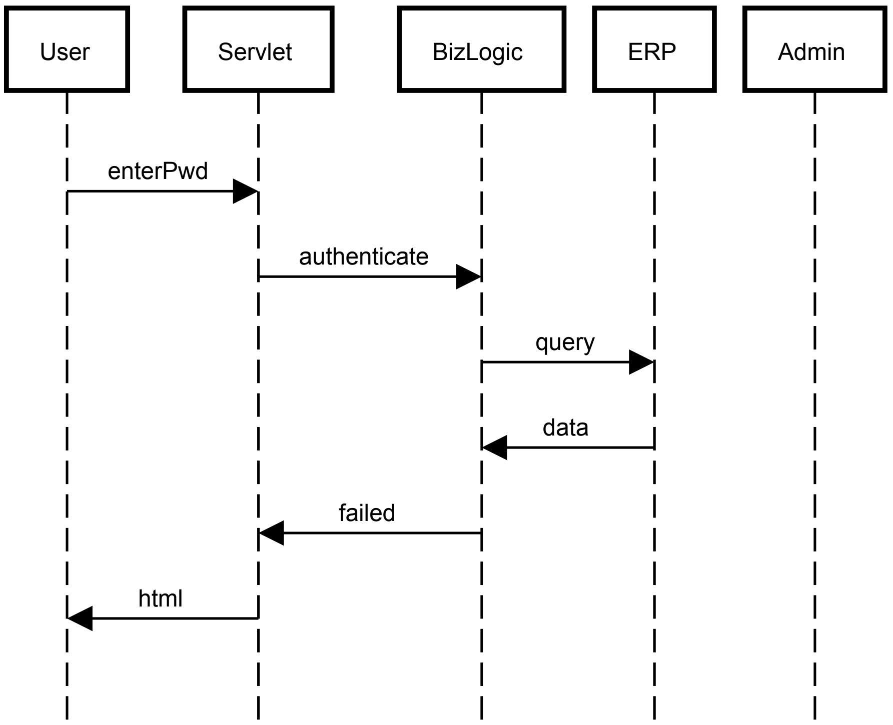


Figure A.29: bMSC for eB2B's FailedLogin scenario.

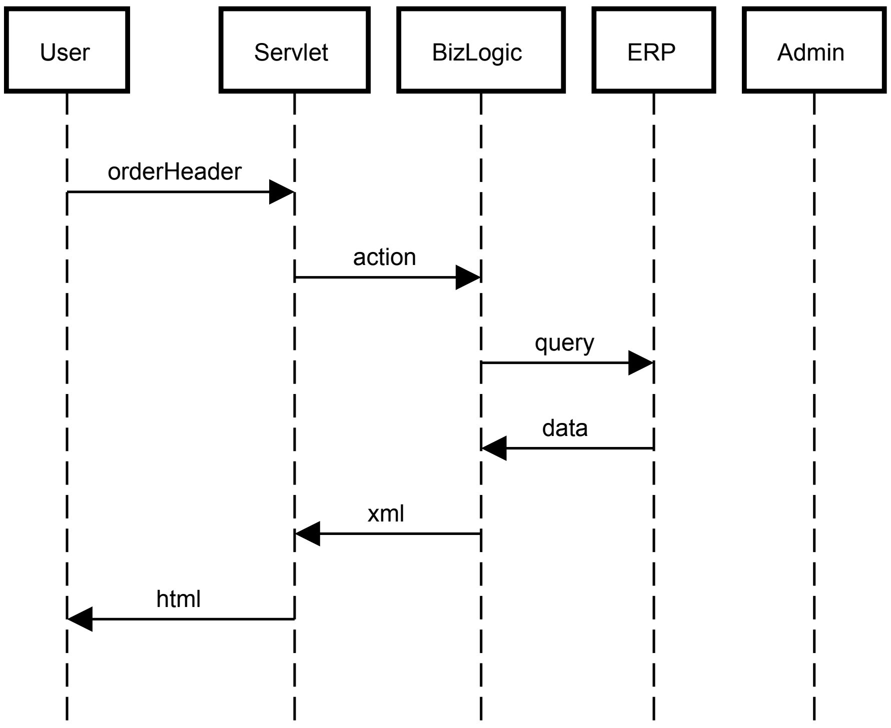


Figure A.30: bMSC for eB2B's HeaderView scenario.

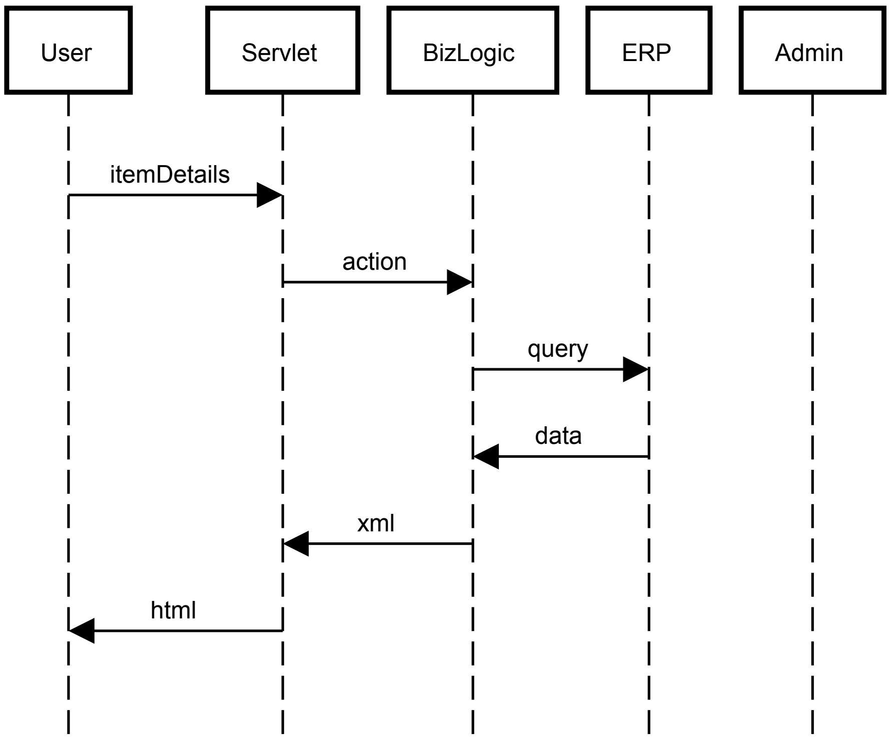


Figure A.31: bMSC for eB2B's ItemDetails scenario.

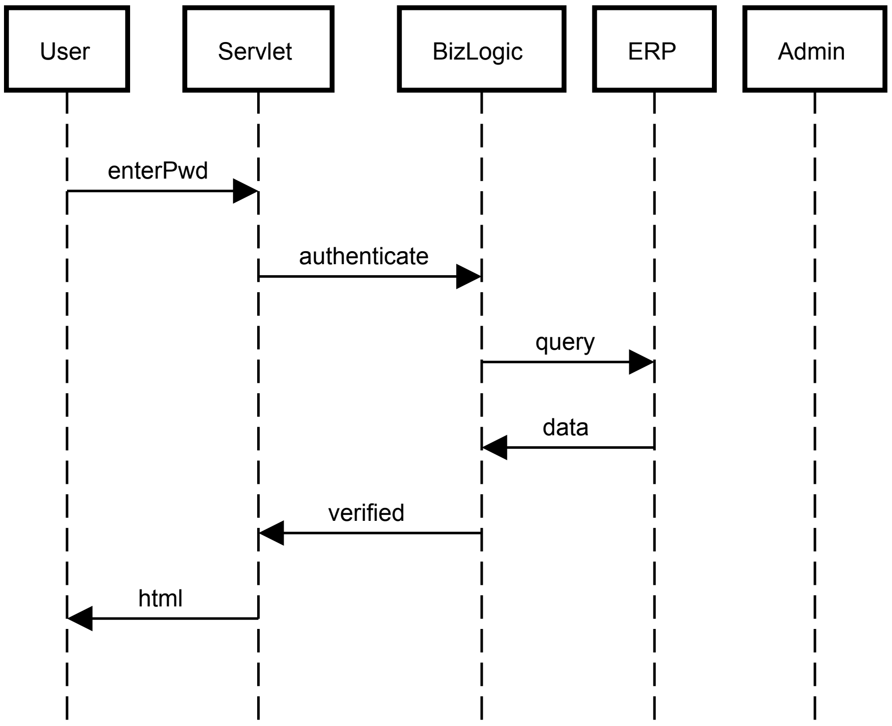


Figure A.32: bMSC for eB2B's Login scenario.

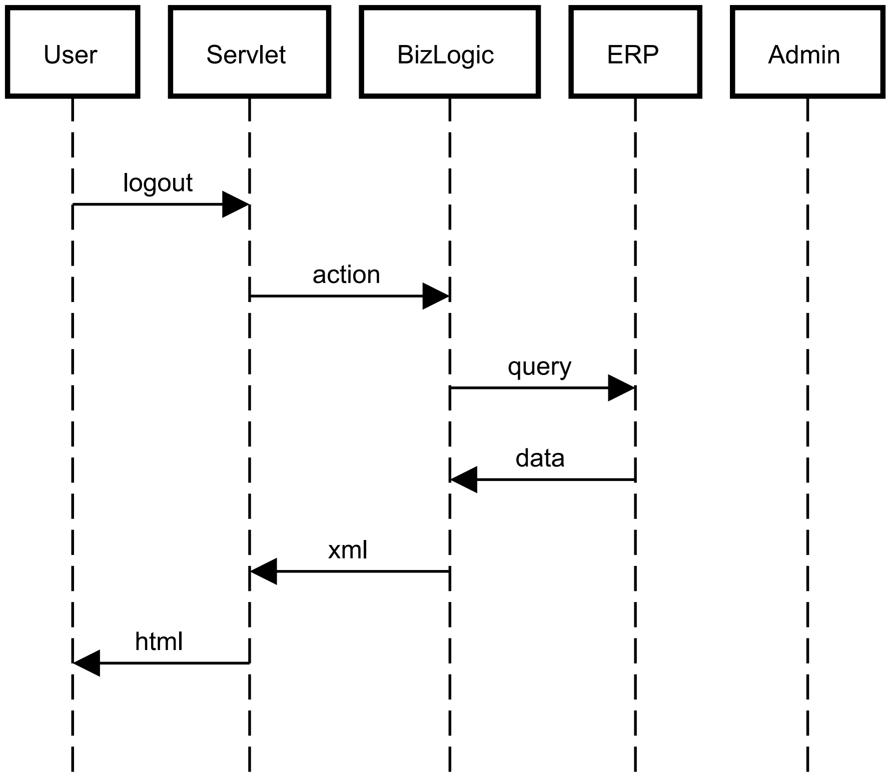


Figure A.33: bMSC for eB2B's Logout scenario.

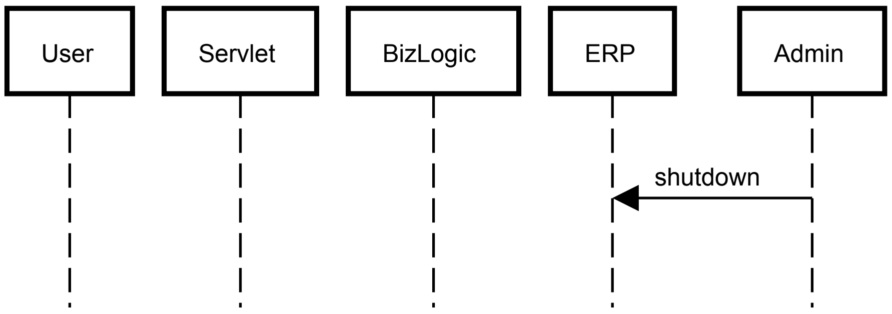


Figure A.34: bMSC for eB2B's ShutDown scenario.

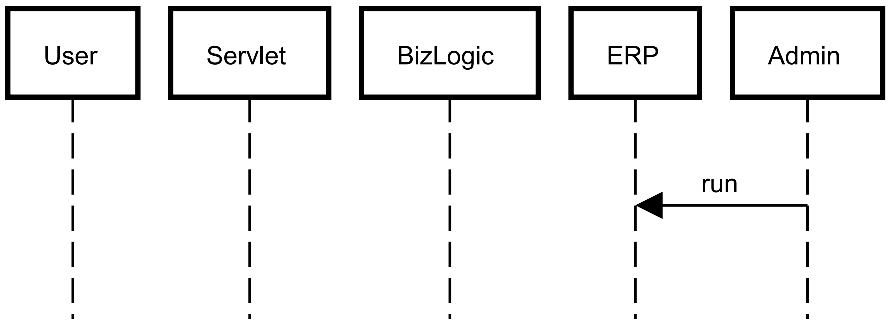


Figure A.35: bMSC for eB2B's StartUp scenario.

A.6 Global System for Mobile Mobility Management (GSM)

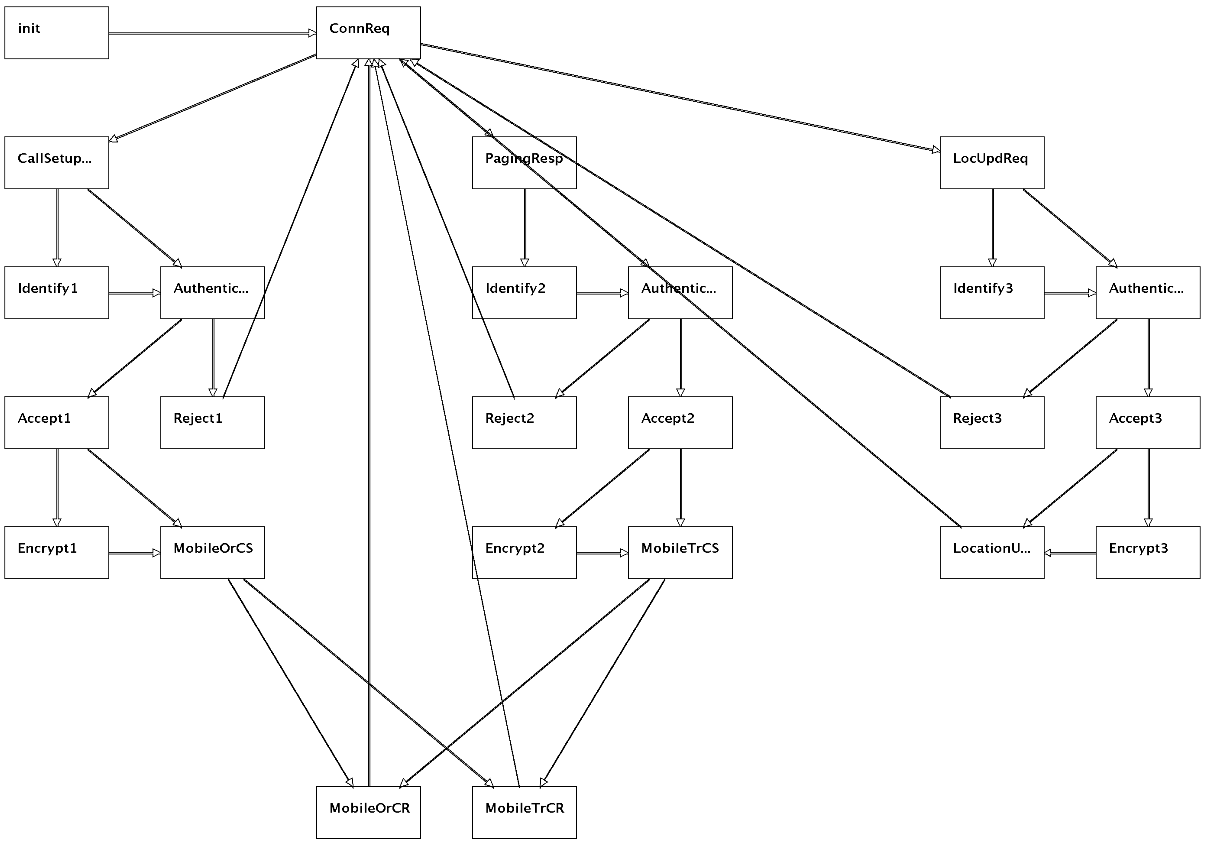


Figure A.36: GSM's hMSC.

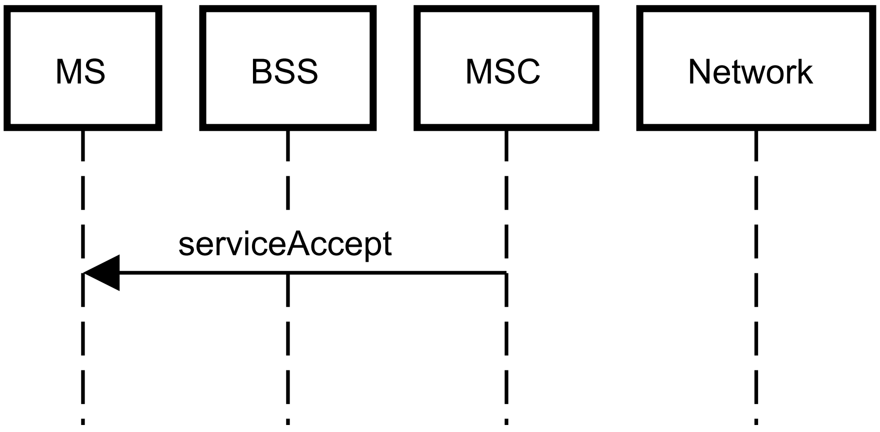


Figure A.37: bMSC for GSM's Accept1, Accept2, Accept3 scenarios.

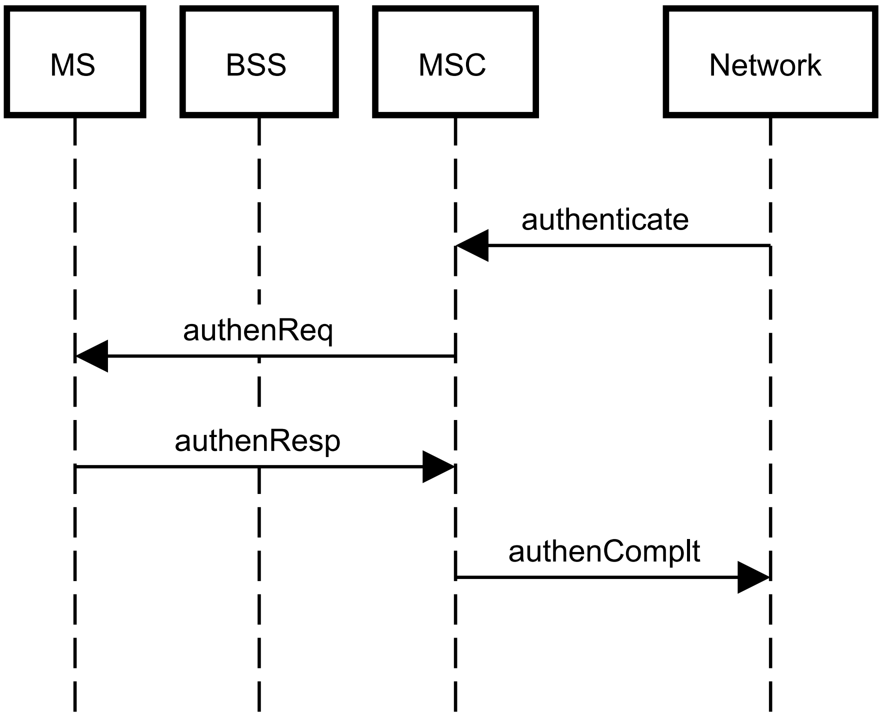


Figure A.38: bMSC for GSM's Authenticate1, Authenticate2, and Authenticate3 scenarios.

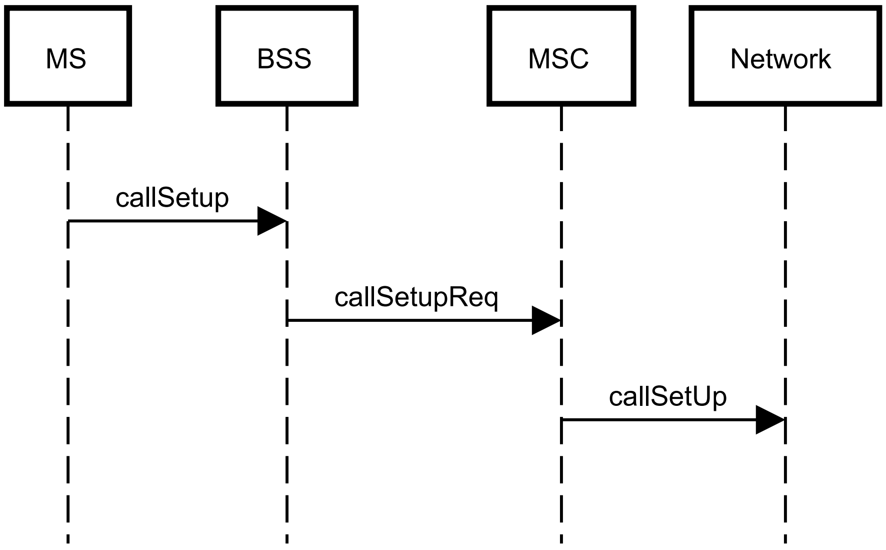


Figure A.39: bMSC for GSM's CallSetupReq scenario.

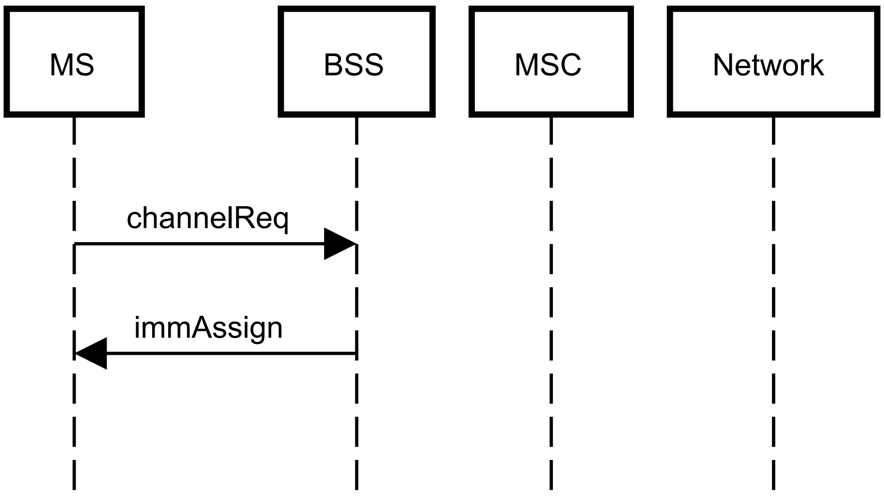


Figure A.40: bMSC for GSM's ConnReq scenario.

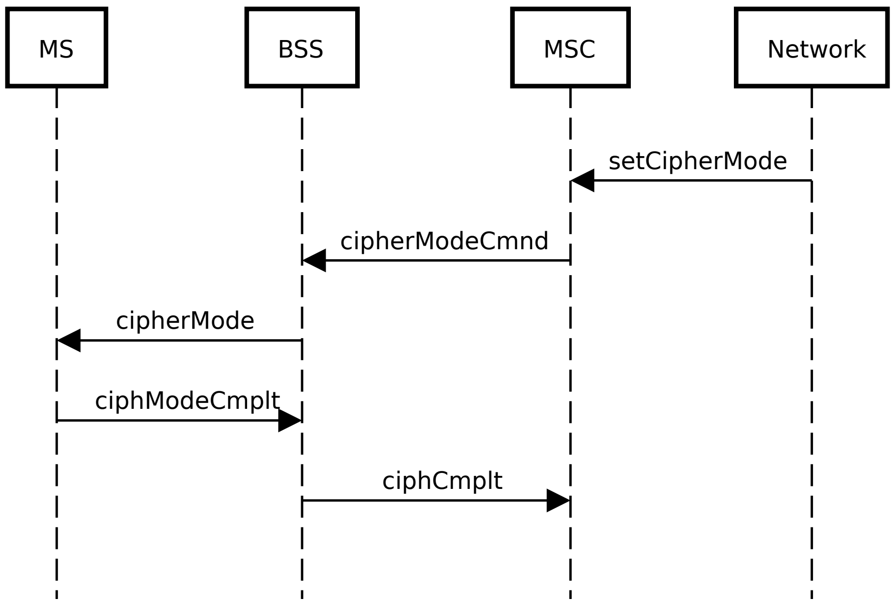


Figure A.41: bMSC for GSM's Encrypt1, Encrypt2, and Encrypt3 scenarios.

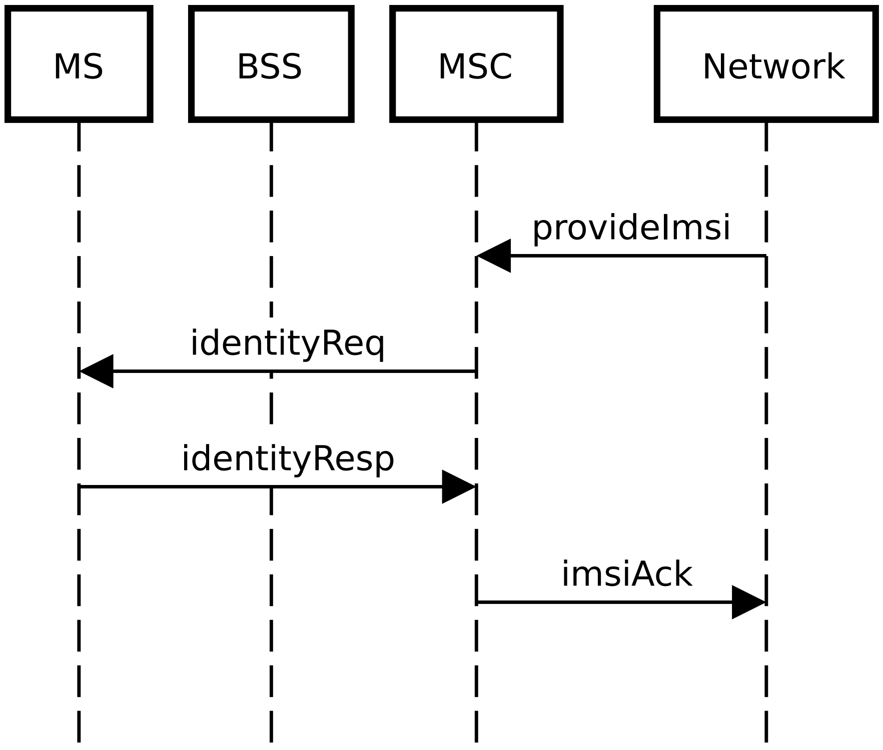


Figure A.42: bMSC for GSM's Identify1, Identify2, and Identify3 scenarios.

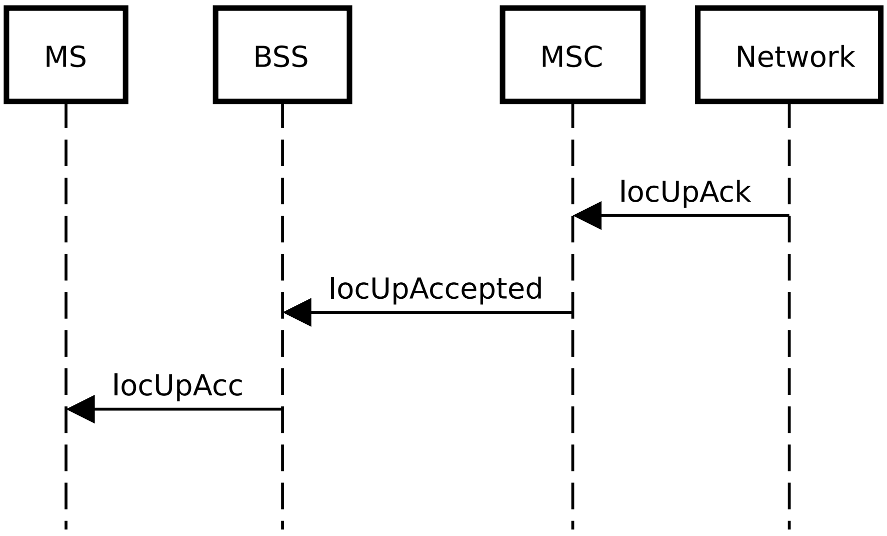


Figure A.43: bMSC for GSM's LocationUpd scenario.

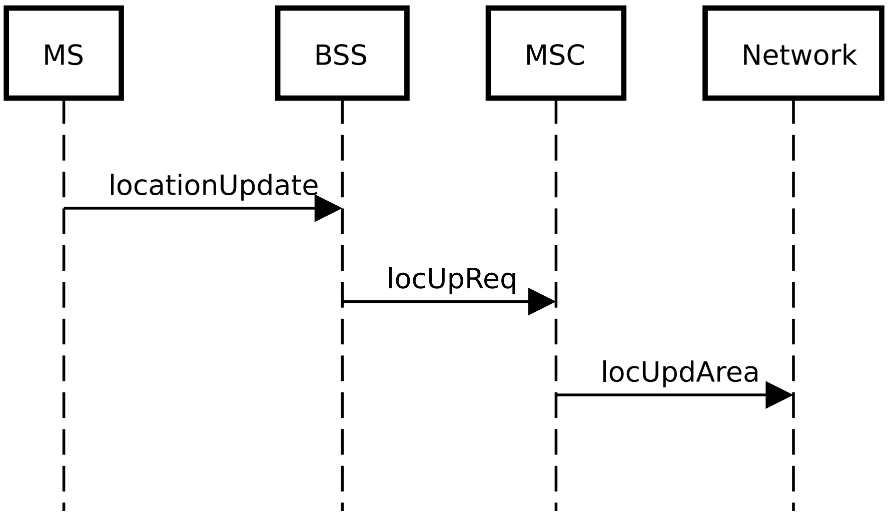


Figure A.44: bMSC for GSM's LocUpdReq scenario.

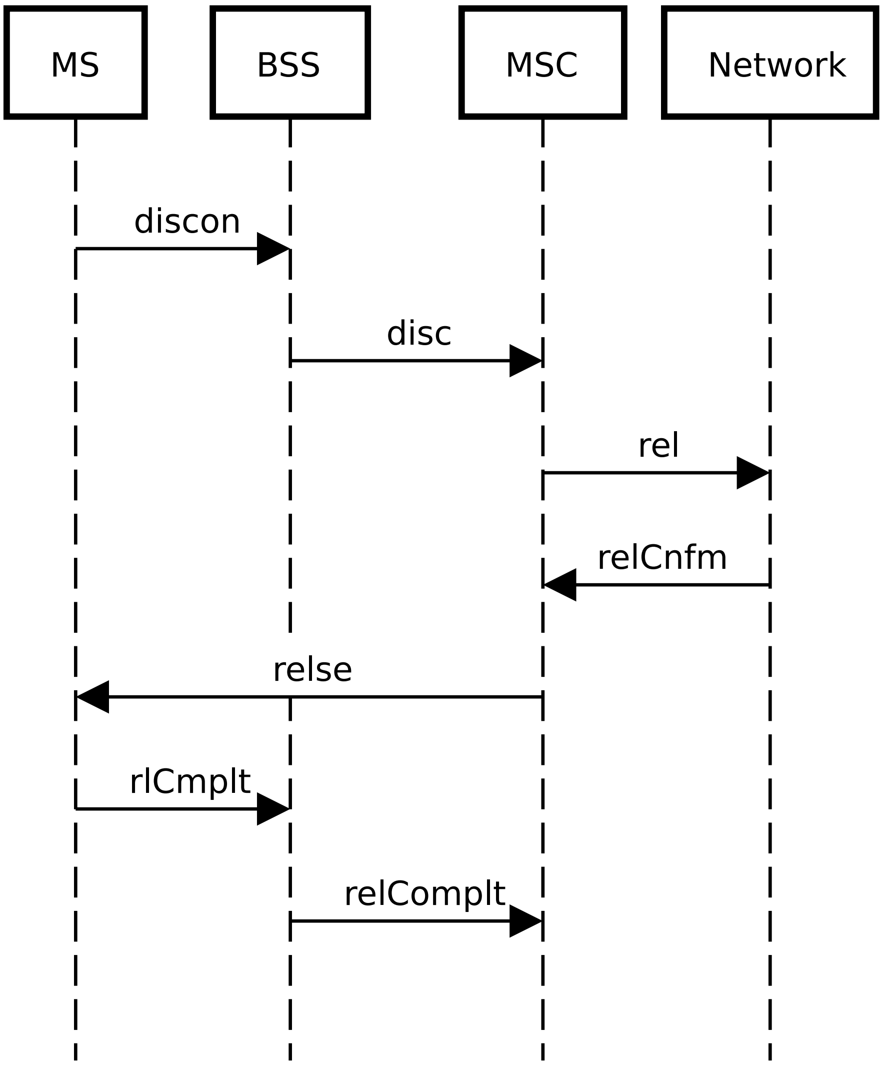


Figure A.45: bMSC for GSM's MobileOrCR scenario.

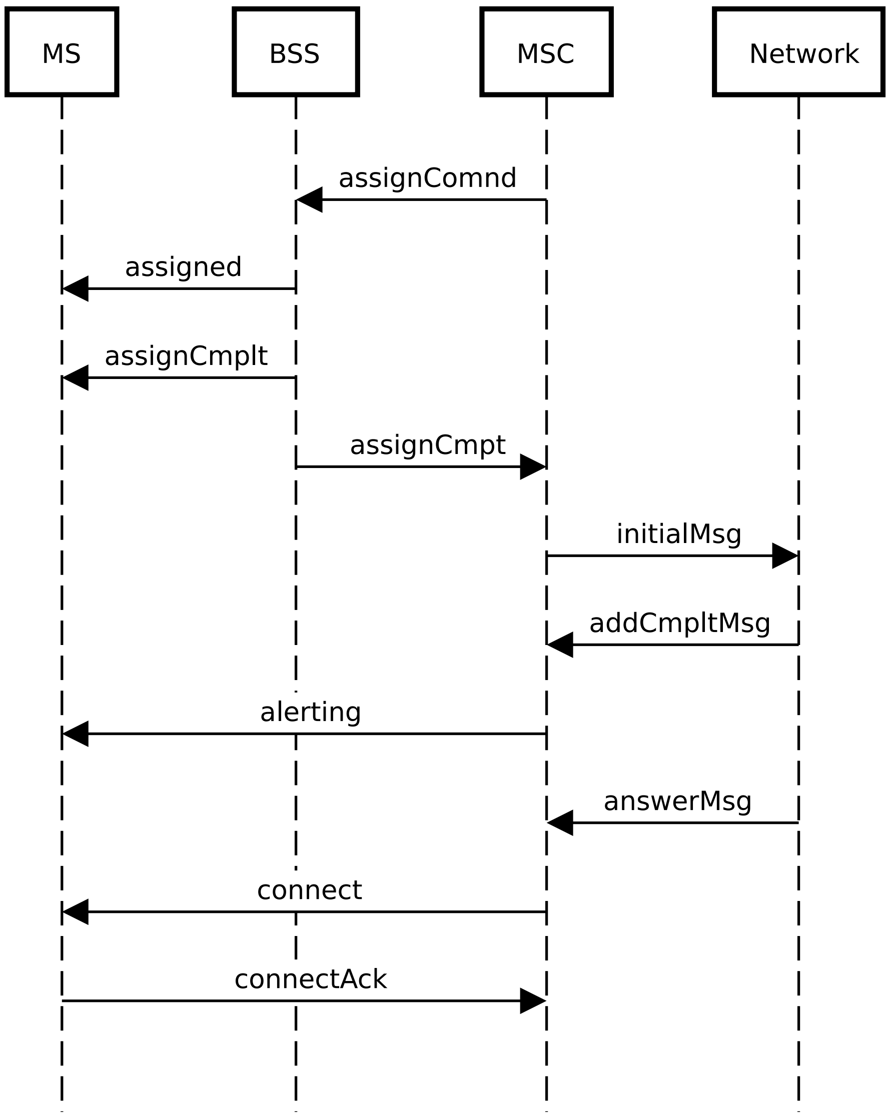


Figure A.46: bMSC for GSM's MobileOrCS scenario.

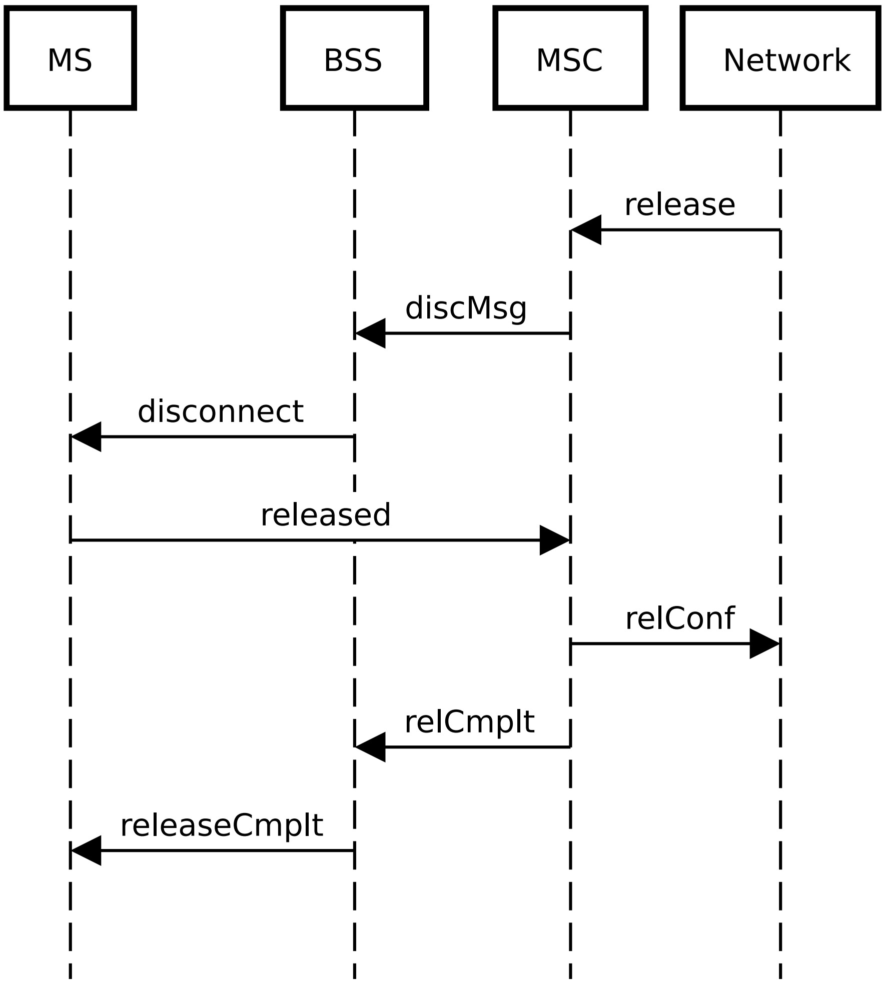


Figure A.47: bMSC for GSM's MobileTrCR scenario.

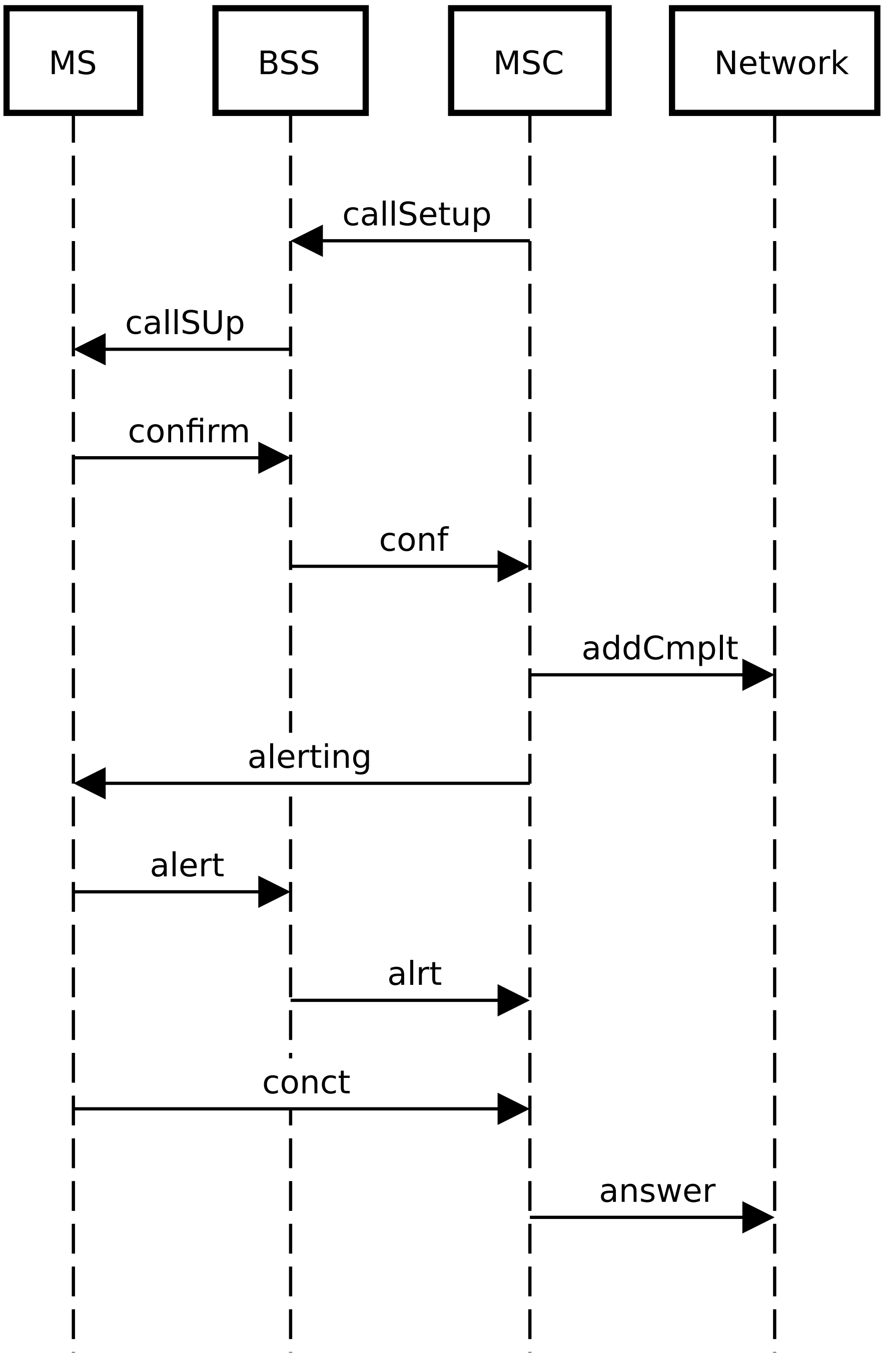


Figure A.48: bMSC for GSM's MobileTrCS scenario.

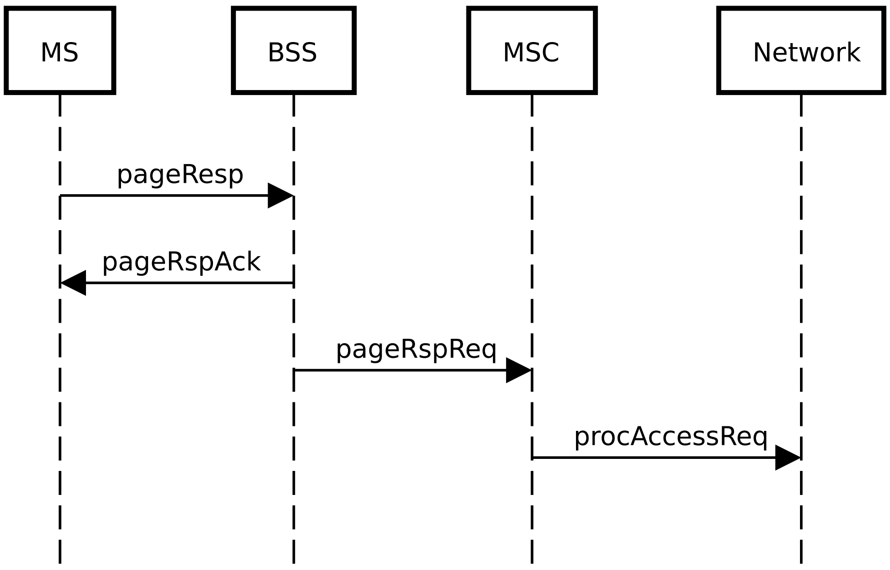


Figure A.49: bMSC for GSM's PagingResp scenario.

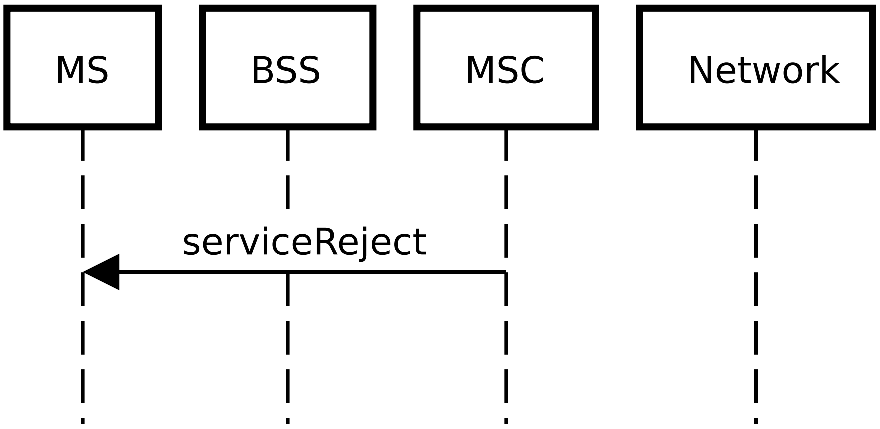


Figure A.50: bMSC for GSM's Reject1, Reject2, and Reject3 scenarios.

A.7 Semantic Search Multi-Agent System (S.S. MAS)

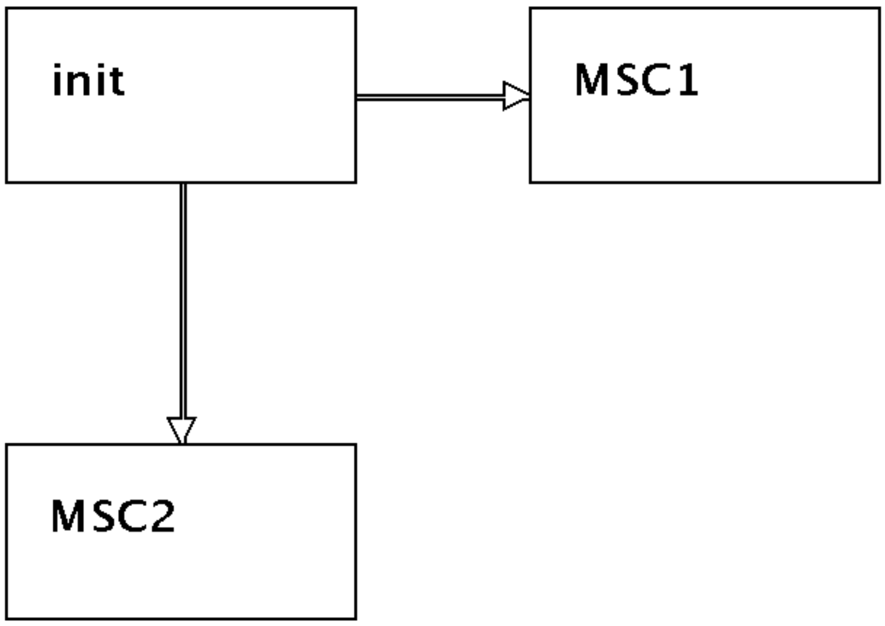


Figure A.51: S.S. MAS's hMSC.

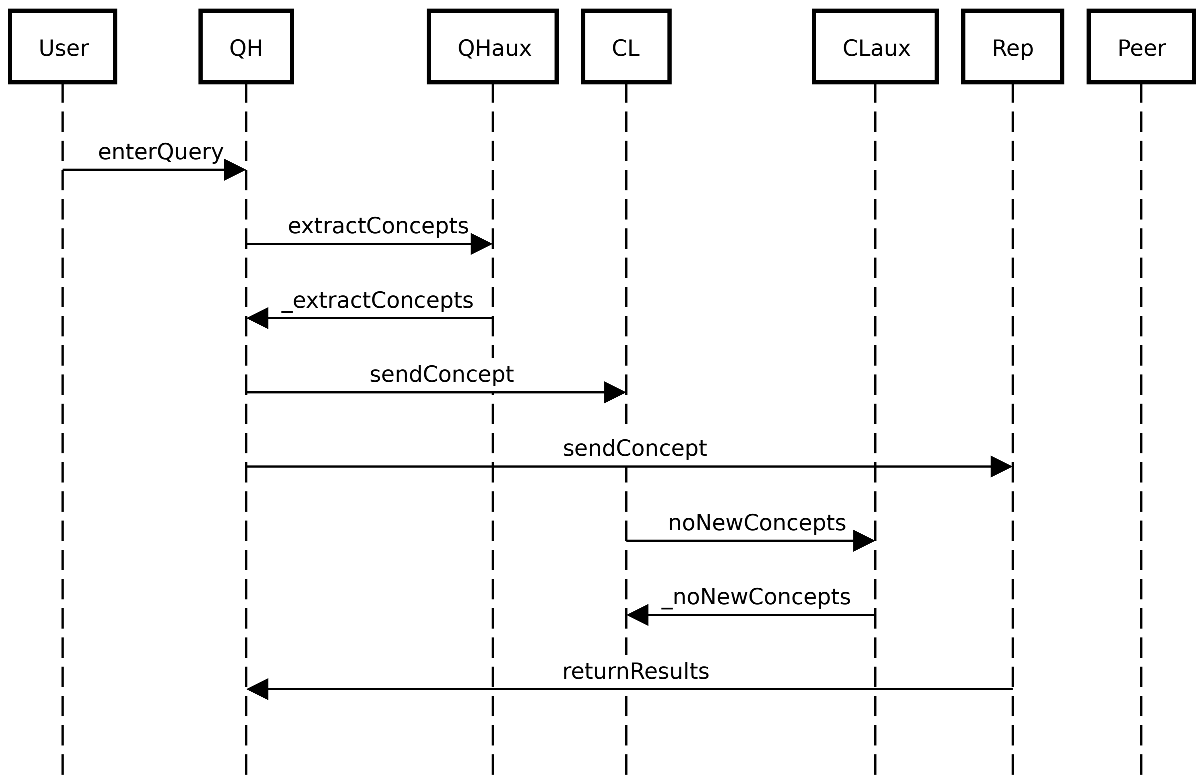


Figure A.52: bMSC for S.S. MAS's MSC1

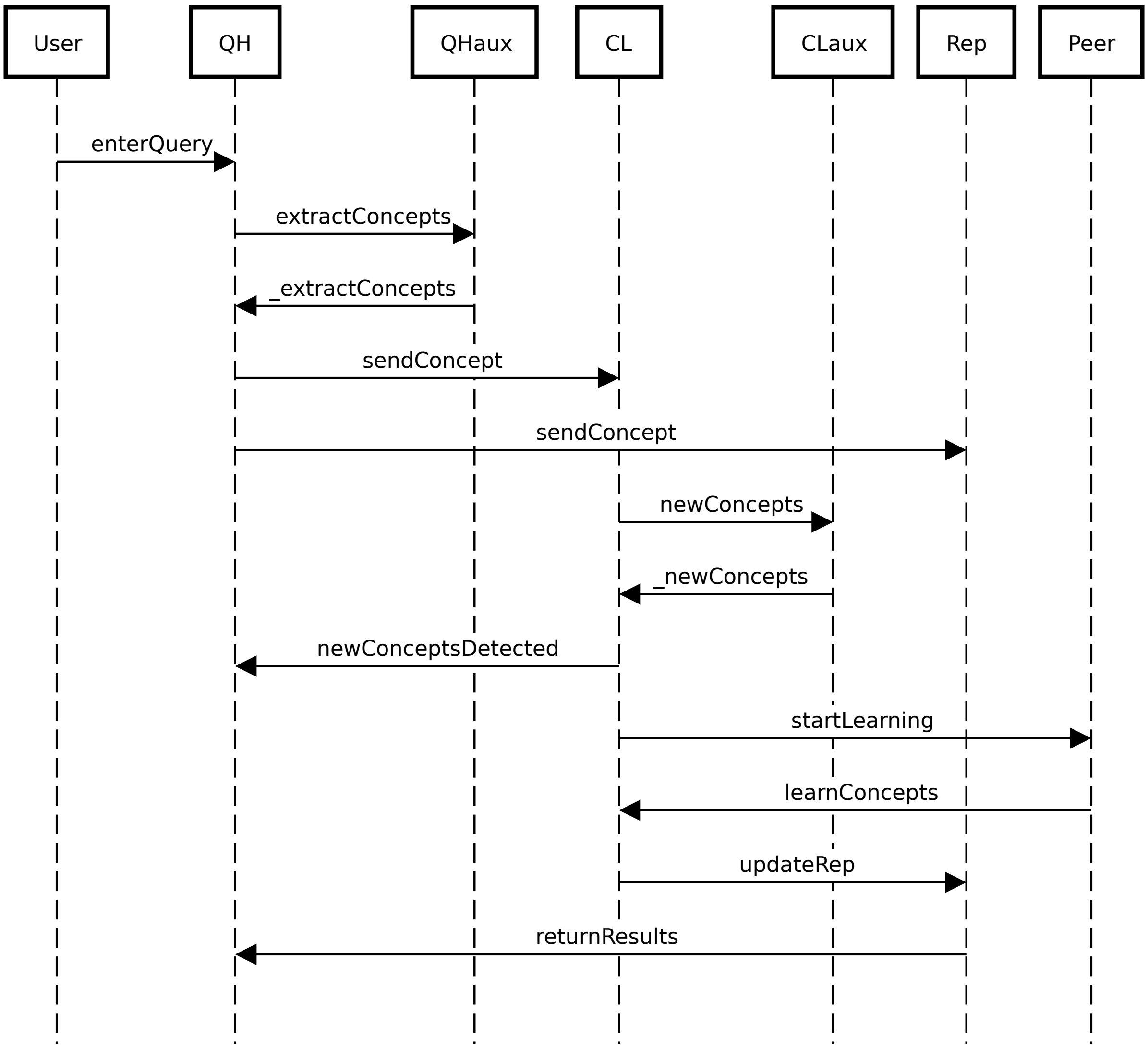


Figure A.53: bMSC for S.S. MAS's MSC2