ABSTRACT

Title of dissertation: BEHAVIORAL REFLEXION MODELS FOR SOFTWARE ARCHITECTURE

Christopher F. Ackermann
Doctor of Philosophy
2010

Dissertation directed by: Professor Rance Cleaveland
Department of Computer Science

Developing and maintaining software is difficult and error prone. This can at least partially be attributed to its constantly evolving nature. Requirements are seldom finalized before the development begins, but evolve constantly as new details about the system become known and stakeholder objectives change. As requirements are altered, the software architecture must be updated to accommodate these changes. This includes updating the architecture documentation, which serves as the design specification as well as a means of comprehending complex systems. Furthermore, the software architecture of the implementation must be adapted in order to ensure that the system complies with both functional and non-functional requirements.

In practice, however, software changes are often applied in an ad-hoc manner. As a result, the implementation frequently deviates from the architecture documentation, rendering the latter useless for supporting system engineers in comprehending

the system and aiding maintenance tasks. Furthermore, errors that are introduced during the implementation lead to discrepancies between the system and the intended architecture design. Consequently, it cannot be guaranteed that the system meets the desired quality objectives, such as reliability and dependability.

We present the behavioral reflexion model approach, which aims to support the system engineer in identifying and resolving discrepancies between software architecture representations. In our approach, the system engineer is supported in producing architecture documentation that reflects the intended architecture. Furthermore, discrepancies between the implementation and documentation are identified. These discrepancies are then illustrated graphically in a reflexion model, which guides debugging activities. In this research, we are concerned with architecture representations of system behaviors and focus in particular on distributed systems.

In this thesis, we describe how architecture discrepancies are introduced and the implications for the reliability and maintainability of the system. We then discuss the individual components of the behavioral reflexion model approach in detail. Finally, we provide an evaluation of our approach in the form of two case studies. In these studies, we applied the behavioral reflexion model approach to two space-mission systems with the goal to resolve problems in their reliability and maintainability.

BEHAVIORAL REFLEXION MODELS FOR SOFTWARE ARCHITECTURE

by

Christopher F. Ackermann

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:
Professor Rance Cleaveland, Chair/Advisor
Professor Atif Memon
Professor Alan Sussman
Professor Amol Deshpande
Professor David Barbe

Dedication

To Ali and Sophie.

Acknowledgments

The research presented in this dissertation was only possible with the support of many. I would like to extend my thanks to all of them.

I would like to thank my advisor Rance Cleaveland for providing ideas and helping me conducting solid research. I also thank him for teaching me essential lessons that I will need in my role as a researcher.

I wish to also express my gratitude to Mikael Lindvall, who has been a mentor and a friend. He has taught me how to write papers, give talks, and perhaps most importantly, he taught me how to be a good colleague. I very much value all he has done for me.

My appreciation also goes out to the members of my thesis committee: Atif Memon, Alan Sussman, Amol Deshpande, and David Barbe. Their insightful feedback helped me to refine my research and this dissertation.

Great thanks also goes to Fraunhofer USA - CESE for supporting my studies and giving me the chance to gather real-world experience. I was very fortunate to have the chance to collaborate with excellent researchers and experienced colleagues. They taught me how to work in a team, under a budget, and with customers. These lessons will be helpful wherever my future may take me.

Special thanks also goes to the interns who supported my research, not only in developing tools, but also contributing ideas and constructive feedback. In particular, I would like to thank Christoph Schulze, Sebastian Windisch, and Bernd Kemmerle. They all have done excellent work.

I am grateful for the many interesting and fruitful discussions with our collaborators at the Johns Hopkins University's Applied Physics Laboratory: Bill Stratton and Deane Sibol. They helped me understand the complex domain of aerospace engineering and patiently answered my many questions.

This work would not have been possible without the support by the National Aeronautics and Space Administration (NASA). I would like to especially extend my gratitude to Lisa Montgomery and Sally Godfrey for believing in our work.

I would also like to thank my family for their great support and encouragement throughout my study: My mom for always being proud of me and my siblings for not being discouraged to visit despite the great distance between us. My wife's family, which I made my own, for supporting me every step on the way.

Finally, a person without whom none of these words would have found the paper they are written on, I must thank my wife Ali. Being married to a graduate student means often times taking over many of the household duties, giving encouragement in difficult times, lots of proofreading, and listening to many practice talks. Ali allowed me to devote much of my time to my research. I will always owe her for that.

Table of Contents

List of Figures

Chapter 1

Introduction

Modern software systems continue to grow larger and more complex as they are expected to fulfill greater and more diverse functionality. When designing such systems, in addition to identifying appropriate algorithms and data structures, it is necessary to also specify the system's high-level organization. That organization is referred to as *software architecture*. Software architecture identifies the system components and describes how they are combined to form the overall system. Components are building blocks of a system, which have certain properties and interfaces to other components. What distinguishes architecture design from detailed design is that the architecture only describes the components, their externally visible properties, and the relationships among them [6]. That implies that it is not concerned with the component-internal design. Contrary to the term architecture used in other domains, software architecture describes not only structural but also behavioral properties of the system. While the structure describes what components the system consists of and the dependencies they share, the behavior specifies how the components interact during runtime.

Designing software architecture is a crucial part of the software design process as architecture design decisions affect the non-functional quality attributes of a system [64]. One of the main goals of the software architecture design activity

is, thus, to construct a system that meets the desired quality attributes, such as maintainability, performance, and reliability. By choosing an appropriate software architecture design, one can construct a system that meets the desired quality attributes. For instance, one way to increase the maintainability of a system is to reduce the number of static and dynamic dependencies between source code units. Reducing the number of dependencies decreases the likelihood for changes to spread between components and reduces the implementation effort [77].

Once the architecture design decisions have been made, different aspects of the software architecture should be documented [10]. A variety of notations exists to describe the design decisions regarding both the structure and the behavior of software architecture. For instance, UML Class Diagrams [59] illustrate the source code units and the static dependencies between them. UML Sequence Diagrams [13], on the other hand, show the sequence and timing of interactions between components. Documenting software architecture is not only necessary as design specification for implementing the system but it may also serve to support other design, implementation, and maintenance activities. During the design, architecture views can be used to communicate system properties on an abstract level among the members of the design team and even to stakeholders. Since the views capture only specific high-level characteristics and hide implementation details, they can describe even large systems in a comprehensible manner. After the architecture has been specified, architecture views are used by the developer to implement a system that is consistent with the software architecture design. The maintenance is supported as new developers can use architecture documentation to comprehend

the system's high-level characteristics. Furthermore, architecture documentation supports reuse [37] and is the basis of several change impact analysis approaches [1].

In practice, however, architecture documentation quickly becomes outdated and does not accurately describe the architecture of the implementation [40]. The software architecture as it is initially documented, the *documented architecture*, often does not reflect the final product, i.e., the *implemented architecture*. As new requirements arise and existing requirements change, the architecture must be adapted to accommodate these revisions. Amid time pressure, architecture changes are often applied only to the implementation while the documentation remains unaltered. As a result, the implementation drifts apart from the architecture documentation [56]. Inconsistencies between documented and implemented architecture renders the documentation unsuitable for aiding comprehension and system maintenance. Discrepancies also exist between the implemented architecture and the mental model of the architecture the developer had in mind when implementing the system. We refer to this mental model as the *intended architecture*. When changes to the architecture are made, they are communicated to the developer and the intended architecture is updated. Thus, the intended architecture reflects the accurate and up-to-date architecture design. The developer implements the software system with the intended architecture in mind. However, during the implementation process architecture errors, similar to functional errors, are inevitably introduced. An architecture error could be a dependency between two components that was not specified in the architecture design or a behavior that violates the behavioral rules defined in the architecture documentation. Architecture errors, unlike functional errors, do often

not cause system failures but may affect the non-functional qualities of the system.

The goal of this research is to identify inconsistencies among architecture representations to allow for controlled changes of the system and resolve issues in its reliability. We aim to update the software architecture documentation to match the designer's mental model of the desired architecture. That means that new documentation is produced if none is available or inaccurate documentation is modified to reflect the desired state of the system. Once the documentation is up-to-date, the implementation can be compared against it and deviations from the intended architecture, i.e., implementation errors, can be identified.

The issue of inconsistent architecture representations has been recognized by various researchers and approaches have been presented to remedy the problem. *Architecture recovery* techniques extract architecture views from the system implementation to update existing inaccurate documentation and support system comprehension [16][30]. *Architecture compliance checking* approaches have been developed to compare an implementation to an architecture design specification and identify discrepancies [27][57].

Unfortunately, these approached focus mainly of structural aspects of software architecture. As we have pointed out previously, the software architecture of a system consists of both structure and behavior. Both impact the qualities of a system, such as reliability and maintainability, and both must be documented as comprehension of these aspects is crucial for the system engineer to be able to maintain the system. In order to address the clear lack of methods for identifying inconsistencies among architecture representations of the system behavior, we focus our

research specifically on such behavioral aspects. The behavior of software architecture describes solely how components interact but ignores the component-internal behavior. In our research, we aim to identify discrepancies between the interaction behavior of a system, the behavioral architecture documentation, and the intended architecture.

While the behavior is an important aspect of any system, it is especially crucial for distributed systems. In distributed systems, components operate autonomously and interact via unreliable communication channels, such as networks or busses. The interactions through these communication channels are slower than local interactions and also less reliable. Interaction protocols are specified as part of the architecture specification in a way so as to achieve certain levels of performance and reliability. In order to guarantee that the distributed system fulfills these quality requirements, it is necessary to ensure that interactions follow the architecture specification. Furthermore, any change to a component must be made so as to not unintentionally alter the interface to other components as that might compromise the quality of the overall system. In our research, we will particularly focus on distributed systems to address the need for consistent behavioral descriptions and the lack of methods for identifying them. However, we do not exclude centralized systems but believe that most of our research applies to a large variety of systems. From a behavioral perspective, distributed systems present a challenge due their complexity and need for reliability. The findings of our research can be applied to systems in which interaction behavior might be of lesser importance.

We have developed an approach for identifying inconsistencies among architec-

ture representations of system behavior that are produced throughout the software lifecycle. In that approach, the system engineer is supported in producing documentation that accurately reflects the intended architecture and means are provided to identify deviations of the implementation from the intended architecture. The ultimate goal is to allow for controlled maintenance of the system and to resolve reliability issues.

1.1 The Behavioral Reflexion Model Approach

Our approach is based on the idea of reflexion models introduced by Murphy et al. [45]. The goal of their approach is to illustrate deviations of the system implementation from the engineer's mental model of its structure. In that technique, the user manually constructs a so-called *high-level model* that describes the desired system structure with its components and dependencies between them. The source code is then parsed to extract architecture-level information from it. Finally, the extracted architecture information that describes the state of the implementation is compared to the model of the intended architecture. The result of the comparison is a new model, the *reflexion model*, which graphically illustrates dependencies that are inconsistent among intended architecture and the implementation. Based on the reflexion model, the system engineer can gain knowledge about the system in preparation of maintenance tasks and identify structural flaws in the implementation. The reflexion model approach has been implemented in several tools [34][36] and has proved to be useful not only in identifying discrepancies but also to support

reasoning about them using the graphical output.

We present the *Behavioral Reflexion Model Approach*, which applies the idea of reflexion models for aligning behavioral representations of software architecture. In the first step of our approach, the system engineer monitors the runtime of the system and collects a set of execution traces. Each execution trace describes the interaction behavior that can be observed during a single system execution.

Subsequently, the system engineer constructs a high-level model describing the intended architecture. In practice, constructing such a model is difficult as accurate documentation is often lacking. Hence, we provide automated support that constructs an initial model, which the system engineer can modify to derive an accurate description of the intended architecture.

Finally, the execution traces that have been collected from the implementation are compared to the high-level model and a reflexion model is computed, graphically illustrating the discrepancies. The system engineer inspects the reflexion models and reasons about the discrepancies. In that stage, decisions are made whether a discrepancy is caused by an inaccurate high-level model or by an erroneous system execution. Depending on the result of that evaluation, the high-level model or the system implementation is updated. This process may be applied several times until all discrepancies have been resolved. The result is a high-level model that accurately reflects the intended architecture and can be used to update the documentation. The implementation also reflects the intended architecture as deviation shave been resolved.

1.2 Contributions and Thesis Organization

The following outlines the organization of this thesis and highlights the contribution of this research.

In Chapter 2, we define basic terms that are used throughout this thesis. Furthermore, we discuss works that are related to our research. In that discussion, we consider works in the areas of software architecture, architecture compliance checking, and distributed systems.

In Chapter 3, we provide an overview of the behavioral reflexion model approach. We describe the basic components of our approach and briefly explain how these components can be applied for the task of aligning architecture representations.

In Chapter 4, we describe the first component of our approach: the trace extraction. In that chapter, we provide details on the information the system engineer extracts from the system implementation and discuss means with which runtime information can be extracted from centralized systems. Since few techniques exist for extracting execution traces from distributed systems, we describe a method that we have developed for extracting runtime information from distributed systems that employ traditional message passing as a means of interactions between components.

In Chapter 5, we discuss what aspects are specified in a high-level model and provide a specific set of rules that is supported. Furthermore, we present a method that we have developed for constructing the high-level model. That includes a detailed description of the machine-learning algorithms employed to construct a

partial high-level model. Furthermore, we describe how the system engineer can use the inferred information to refine the high-level model.

In Chapter 6, we discuss the comparison of the execution traces to the high-level model and the subsequent computation of the reflexion model. In that chapter, we first discuss the mapping of execution traces to the high-level model. In particular, we present an automated mapping approach, which enables system engineers to map even large execution traces with little effort to the high-level model. We then describe how execution traces are compared to the rules specified in the high-level model and violations of these rules are identified. Finally, the chapter elaborates on the construction of the reflexion model, which graphically illustrates the violations.

In Chapter 7, we present two case studies in which illustrate how the behavioral reflexion model approach is applied to real-world systems. We discuss how the application of our approach differs depending on the goal of the analysis. We also provide a performance evaluation and discuss limitations.

Chapter 2

Background and Related Work

With this research, we are addressing issues in the area of software architecture relating to inconsistencies among architecture representations. The following introduces basic concepts of software architecture and describes the representations of it that are produced throughout the software lifecycle. Subsequently, we discuss other approaches whose goal it is to identify inconsistencies between software artifacts, such as documentation and source code. As we focus in this research primarily on distributed systems, we also provide basic terminology and describe challenges that are specific to ensuring maintainability and reliability in such systems.

2.1 Software Architecture

The size and complexity of today's software systems is increasing steadily and with it the challenge of designing, implementing, and maintaining them. Software architecture helps mitigate these challenges by providing an abstract view of the system [67]. The abstract view hides implementation details and allows the system engineer to reason even about large systems. From a software architecture perspective, a system is viewed as being composed of components [6] that are interdependent and interact with each other during runtime. Software architecture is only concerned with the high-level characteristics of a system and hides the detailed design. More

specifically, only those properties that are discernible at the component interfaces are architecture relevant [67]. These include structural dependencies among components as well as their interaction behavior. Structural and behavioral properties that do not surface at the component interface are hidden. This allows system engineers to reason about systems on a high-level without dealing with implementation details, such as algorithms and data structures.

While architecture in other domains is a purely structural notion, software architecture describes both structural as well as behavioral characteristics. The structural aspect of a software architecture describes the components and dependencies between them. In an object-oriented system, the structure could describe classes that are connected by function-call or parameter-access dependencies [10]. The structure illustrates the composition of components and emphasizes the relationships amongst them. Behavior is also a vital part of the architecture as it constitutes a central part of the system itself and must be addressed in the high-level or architectural design [67]. The behavior describes how components interact during runtime. The architecture design specification defines rules on different behavioral characteristics, such as the order in which interactions occur.

The software architecture design of a system is driven by a variety of constraints as well as stakeholder's objectives [15]. A system often has several stakeholders who have different concerns that they wish the system to guarantee or optimize [6]. These could be business concerns, such as low cost or short time to market. End users and developers might have more technical objectives, such as high performance, low cost of maintenance, and high reliability. Such objectives are

considered non-functional, as they do not describe functions of a system but rather how the system should fulfill these functions [17]. Other non-functional quality attributes are modifiability, availability, usability, interoperability, and compatibility. The key objective of a software architect is to design an architecture that meets such non-functional quality goals.

The architecture design that is created during the design phase is typically captured in architecture documentation. The documentation usually describes different architecture characteristics using a mix of graphical models and natural language [20]. The graphical models, or architecture views, are useful to describe different architecture characteristics in a concise manner. Natural language descriptions can be used to support architecture views by explaining the background for the design decisions, i.e., the design rationale. Various notations have been proposed for describing architecture views. Perhaps the most commonly used are specified as part of the Unified Modeling Language (UML) [10]. UML provides notations for describing a range of different characteristics using a set of diagrams. The two main categories that are distinguished by the UML are structural and behavioral diagrams. An example for a structural diagram is a class diagram, which illustrates the classes and the call or access dependencies they share. Behavioral diagrams are timing diagrams, state charts, and sequence diagrams. Each of the diagrams shows a different behavioral aspect and allows the user to focus on specific characteristics, while hiding others.

Software architecture views also play a major role in comprehending systems [67]. Since they abstract from implementation details and show only specific as-

pects, they can be used to comprehend even large and complex systems. As such, they serve to aid developers in implementing modifications to the system through the entire lifecycle [53]. This becomes especially important during software maintenance when the original developers have left the organization and new members have joined the team. Every maintenance task requires knowledge of particular system characteristics and of specific parts of the system. Software architecture views document different characteristics in a way that allows developers inexperienced with the system to gain knowledge about its structure and behavior [15]. Based on that knowledge, the developer can identify what parts of the system need to be changed. She can assess the impact the change will have on the system and estimate the change effort [4]. Furthermore, only with knowledge about the existing architecture can the developer implement a change that preserves the non-functional qualities.

Over the lifetime of a system, a number of architecture representations are created. Prior to the implementation, the software architects construct an architecture that meets the non-functional stakeholder objectives. Before design decisions are documented, the software architect has a mental model of how the software architecture should look like. We refer to this mental model as the *intended architecture*. The intended architecture is not a concrete artifact but exists in the minds of architects and developers. In order to communicate architecture decisions to developers, the architecture is documented in views supported by natural language descriptions [20]. In this process a new architecture representation is created, the *documented architecture*. The documented architecture serves the developers as design specification and plays a major role during maintenance as it facilitates

comprehension of high-level system characteristics. The task of the developer is to implement a system that adheres to the architecture design specification, i.e., the intended architecture. She obtains the intended architecture from the design documentation and implements the system accordingly. The system implementation constitutes yet another architecture representation, the *implemented architecture*. The implemented architecture is perhaps the most important representation as it determines the system's compliance to the desired non-functional quality attributes.

Even if the architecture is designed with greatest care, it will almost certainly experience updates during the development process as stakeholder objectives change and new details about the system become known [56]. Changes to the architecture are also made during software maintenance, when bugs are fixed, new functions are added, and the system is adapted to different computing environments . It is crucial that architecture modifications are applied in a controlled manner [5]. That means that they should retain the desired non-functional qualities. In addition, every time the software architecture is modified, the different architecture representations must be updated and consistency among them must be preserved .

2.2 Architecture Compliance Checking

Maintaining consistency among architecture representations can be difficult as the architecture may be subject to modifications and evolves continuously [67]. Typically, the architecture representations are independent from each other and only if modifications are applied to all representations can consistency among them

be maintained. In practice, however, only certain architecture representations are updated while others remain unchanged [2]. Architecture documentation is often created prior to development to serve as architecture specification. When requirements change and the architecture is modified, changes are applied in an ad-hoc manner [70]. That means that they are often only communicated directly to the developer who implements them. However, since the time is limited, the documentation remains unaltered. As a result, the implementation already deviates from the architecture documentation when the system is deployed [56]. The documented and the implemented architecture drift further apart as additional changes are implemented during software maintenance. Such uncontrolled changes lead to a deterioration or degeneration of software architecture [40]. That means that it is not only inconsistent with its documentation but it also does not satisfy the desired non-functional qualities.

Techniques to address this problem can be proactive or reactive. Proactive approaches aim to support the developer during implementation and prevent the introduction of inconsistencies between architecture documentation and implementation. Reactive approaches, on the other hand, are used to re-align the documentation with the architecture (or vice versa) after discrepancies have been introduced.

Together [11] falls into the category of proactive approaches. It continuously maintains consistency between implementation and architecture views. Any updates to the implementation are immediately applied to the views and changes to the views are propagated to the implementation. This prevents architecture discrepancies from being introduced. Other tools prevent architecture inconsistencies

from being introduced via code generation. With HUGO [33], the system engineer can specify a state chart describing the desired behavior of the system. The tool then generates Java source code that exhibits the specified behavior. In a similar approach, code is generated from class and sequence diagrams [41]. Code generation approaches only ensure consistency when changes are made to the model but do not update the model when the source code is modified. While prevention of architecture deviations is commonly known to be less costly than resolving them, proactive approaches are not widely applied in practice. One reason might be that an immediate benefit of adhering to the architecture specification is not visible to the developer. Functionality often takes precedence over non-functional quality objectives when the time is limited. Furthermore, proactive approaches often require the developer to change his process and habits, which can only be achieved with great commitment of management and the development team.

Architecture compliance checking is a reactive approach as it aims to detect errors after they have been introduced. It refers to techniques that compare an implemented software system to its architecture design specification in order to identify deviations between the two. The approaches differ in the types of architecture discrepancies that they detect and the notation that is used to describe them. Furthermore, compliance checking approaches have different ways of illustrating deviations, which becomes important when the aim is to support the process of resolving the detected problems.

The first approach that triggered interest in architecture compliance-checking was Murphy's work on reflexion models [45]. In the reflexion model approach,

the system engineer specifies the desired software structure in a so-called high-level. Then, structural information about the implementation is extracted from its source code. Finally, a reflexion model is computed, which shows deviations in the dependencies that were specified in the high-level model and the dependencies implemented in source code. The graphical illustrations of deviations supports the system engineer in reasoning about them and can be used in support of resolving the discrepancies. The Software Architecture Visualization and Evaluation (SAVE) tool [34] implements the idea of reflexion models. It allows one to specify high-level models in a graphical editor, guides the user in mapping source code units to the high-level components, and produces a reflexion model that graphically illustrates the deviations between the high-level model and the implementation. The reflexion model idea is also implemented in the Bauhaus tool [36]. Other methods allow the evaluation of an implemented architecture in respect to more formal design rules that specify the desired or expected structural properties [35]. Two examples of such rules are relation conformance rules [57] and component access rules (similar to ADL [42]). Compliance-checking in this form, i.e. of static architectures, has been applied to real world systems with great success [66]. Compliance checking tools based on static analysis are, however, insufficient for highly dynamic systems. In systems that are implemented using component-based frameworks (e.g., Java Beans), components can be added and removed from the system at runtime, leading to a continuously evolving architecture. Colored Petri-Nets may be employed to track changes in such an evolving architecture and to compare it to the architecture specification of the system [27].

Architecture compliance checking approaches have focused on evaluating the adherence of the implementation to a structural model. Behavioral characteristics have largely been ignored in the area of architecture compliance checking. This suggests that behavior is not a vital part of the architecture or that it is free of errors. However, both statements are false. The way components interact can determine whether a system can be sufficiently reliable, efficient, and maintainable. Part of the design process should thus be to specify appropriate interaction protocols and document the overall interaction behavior. In order to ensure that the system indeed adheres to the desired quality attributes, it is essential that the system's compliance with the behavioral specification is verified.

While the verification of behavioral properties in the context of software architecture compliance checking has not been addressed, a large number of testing and verification approaches have been proposed to assess the adherence of system implementations to various behavioral properties. Much of this work is published under the term "protocol verification/testing". A protocol is defined as "*a set of rules that govern the exchange of information between processes in a communication system*" [73]. Since the definition of protocol is general, the protocol verification techniques differ greatly in the kinds of behavioral attributes they are evaluating. A common way of defining protocol specifications is in form of Finite State Machines (FSM) or Labeled Transition Systems (LTS) [9][71]. Both notations allow for specifying a sequence of events combined with constraints on parameter values. While the verification of protocols is applied in order to ensure proper communication amongst elements in general [69], they are also employed in a more specific context, such as

security [19] or Multi Agent Systems [71]. Other techniques go beyond sequencing and parameter properties and include timing in their specification and verification. Uppaal [7] models systems as timed automata, in which transitions between states are guarded by temporal conditions and verifies whether system adheres to these properties [25]. The Temporal Rover [24] is a tool for verifying a programs adherence to specifications in Linear-Time Temporal Logic (LTL) and Metric Temporal Logic (MTL) during execution. LTL captures properties on the temporal sequence of events and MTL is an extension to LTL that includes information about the timing of events.

Testing approaches are applied before the system is deployed in order ensure its proper behavior. In combination with coverage criterion one can make claims about the degree to which the program has been verified. Model checking can be employed to verify that the properties expressed in temporal logic holds for a program or more specifically for a model thereof [18]. The advantage of model checking approaches is that they can guarantee the absence of property violations. However, they can often only be applied to systems of limited size. The disadvantage of verification approaches in general is that they only produce a yes/not-sure answer indicating whether a desired property could be established [28]. While the goal of architecture compliance-checking is certainly to compare the behavior of the implemented system to its specification, its strength lies in producing views that show deviations in the context of the entire architecture. This facilitates comprehension and allows the engineer to reason about deviations that might have been uncovered. As such, it may act in support of conventional testing tools.

2.3 Distributed Systems

Architecture analysis has focused primarily on single systems. However, the current state of practice is shifting from single systems to distributed systems. A distributed system consists of multiple distributed processors which execute largely independent processes [68]. Potential benefits of distributed systems are increased performance as processes can be executed in parallel, increased reliability since a process might have multiple copies, increased reliability due to the higher likelihood to recover with the help of backup processes, and increased flexibility as new components can be added more easily [76]. Also, some systems must be implemented as a distributed system since its components are geographically dispersed. An example for such a system could be a space mission where telemetry data is collected in space and transmitted to earth for processing and analysis.

Components form a distributed system by collaboration via inter-process communication [68]. One way for processes to communicate is by means of shared memory. In such a setup, the processes read to and write from the same memory [58]. This is referred to as a strongly coupled system. Conversely, in a loosely coupled system, components interact by exchanging message via networks. A network establishes communication channels between components and presents a means for transporting messages from the sending to the receiving component. Different network topologies exist, such as ring, star, or bus topology [22]. The topology determines how processors are connected to each other and how messages travel through the network. In the research presented in this thesis we are concerned with such loosely

coupled systems.

The fact that components of a distributed system collaborate via message passing on networks has a significant impact on the way the components are designed, developed, integrated, and maintained. Components of a single system typically interact via function calls. With a function call, a component can transfer a virtually unlimited amount of data in an instant reliably to another component. Transmitting data across a network is far more limited. The time a message travels from the sending to the receiving component on a network is significantly larger than the exchange of information within a single system. Furthermore, networks are generally unreliable and messages are frequently lost during transmission. In order to nevertheless provide a communication platform that is sufficiently reliable, specific communication procedures are defined that include mechanism to check that data arrives successfully at the receiving component. The communication procedures are referred to as protocols and describe rules that govern the interaction between components of the distributed system [76].

During the design phase of a distributed system, one or more protocols are specified in a way so as to achieve a certain reliability and performance. During the implementation, the protocol specifications are used to build components whose interfaces adhere to the documented interaction behavior. Upon completion of the implementation, the internal behavior of each component as well as its interaction behavior must be verified. Only if the interface behavior of each individual component is according to the specification can the components be combined to form a distributed system that fulfills the desired functional and non-functional re-

quirements. Similarly, components that are added during system maintenance must adhere to the protocols employed by legacy components.

Although the challenges of developing a distributed system are well known, many systems today suffer significant failures of a non-functional nature [75]. For instance, consider the example of a component that requests certain data from another component. A reliability failure occurs since the correct information does not reach the receiving component. Such a failure could be caused by a variety of different errors. The requesting component could have failed to formulate the query correctly. Messages could have been lost during transmission and not been recovered. Another option could be that the sending component failed to process the request properly. In order to investigate a failure that occurs in a distributed system, mechanisms must exist to check the system's compliance with the rules that are specified as part of the protocol. This provides insight into whether any violation of the desired interaction behavior exists.

Protocol testing and verification approaches have been presented that support the identification or errors in the interface behavior of components [9]. Different approaches and tools are used to verify different behavioral properties. Some tools evaluate whether a system always produces proper sequences of events [65], others focus on the timing behavior [23], and again others verify whether constraints on data are adhered to [69]. In combination with a method for ensuring a certain degree of coverage, it is possible to make claim as to what extent a system adheres to its specification. The coverage techniques can also be divided into ones that focus on sequencing [74] and timing [18] properties. The testing tools aim to evaluate the

correctness of single systems, rather than a distributed system as a whole. However, misbehavior often surfaces only when the heterogeneous systems are composed. Furthermore, errors may propagate through a distributed system. Testing approaches that are limited to single systems are not able to capture this propagation, making it difficult to localize the source of a problem.

In order to reason about issues in distributed systems, the system engineer often needs in-depth knowledge about its components and how they interact. Unfortunately, methods that support the comprehension of interaction behavior in distributed systems interactions are rare. VisuSniff [50] is a tool that captures the network communication between systems and visualizes it on-the-fly or offline in a sequence diagram. The goal of VisuSniff is to increase the understanding in network communication. The authors have recognized that the network traffic is cryptic and it is virtually impossible to effectively analyze it without appropriate tool support. VisuSniff is limited as it only directly displays the information that is contained in the packets but does not visualize the messages that are exchanged on application level. Other network analysis tools such as Wireshark [52] capture network traffic and simply present the headers and contents of packets. However, this leaves the user with decrypting the application level information that is contained in the packets, which is time consuming and requires in-depth knowledge of the protocol. A more high-level approach uses plots that illustrate behavioral data of distributed systems to identify anomalous behavior [44]. The approach is limited to systems that interoperate via CORBA and provides only little detail about the interaction events. The tools for supporting integration and maintenance activities

are rudimentary. They operate on the network level and are, therefore, insufficient for reasoning about distributed systems on an application level. There is a clear need for an approach that focuses on the distributed system as a whole. Especially the integration and maintenance activities are not sufficiently supported. Tools to derive an understanding of a distributed system behavior are needed in order to reason about its functional and non-functional properties.

Chapter 3

Behavioral Reflexion Models

In the previous chapters, we have described what architecture representations are produced throughout the software lifecycle and have described the problems that arise if these representations are inconsistent. Furthermore, we provided the basic terminology and discussed related works from the areas of software architectures, compliance checking, and distributed systems. In this chapter, we describe how the behavioral reflexion model approach helps to identify and resolve inconsistencies among architecture representations. We will first provide an overview of the behavioral reflexion model approach and then present a summary of the individual steps involved.

Various representations of software architecture are produced throughout the software lifecycle. The software architecture has a mental model of the architecture that is expected or desired to be implemented: the *intended architecture*. The *documented architecture* serves as high-level design specification that can be used to comprehend the system when conducting integration and maintenance tasks. The *implemented architecture* is perhaps the most important architecture representation as it determines the system's non-functional qualities, such as performance and reliability. All architecture representations must be in agreement in order to achieve confidence in the system's compliance with desired quality goals. Furthermore, the

documented architecture can only fulfill its role of providing comprehension if it accurately describes the system's implemented architecture.

Since the architecture documentation coexists independently from the implementation, however, consistency among them can only be achieved if the architecture is carefully maintained and architecture updates are applied to all of its representations synchronously. In practice, time constraints and the pressure to add functionality rather than maintaining software quality make the maintenance of software architecture a daunting task. As a result, software architecture representations frequently start diverging already during implementation and continue to grow apart during maintenance as additional changes are implemented. The result is architecture documentation that does not accurately reflect the system implementation. That leaves the system engineer without confidence that the system adheres to the desired quality attributes. Furthermore, it renders the architecture documentation ineffective for guiding integration and maintenance tasks.

Software architecture compliance checking has been proposed as a method for addressing inconsistencies among architecture representations in a reactive fashion. After the system has been implemented and architecture documentation has potentially drifted apart from the implementation, architecture compliance checking can be employed to identify inconsistencies among them. The architecture specification is first formulated in the notation used by the respective compliance checking approach. That notation could be textual rules [57] or graphical models [45]. Architecture information is then extracted from the system implementation. The kind of information that is extracted depends on the characteristics the compliance checking

approach aims to verify (e.g., the ordering of interactions). The extracted information is finally compared to the architecture specification and discrepancies between the two are identified.

Various compliance checking approaches have been proposed for identifying structural deviations between documentation and implementation. These approaches identify inconsistencies in the dependencies between components [35] and violations of component access rules [42]. These approaches can be used to identify structural problems but do not take into account an essential characteristic of every system and its architecture, the behavior. The behavior describes how the architecture components interact with each other during runtime. Behavioral inconsistencies between architecture documentation and implementation can have severe consequences for the quality of the system and the usefulness of the documentation. The interaction behavior is an especially critical aspect of distributed systems. Components form one system by collaborating via message passing through networks. Only if all components adhere to the behavioral rules defined in the architecture documentation can the system achieve the desired performance and reliability. The behavior also plays a crucial role in the integration and maintenance of distributed systems. When adding a new component to the distributed system, the system engineer must ensure that the component adheres to the interface behavior of the legacy components it collaborates with. Only with sufficient knowledge about the interaction behavior is a successful integration possible. Accurate architecture documentation is necessary to derive that knowledge.

We have developed an approach for identifying inconsistencies among archi-

tecture representations. The approach is targeted towards distributed systems, as behavioral inconsistencies have been a cause for severe problems in their integration and maintenance [75]. The goal is not only to align the architecture documentation with the implementation but make both consistent with the *intended architecture*. The intended architecture is the mental model of the architecture that the developer had in mind when implementing the system. The intended architecture represents, thus, the desired state of the software architecture. Several reasons prevent both the documented as well as the implemented architecture to comply with the intended architecture. After the architecture documentation has been created prior to the implementation, it often remains unaltered, while the intended architecture evolves as new features are added and existing features are changed. Due to errors that are introduced during development, the implemented architecture also deviates from the intended architecture. As a result, neither the documentation nor the implementation represents the desired state of the architecture. In our approach, an accurate model of the intended architecture is first constructed. The model can be used to update the architecture documentation by either manually adjusting existing documentation or adding the derived model of the intended architecture to the documentation. The implementation is then compared to the model of the intended architecture and deviations are identified.

3.1 Reflexion Models

Our approach is based on the reflexion model approach by Murphy et al. [45] for identifying structural deviations between implementation and architecture specification. In that approach, the system engineer manually specifies the desired architecture in a so-called *high-level model*. A high-level model consists of components and directed dependencies between them. An example of a high-level model is illustrated in Figure 3.1. Each box represents a component and the lines represent the dependencies.

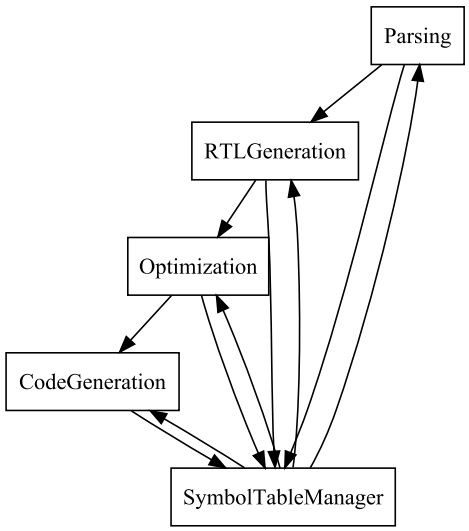


Figure 3.1: Structural high-level model [45].

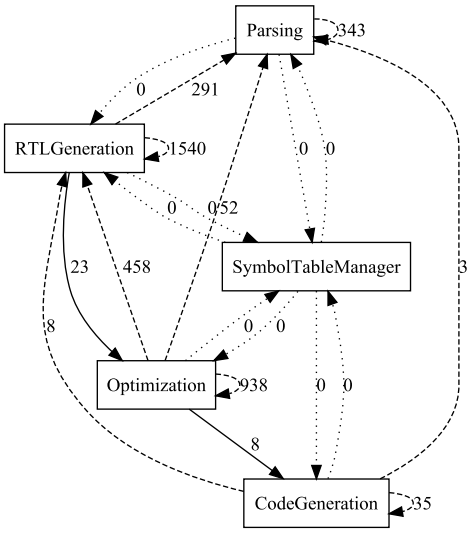


Figure 3.2: Structural reflexion model [45].

After the user has specified the high-level model, the source code of the system is parsed and a *source model* is extracted. The source model describes the structure of the source code with its source code units (e.g., classes) and dependencies between them. In the next step, the system engineer maps the source model to the high-

level model. That mapping is established by assigning one or more source code units to a component in the high-level model. With the source code model mapped to the high-level model, the reflexion model is computed, which illustrates deviations between the two models. The reflexion model that is computed with the source-code and high-level models above is shown in Figure 3.2. Three types of dependencies are shown. The dashed lines represent *divergences*, which are dependencies that were not specified in the high-level model but that exist between the respective source-code units. Dependencies that are specified in the high-level model but do not exist in the source-code model are referred to as *absences*. Solid lines in the reflexion model represent *convergences* and indicate dependencies that exist in both models.

The structural reflexion model approach has shown to be successful in guiding the maintenance of systems [66]. With the graphical representations of deviations between source code and high-level model, the system engineer can reason about them and initiate actions to resolve problems in the system or adjust the documentation with an accurate structural model. Comprehension is also helpful and necessary when analyzing the behavioral compliance of systems to the intended architecture. A failure of the system to provide the expected reliability is often the result of not one but a series of behavioral errors. Comprehension is necessary to identify and reason about the errors that lead to a failure. Furthermore, an interaction error caused by one component might propagate through the system and affect other components and their interaction. The system engineer must comprehend the relationship between errors that occurred at different points of the system to identify the responsible component.

3.2 Behavioral Reflexion Models

We have developed the *behavioral reflexion model approach* for identifying discrepancies among architecture representations. We are focusing in particular on representations of the interaction behavior that occurs between components of a system. Our approach follows the basic idea of the structural reflexion model approach while introducing a sensitivity to behavior.

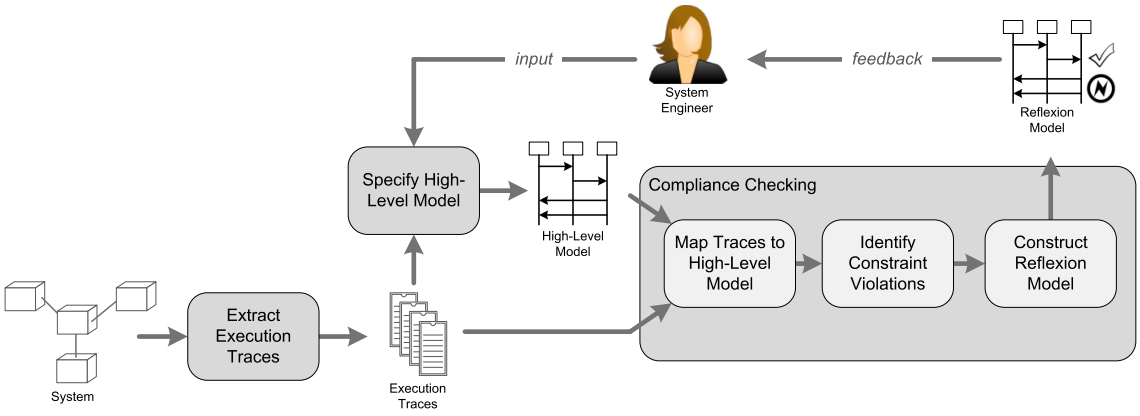


Figure 3.3: Overview of behavioral reflexion model approach.

An overview of the main steps of the behavioral reflexion model approach is illustrated in Figure 3.2. First, execution traces are extracted from the system implementation that describe how components interact during runtime. Second, the system engineer specifies the high-level model, which describes the intended architecture. Third, the extracted behaviors are compared to the high-level model and a reflexion model is constructed that illustrates discrepancies between the two. The compliance checking step can, similar to Murphy's approach, be divided into three steps. In the first step, the behaviors are mapped to the high-level model.

After the mapping has been established, the behaviors are checked for violations of constraints defined in the high-level model. Finally, a graphical reflexion model is constructed that illustrates the violations in the context of the overall behavior. The following describes these steps in more detail using a simple example.

3.2.1 Extracting Execution Traces

In the first step of our approach, the system engineer collects runtime information by monitoring the system execution. To collect that information, the network on which messages are exchanged between the components of the distributed system is instrumented and the network traffic is recorded. Application-level information is extracted from the network traffic to derive an abstract *execution trace*.

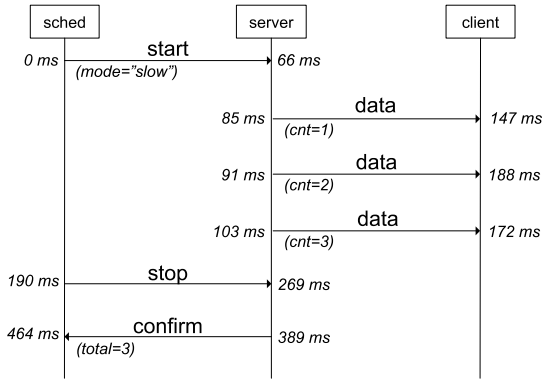


Figure 3.4: Example execution trace.

An execution trace describes a sequence of interactions that occur between a set of components. It captures three behavioral dimensions: *sequencing* of interactions, the *timing* of events, and *data* parameters that are exchanged as part of messages. An execution trace can be illustrated graphically as shown in Figure 3.4.

The diagram shows the execution trace in UML Sequence Diagram notation [10].The components are represented by lifelines and the interactions are illustrated as arrows. Each interaction denotes the transmission of a message from a sending component to a receiving component. Each message has a type and a set of parameters. The type indicates the role of the message in the execution. For example, the type "stop" indicates that the transmission of data should be terminated. Typically, these types are specified in the documentation and used by system engineers to communicate issues related to the interaction behavior. Each parameter carries data that is transmitted along with the message. It consists of a name and a value. For instance, the message "start" has a parameter with the name "mode" and the value "slow". Furthermore, the instance in time in which the message was sent and received annotates the respective ends of the interaction arrow. For example, the message "stop" was sent 190 ms relative to the beginning of the behavior and it was received 79 ms after that.

An execution trace describes the interaction behavior a system exhibits when executing a single test case or scenario. Typically, one would execute an array of scenarios to capture the behavior of certain parts of the system. When capturing execution traces during system testing, the system engineer may derive a set of traces that sufficiently capture the entire system. In other instances, it might be beneficial to only analyze a certain part of the system. For instance, to ensure that the system still behaves as desired after a change has been implemented, the system engineer might only cover the part of the system that was affected by the change.

3.2.2 The High-Level Model

In the second step of our approach, the system engineer specifies a high-level model that describes the intended architecture. In the structural reflexion model approach, the high-level model consists of components and dependencies. The behavioral high-level model describes the set of valid behaviors via constraints on sequencing, timing, and data characteristics. The behaviors that are exhibited by the system must comply with these constraints. A high-level model can also be illustrated in sequence-diagram notation as shown in Figure 3.2.2. The diagram specifies a set of components and messages that are exchanged between them. *Sequence constraints* are defined using the vertical alignment of message arrows and the advanced sequencing operations *loop* and *alternative*. Loops denote the repetition of a message sequence, while alternatives indicate branching behavior. In addition, the user specifies timing and data constraints. *Timing constraints* define rules on the time that may elapse between two events. For instance, the timing constraint "t $\leq$ 30s" expresses that the sending of the *stop* message may not occur later than 30 seconds after the sending of the *start* message. Timing constraints are crucial as many distributed systems are also real-time systems whose correct function depends on the timely occurrence of events [68] . *Data constraints* may be used to describe properties of a single parameter. For example, the constraint "mode {"slow","fast"}" indicates that the parameter "mode" may only take the values "slow" or "fast". Many system failures are due to components formulating the queries to other components incorrectly by sending information that cannot be

interpreted by the receiving component. The data constraint can be used to identify such problems. Data constraints can also specify a relation between multiple parameters. The constraint "cnt $\leq$ total" describes such a relation.

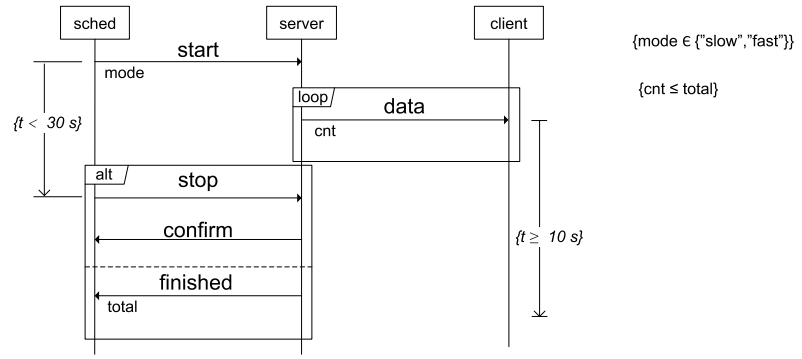


Figure 3.5: Example behavioral high-level model.

The high-level model describes the intended architecture. In practice, it is often difficult to find or construct a model of the intended architecture as both the documentation and the implementation deviate from it. We have developed a method to support the specification of the intended architecture. In that method, the captured execution traces are used to infer clues about sequencing, timing, and data characteristics that are potentially part of the intended architecture. Machine-learning algorithms are then employed to extract a model that describes sequencing constraints as a Finite State Machine (FSM). Each edge in the FSM is annotated with a measure of confidence that that transition is indeed part of the intended architecture. That measure is based on the frequency with which that transition occurred in the execution traces. The system engineer can control the construction of the FSM and also manually add and remove transitions. Machine-learning algo-

rithms are also used to infer suggested timing and data constraints. The constraints are based on the values in the execution trace and additional threshold parameters. The system engineer controls the threshold parameters and can accept, reject, or modify constraints returned by the machine-learning algorithm. With this method, the system engineer can construct a high-level model that describes the intended architecture despite inaccurate documentation and a buggy implementation.

3.2.3 Mapping

Before the behaviors can be checked for violations of constraints specified in the high-level model, a mapping between the two models must be established. In the structural reflexion model approach, source code units in the source model are mapped to components in the high-level model. In our approach, interactions in the execution trace are mapped to interactions in the high-level model. A simple example of such a mapping is illustrated in Figure 3.2.3.

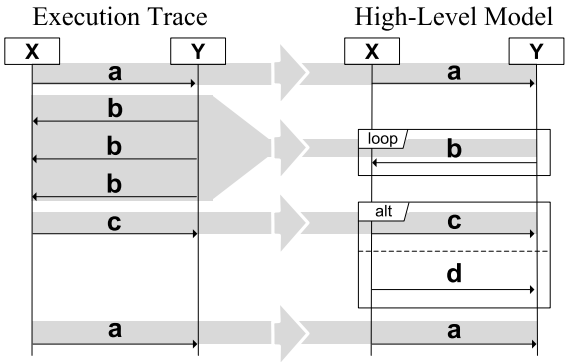


Figure 3.6: Mapping of execution trace to high-level model.

The gray shading indicates the mapping of each interaction (represented by an

arrow) in the execution trace to an interaction in the high-level model. The mapping is based on the name of the message as well as the position of the interaction in the trace. In the structural reflexion model approach, the user manually maps source code units to high-level components. A manual mapping is, however, not feasible in our approach since a potentially large number of execution traces have been collected and each execution often consists of a large number of interactions. We have, thus, developed a method to automate the mapping of execution traces to the high-level model.

3.2.4 Compliance Checking

With the mapping, the interactions in the execution trace can be checked for compliance with constraints specified on the interactions in the high-level model. In an automated process, violations of sequencing violations are first identified and recorded. Subsequently, the execution trace is checked for violations of timing and data constraints. That is, the timing and parameter values in the execution trace are checked to identify if they cause a violation of any of the constraints. The result of the verification is a list of violations that were detected. Each violation carries information about the nature of the violation and contains a reference to the element(s) in the execution trace as well as the high-level model that caused the violation. The references are needed when constructing the reflexion models. The verification is applied for each execution trace individually.

3.2.5 Constructing Reflexion Models

The reflexion model illustrates the deviations that were found during the verification step graphically. Figure 3.2.5 shows an example reflexion model that is constructed based on the execution trace in Figure 3.4 and the high-level model in Figure 3.2.2.

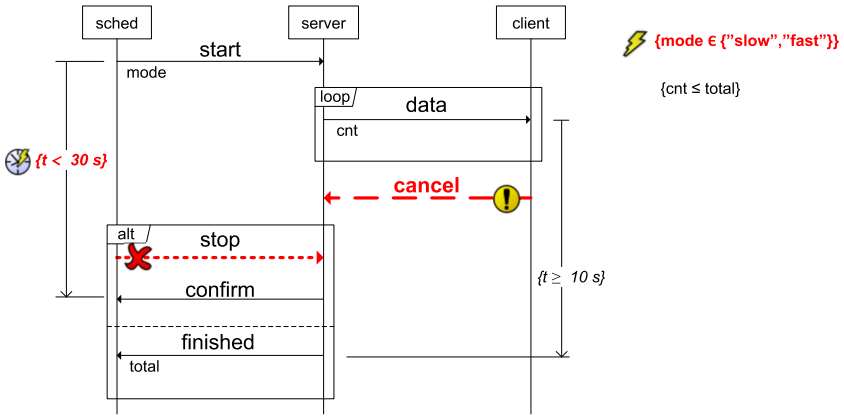


Figure 3.7: Example behavioral reflexion model.

The reflexion model resembles the high-level model. Violated sequencing, timing, and data constraints are highlighted. Timing violations are annotated with a clock symbol. The timing violation in the example reflexion model expresses that in one of the behaviors that was exhibited by the system, the time between the two respective events exceeded 30 seconds. Data violations are annotated with a lightning bolt, indicating that the values of the respective parameter in the observed behavior violated the constraint. The "mode" parameter of the behavior shown in Figure 3.4 has the value "medium", which is not considered valid. Finally, we distinguish two types of sequencing constraints. *Insertions* are interactions that were not specified

in the high-level model but occurred in the behavior. These are annotated with an exclamation mark and the interaction arrow is drawn as a dashed line. Conversely, the arrow drawn with a dotted line and annotated with a cross indicates an *deletion*, which is an interaction that was specified in the high-level model but did not occur in the behavior.

The reflexion model provides an overview of all violations that were identified. Even if comparing many behaviors against the high-level model, the reflexion can provide insight into how well the behaviors match the high-level model. However, the reflexion model shown in Figure 3.2.5 lacks some details that are necessary to further investigate the detected violations. For instance, it is not shown what concrete values violated the timing and data constraints. We developed three types of reflexion models varying in detail and the amount of information that is displayed. We also describe how the different models can be in the analysis of behavioral issues.

3.2.6 Summary

The behavioral reflexion model approach represents a means to align the documented architecture with the implemented architecture. In addition, it helps to establish consistency of both architecture representations with the desired state of the system as represented by the intended architecture.

The process of applying the reflexion model approach is often an iterative one. Typically, the system engineer fist constructs an initial using the automated support. She then identifies discrepancies between the behaviors and the initial high-level

model. Using the graphical representation of the violations in the reflexion model, the system engineer can assess whether the violations are indeed significant. If that is the case, she can further analyze the violations and resolve the problems. In the violations do not appear to be significant the system engineer adjusts the high-level model by modifying the parameters of the mining algorithms or by manually modifying constraints. After a few iterations, the system engineer is able to not only reconstruct a desirable model of the intended architecture but also identify significant violations in the system behavior. For details on the application of the behavioral reflexion model, see Chapter 7.

The following chapters discuss each of the steps of the behavioral reflexion model approach in detail. Subsequently, an evaluation of our approach is presented where we describe its application in two case studies.

Chapter 4

Trace Extraction

As the first step of the behavioral reflexion model approach, the system engineer extracts a set of execution traces from the implementation. An execution trace describes the interaction behavior that is exhibited during a single system execution. In this chapter, we describe methods that exist for extracting runtime information from centralized systems and present a method that we have developed for extracting execution trace from distributed systems.

Extracting runtime information from implementations is part of many reverse engineering [16] and verification [24] approaches. In such approaches, the system is generally instrumented and the system is executed. Events are logged as the system executes. Subsequently, the information is further processed either for the purpose of identifying implementation errors or to build abstract models of the system's behavior.

An array of approaches exists to retrieve runtime information from centralized systems. The approaches differ in the technology they employ to monitor the system and the types of systems they can operate on. A common approach for monitoring the system is to insert statements into the source code that, when executed, produce runtime events. While the approach is simple, it requires modifying the source code; a process in which bugs might be introduced and the runtime of the behavior might

be changed. Less intrusive alternatives are instrumentation of the debugger [72] or the runtime environment [43]. Extraction approaches have been presented for different programming languages and runtime environments. In order to apply the behavioral reflexion model approach to centralized system, the system engineer can choose the extraction method that best suites the system and preferences. For a detailed discussion of available approaches, we refer to [12]

In our research, we are particularly concerned with distributed systems as inconsistencies in behavioral representations have led to severe problems in their reliability and maintainability. Unfortunately, the methods for extraction runtime information from distributed systems are limited. Several approaches exist for the analyzing distributed systems that are implemented using RMI or CORBA (e.g., [12]). However, these techniques cannot be applied to systems that employ traditional message passing as their means of interaction. At the same time, we have identified a need for ensuring proper behavior of such systems as well as accurate behavioral documentation.

To allow the application of the behavioral reflexion model approach, we have developed a technique for extracting runtime information from such message-based systems. The following will first provide an overview of our extraction technique and then elaborate on each of the steps in more detail.

4.1 Approach Overview

The goal of our extraction method is to monitor a distributed system during runtime and construct an execution trace that describes the observed interaction behavior. Similar to other extraction approaches, we instrument the system and observe the execution via instrumentation probes. However, unlike traditional extraction approaches, we instrument the network rather than the source code or the programming environment. When the system is executed, the network traffic that passes through the instrumented part of the network is captured. The captured network traffic is then processed to (1) extract application-level information and (2) construct an execution trace.

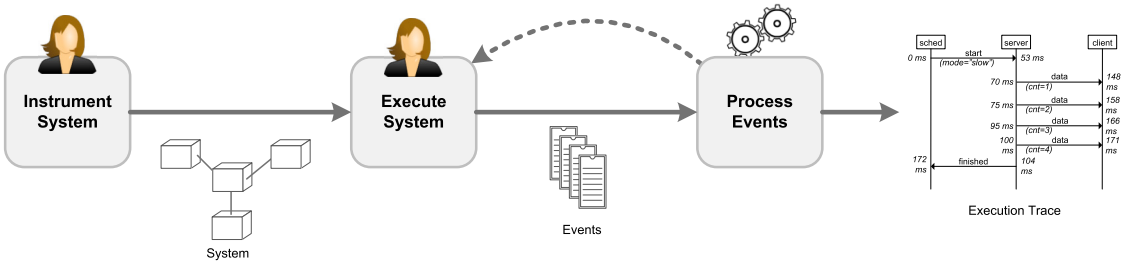


Figure 4.1: Trace extraction process overview.

Figure 4.1 illustrates an overview of the trace extraction approach. The system engineer instruments the system by placing a so-called *observation point* at the network interface of each component in the system. When the system is executed, each observation point monitors the messages that pass through it. More specifically, events are recorded that denote the sending or receiving of these messages by the respective component in the system. After the system has been executed and

the events have been recorded, application-level information is extracted from the observed low-level messages. Finally, a single execution trace is constructed from the recorded events, which shows the overall interaction behavior.

The system engineer instruments the system only once but executes it multiple times. For every execution, events are collected and an execution trace is constructed. The system engineer repeats the execution until a satisfactory set of execution traces has been collected.

4.2 Monitoring

In traditional extraction approaches, instrumentation is inserted in either the source code or the application environment. Runtime information is then collected via these instrumentation points as the system executes. In our approach, we instrument the network rather than the components. More specifically, so-called *observation points* are placed on the network interface of each component in the system to observe the messages that are sent and received by that component. Whenever a message passes through an observation point, an event is recorded that denotes the sending or receiving of that message. Formally, an event is a tuple e = *(index, time, type, location, sender, receiver, m)*, where

index is the logical position of the event in respect to other events at the same observation point,

time is the time when the event was observed,

type is the type of the event that was observed (i.e., send or receive)

location is the component at which the event was observed,

sender is the component that sent the message,

receiver is the component that received or receives the message,

m is the message that is observed.

An example of event traces that are captured during a single system execution is shown in Figure 4.2. The distributed system consists of three components: a "scheduler", a "server", and a "client" component. The lines illustrate the network and the dots the observation points that are placed at each component. Below each observation point is the set of events that was recorded by the respective observation point during a single execution of the system. Each event is marked as either a send- (snd) or receive-event (rcv) and with the time at which it was observed. The symbol "m" represents the message that was observed.

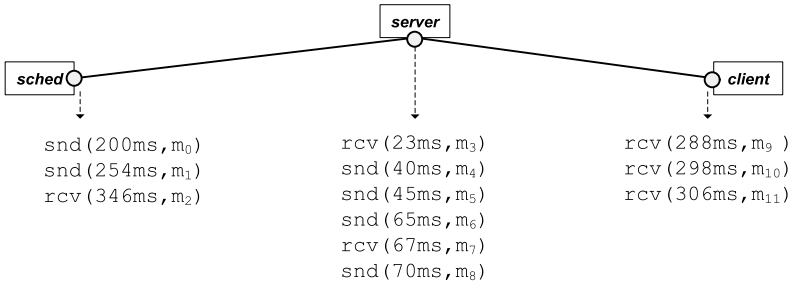


Figure 4.2: Event traces captured at observation points.

Various tools already exist to capture such event information. In our research, we have used UNIX Snoop, Wireshark [52], and bro [55]. We discuss such approaches in more detail in Chapter 7.

4.3 Extracting Application-Level Information

A message that is transmitted between components is first prepared by the sending application to be transmitted across the physical network. The application program specifies the type of the message and provides a set of parameters that should be sent along with a message. It then passes that information on to the communication program that is responsible for transmitting the message reliably across the network [68]. That program encodes the message name and the parameters in a pre-defined format. The message is then emitted to the network, which delivers it to the receiving component. The receiving component must process the message to decode the message type and parameters.

When monitoring the communication traffic via observation points, the messages that are observed have been prepared for transmission by the sending component. In order to derive the application-level information, i.e., message type and parameters, the messages must be processed after they have been captured. The process of extracting information from network-level messages is also referred to as *dissecting*. A message that is observed on the network is a sequence of bytes. The bytes encode the message type and parameters as a set of fields. Each field has a certain length (i.e., a certain number of bits). The byte sequence must be parsed to identify what parts of the sequence represent the value of what parameter. Subsequently, the binary value must be converted into a numerical or textual parameter value. The message type is encoded in one of the fields and is retrieved in the same manner.

The size of the fields as well as a mapping that describes how the binary values must be interpreted is specified in the *message format specification*, which is part of the protocol specification. The message format specification describes for each message type the set of parameters it can carry and how they are encoded. The sender uses the message format specification to encode message type and parameter and the receiving component uses the same message format specification to decode that information. An example for a message format specification of a single message type is shown in Figure 4.3. Each block represents a field with a name and a length. Each field also has mapping between binary values and application-level values. For instance, the table below the "mode" field indicates that the binary value "01" translates to the value "fast".

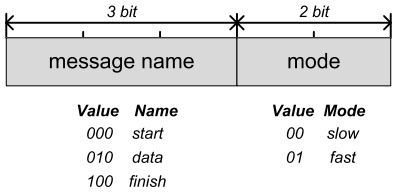


Figure 4.3: Example message format specification.

Every byte sequence (i.e., message) that is captured is parsed to identify the fields and then interpret their binary values. Various tools exist that support the dissection of network-level messages (e.g., [55][62]). In order to extract application-level information from messages, we use the NetPDL library [62]. The message format specification must be given in the NetPDL format, which is an XML-based notation. The NetPDL library takes that specification and dissects the captured

byte sequences. For each byte sequence that represents a message, it retrieves the message type and the set of parameters.

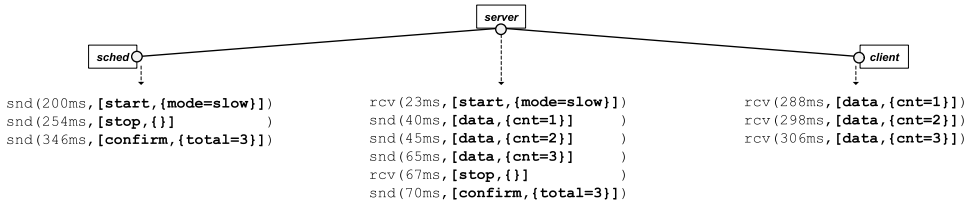


Figure 4.4: Events with application-level messages.

By dissecting each message, the network-level messages are converted into application-level messages. Figure 4.3 shows the same events as illustrated in Figure 4.2. The message types and parameters have been extracted (indicated in bold font).

4.4 Constructing the Execution Trace

After the application-level information has been extracted from the messages, an execution trace is constructed from the observed events. First, using the information collected at the observation points, a set of interactions is inferred. Subsequently, the timestamps of the events are adjusted to account for clock-drift and a total ordering among the interactions is established.

4.4.1 Constructing Interactions

An execution trace consists of a sequence of interactions that occur between a set of components. An interaction denotes the sending and receiving of a message.

In order to construct a single interaction, the send-event and the receive-event that correspond to the same message are identified. In order to identify corresponding events and construct interactions, the events of all components $c_i, c_j \in C$ are aligned as follows.

First, the events of both components are filtered to remove all events that do not mark messages that are exchanged between c_i and c_j. That is, all events, in which the sender or the receiver is not c_i or c_j are removed. As a result, only events are retained of messages that were exchanged between the two components. We assume that messages are received in the same order in which they are sent. Given that assumption, the events align such that every event at c_i appears at the same position as its corresponding event at c_j. An example is illustrated in Figure 4.4.1. It shows the events at the "scheduler" and the "server" component. Send-events are shown as white dots while receive-events are shown as black dots. Events that have been removed are indicated by the gray shading. The example shows that after filter out irrelevant events, the remaining events that belong to the same message appear at the same position in the event traces. For instance, the first event at the "scheduler" component and the "server" component belong to the same message (i.e., "start"). The dashed lines indicate what events line up with each other and, thus, belong to the same message.

For each pair of events e_{snd} and e_{rcv}, at which the same message m was captured, an interaction i is constructed. An interaction i is a tupel consisting of the send-event e_{snd} and the receive-event e_{rcv}:

$$i = (e_{snd}, e_{rcv}).$$

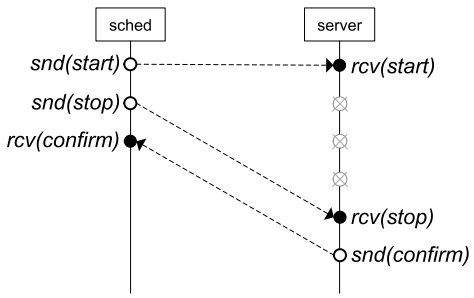


Figure 4.5: Alignment of events.

In order to derive all interactions of the execution trace, the alignment must be made for all pairs of components $c_i, c_j \in C$ and interactions must be constructed for all event pairs. The result of this step is an unordered set of interactions.

4.4.2 Time Adjustment

Components in a distributed system and the observation points monitoring their network interfaces are geographically dispersed. The timestamps that are recorded for the events are based on the local time of the respective component. Generally, each component has its own local physical clock to order local events. However, due to the drifting nature of quartz-controller oscillators, no two ordering nodes run at exactly the same rate [68]. In order to be able to reason about timing across the entire network, the time measurements must be based on a global clock. Therefore, the clocks at the components must be synchronized to provide a system-wide global time. One approach for establishing consistent timing when extracting runtime information is to collect events at a central location. In that case, the clock of the central location determines the event timestamps. Another way is

to use the network to exchange clock synchronization signals among components during operation. Both approaches impose a burden on the system engineer and the components. It requires setting up additional network connections for exchanging synchronization information and/or uses resources of the system that is being monitored. In practice, these solutions are often not feasible.

In order to reduce the effort required to monitor the system in our techniuqe, the clocks are synchronized after the events have been captured. Hofman et al. [31] present an approach for computing a global clock for communicating processes in a distributed system. Based on the fact that message are received after they have been sent and the observed times for send- and receive-events of messages, the algorithm computes a global reference clock. It then determines the offset of each component to that global clock signal.

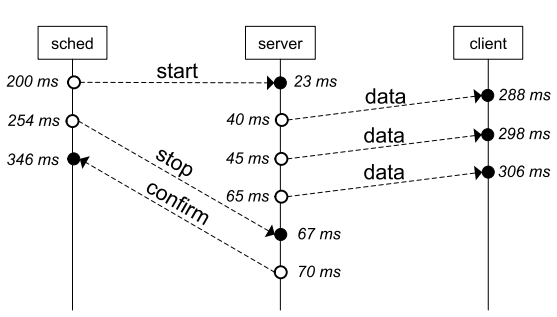


Figure 4.6: Preliminary behavior trace model with inconsistent timing.

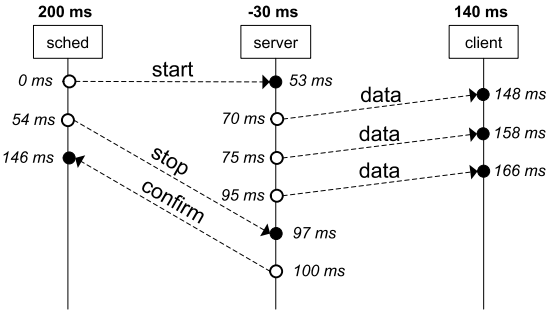


Figure 4.7: Preliminary behavior trace model with consistent timing.

An example is illustrated above. Figure 4.6 shows the preliminary behavior trace model. An example for the inconsistency among timestamps taken at different component is illustrated by the send- and receive-events of message “start”. The

model illustrates that the message was sent at 200 ms and received at 66 ms. Of course, a message is always received after it was sent and the timestamps must be adjusted accordingly. Using Meyers algorithm, a global clock is computed and offsets of local clocks to that global clock is calculated for each component. The offsets are shown above each component in Figure 4.7. The timestamps of the events are adjusted using the offset for the respective component. Figure 4.7 shows the corrected timestamps. The timestamps are consistent across components and all send-events are ordered in time before their respective receive-events.

4.4.3 Ordering of Interactions

A global clock not only establishes consistency in the timing among events but also allows for ordering them. Since the timestamps are consistent across all components, the interactions can be ordered by the timestamps of either their send- or receive-events. The timestamp-based ordering is a partial order since two events may potentially occur at the same time. In order to establish a total order among interactions, we use the idea presented by Lamport [39]. In his approach, a partially ordered set of events occurring at concurrent processes is totally ordered based on priorities of processes. Each process is assigned a priority. Whenever the order of two events cannot be determined, the event of the process with the higher priority is ordered before the event at the process with lower priority.

When instrumenting the system, the system engineer assigns priorities to each of the components. After events have been captured and interactions are produced,

they are ordered based on the timestamp and the priorities of processes. More specifically, the interactions are ordered by the timestamp of their send-event. Whenever two events have the same timestamp, the order is determined based on the component priorities based on the approach by Lamport.

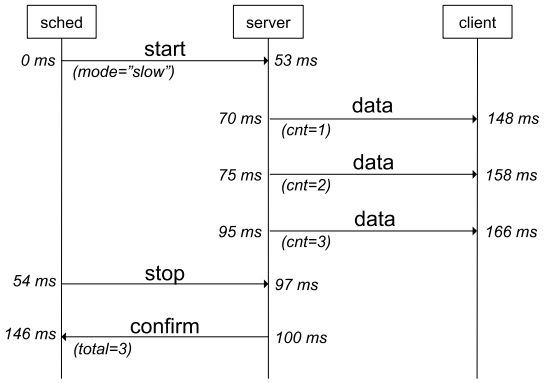


Figure 4.8: Example execution trace.

With that last addition to our approach, we can establish a total ordering among interactions and complete the construction of the behavior trace model. The final example model is shown in Figure 4.4.3. Typically, a system is executed several times and a behavior trace model is constructed for each execution.

4.5 Discussion

In this chapter, we have discusses methods for extracting execution traces from centralized systems and introduced a method that we have developed to extract such traces from a distributed system in which components interact via message passing. By observing the network rather than the components, we avoid ties to programming language or platform. The approach is, thus, suitable for heterogeneous systems

unlike traditional reverse engineering approaches that require instrumenting source code or environment of the target component. While the approach is independent of the component's programming language, the process of retrieving application-level information depends on the protocol. Protocols differ in their message format. In order to be able to extract application-level information by dissecting packets, the system engineer must provide the respective message format specification. However, adopting the extraction to another protocol is not as difficult as adopting traditional reverse engineering approaches to different programming languages. The NetPDL library [62] that is used for dissecting packets, reads the message format specification in a standard NetPDL format. The format is XML based, which is familiar to many system engineers. Furthermore, as the popularity of the NetPDL format increases, we expect the message formats to documented in that notation, essentially eliminating the effort for adopting approach to different protocols.

Similar to most other reverse engineering approaches that construct behavioral models, we apply dynamic analysis to collect behavioral information from the system implementation. That is, the system is executed and information is recorded during runtime. The chief disadvantage of dynamic analysis is that the special attention must be paid to achieving sufficient coverage of the system's behavior. A behavior trace model that is constructed from a single system execution only describes a single behavior. The behavior of the system changes when a different scenario is executed. Depending on the goal of the analysis, the system engineer might choose to cover the entire system behavior or only parts of it. In order to achieve the desired coverage, the system must be executed in specific ways (i.e., with a specific set of

input values). One way to achieve this is to monitor the system during testing. In testing, a system is executed in a way so as to achieve a certain coverage criterion (e.g., line coverage or branch coverage). By monitoring test executions, the system engineer can gain the same confidence in the result of the interaction analysis as provided by the testing or coverage technique.

Chapter 5

The High-Level Model

In the beginning of the behavioral reflexion model approach, a high-level model is specified that describes the intended architecture of the system. Subsequently, the information that is extracted from the system implementation is compared to the high-level model and discrepancies are identified. In this chapter, we discuss the behavioral high-level model in detail. First, we describe the characteristics that are specified in the high-level model and against which behaviors are checked. We then give a notation for specifying these characteristics in a concise manner. Subsequently, we describe a semi-automated approach for constructing the high-level model.

In the structural reflexion model approach, the system engineer specifies a valid structure of a system with its components and dependencies. Source code dependencies are then compared to the high-level model to identify invalid dependencies.

In the behavioral reflexion model approach, the high-level model describes valid interaction behaviors of a system. A behavior is a sequence of interactions that take place between a set of components. As the state of the system changes and different input values are provided, the interaction behavior changes as well. The high-level model specifies the set of interaction behaviors that are considered

valid. In other words, it specifies the valid system semantics.

5.1 Description of the Behavioral High-Level Model

Depending on the focus of the analysis, one may choose to specify different characteristics in a high-level model. In our research, we are aiming to identify violations of the semantics that lead to problems in the maintenance and reliability of distributed systems. During the study of such problems in the context of real-world systems, we have identified the following characteristics to play a major role in a system's proper behavior:

- Sequencing: A large number of protocol verification approaches focus solely on the order of events as it is crucial for the correctness of the system. Messages must be exchanged in the appropriate order to ensure that the information is transmitted reliably and efficiently.

- Timing: Many distributed systems are also real-time systems as certain events must occur within a pre-defined time. Failure of a system to adhere to hard real-time constraints can lead to performance and reliability problems.

- Data: Messages that are exchanged between components often contain a set of parameters. Each parameter carries data that the sending component seeks to provide to the receiving component. Many protocols specify rules to which these parameters must adhere. For instance, it might define that only certain discrete values are allowed for a parameter. Other rules might specify relationships between multiple parameters, such as equality.

In order to be able to identify reliability and performance problems, these characteristics must be captured in the high-level model. Thus, a high-level model should contain a set of rules on sequencing of interactions, the timing of events, and the data that is carried by parameters. A notation that allows for describing rules on sequencing, timing, and data is the sequence diagram as described in the Unified Modeling Language (UML) [10]. We have adapted traditional sequence diagrams for specifying behavioral high-level models.

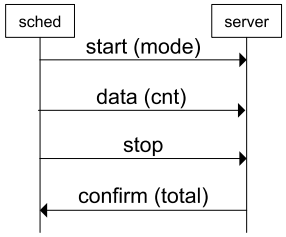


Figure 5.1: Basic sequence diagram.

Figure 5.1 illustrates the basic elements of a sequence diagram. The diagram consists of a set of components and interactions between them. Components are indicated by labeled lifelines. Interactions are shown as arrows that are directed toward the receiver of the message. Each interaction arrow is labeled with the type of the message that is transmitted and a set of parameters in parentheses. Unlike execution traces, parameters in a high-level model do not carry values as they solely describe the message signature and not concrete messages. The following describes how sequencing, timing, and data rules can be expressed using the sequence diagram notation.

5.1.1 Sequencing rules

The order of interactions is specified graphically. A strictly sequential order is expressed by aligning interaction arrows vertically. The topmost arrow represents the first interaction in the behavior. In addition, the UML specifies a variety of so-called interaction operators, with which one can describe more complex sequences. For the context of this thesis, we will limit the sequencing operators to *loops* and *alternatives*. A loop expresses that one or more interactions may be executed multiple times. Graphically, a loop is expressed as a rectangle labeled "loop" (see Figure 5.2) enclosing the respective interactions. The UML also allows for specifying the allowed number of iterations in a loop. However, we will not support that feature. A loop in a high-level model means that the enclosed interactions may occur zero, one, or more times.

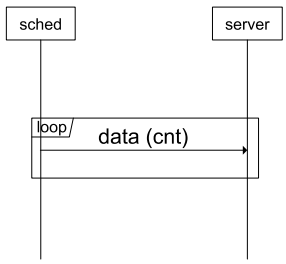


Figure 5.2: Sequence diagram loop construct.

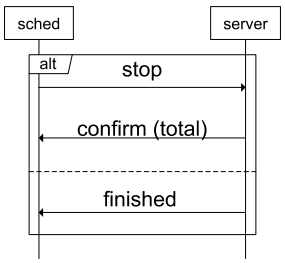


Figure 5.3: Sequence diagram alternative construct.

The second interaction operand, alternative, allows for specifying branching behaviors. At a certain point in the behavior, it might be valid to exhibit different interaction sequences. This can be specified with the alternative operator (see Figure 5.3). UML also provides a notation for specifying a branching condition. For the

context of this research, conditions will not be supported. A behavior is considered valid if it exhibits one of the alternative interaction sequences.

5.1.2 Timing Rules

Timing rules can be specified between any pair of events in the sequence diagram. Two types of events are defined in the high-level model: the event of sending a message (send-event) and the event of receiving a message (receive-event). A rule imposes a relation on the time difference between two events via a so-called *constraint*. A constraint is a Boolean expression, which evaluates to true if the constraint is satisfied and to false if the constraint is violated. The majority of timing constraints in real-time systems define some upper or lower bound on the time that may elapse between the occurrences of two events [68]. We support such constraints:

- Upper bound: An event y must occur no later than a certain time after event x. Example: $t \leq 3s$ (where t is the time difference between the occurrence of x and the occurrence of y).

- Lower bound: An event y may not occur earlier than a certain time after event x. Example: $5s \leq t$.

- Upper/lower bound: An event y may not occur earlier or later than a certain time after event x. Example: $3s \leq t \leq 7s$.

In a distributed system, there are two factors that can affect a system's adherence to timing constraints: the responsiveness of components and the performance

of the network. After a component receives some stimulus from the environment (e.g., the receipt of a message) it must respond within a certain amount of time. A message must arrive at the receiving component a certain time after it has been sent. We refer to a constraint that specifies a threshold on the response time of a component as *delay-constraint*. A constraint that defines a threshold on the duration a message travels on the network as *latency-constraint*.

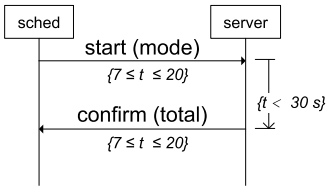


Figure 5.4: Example timing constraints.

Figure 5.1.2 shows example timing constraints. The constraints that annotate message arrows are latency constraints. They specify rules on the time that may elapse between sending and receiving the respective message. A delay constraint is shown to the right of the model. Lines are used to indicate what events the constraint refers to. In the example, the constraint refers to the receive-event of the "start" message and the send-event of the "confirm" message. The time between these two events must be below 30 seconds. The only addition we made to the sequence diagram notation specified by UML is the notation of latency constraints. These constraints are shown below the respective message arrow. While UML does allow for specifying such types of constraint, the notation is rather cumbersome. With this minor extension of the notation, we can concisely specify latency constraints.

5.1.3 Data Rules

Just like timing rules, data rules can also be specified via constraints. Data constraints specify Boolean expressions that specify properties of single parameters or define relationships between two parameters. The data constraints that are supported for the context of this research are:

- Element-of: A parameter may only take a pre-defined set of values. For instance, the parameter "speed" may only take the values "slow" and "fast". This can be expressed with an element-of constraint (e.g., $speed \in \{slow, fast\}$). The parameter may only take the values that are specified in the set.

- Range: For numeric parameters, valid values often lie within a specific range. Such a range can be expressed with a range constraint. For instance, the parameter "length" may only lie between 0 and 100. The constraint $0 \leq length \leq 100$ expresses that rule.

- Equality: The value of two parameters should always be equal (e.g., $cmd\_server = cmd\_proxy$).

- Inequality: Some parameter might represent an upper or lower bound on another parameter. This can be specified with an inequality constraint (e.g., $cnt \leq total$ or $index < total$).

In the sequence diagram notation, data constraints are added as text to the diagram. Similar to the timing constraints, data constraints are enclosed in brackets. The parameter names in a data constraint are used to identify what parameter the

constraint refers to. In other words, the constraint pertains to all parameters with that name in the sequence diagram.

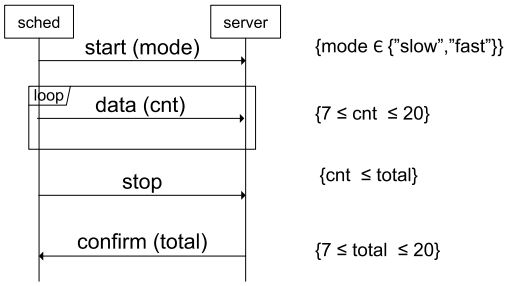


Figure 5.5: Example data constraints.

The example shown in Figure 5.1.3 illustrates data constraints on single parameters and constraints that describe relationships between two parameters. Since the parameter names must be unique, the constraint variables are unambiguously mapped to parameters (and interactions).

5.2 Construction Overview

In the structural reflexion model approach, the system engineer manually specifies the components and dependencies of the high-level model. However, describing a high-level model may be challenging as accurate documentation is often missing and the original developers have left the team. We have developed an approach for constructing a high-level model in which the system engineer is supported by a set of machine learning algorithms. The algorithms infer information about the system semantics from the set of execution traces that were extracted from the system implementation and construct an initial set of sequencing, timing, and data rules.

The rules serve as a starting point for the system engineer when constructing the high-level model. Furthermore, they facilitate comprehension as they describe the semantics of the implementation.

The goal of the machine learning algorithms is to provide the system engineer with an initial set of rules that closely resemble the intended architecture. Creating such rules from behaviors that have been extracted from the system implementation is a non-trivial task. While many reverse engineering tools infer abstract models from implementations, they construct models that reflect the implemented architecture. The high-level model, on the other hand, describes the intended architecture. The implemented architecture differs from the intended architecture as it may contain errors that have been introduced during the implementation and cause the system to exhibit undesired behaviors. A main challenge in aiding the system engineer in the construction process of the high-level model is to provide information that allows her to decide which inferred rules describe desired system semantics and which of the rules are due to implementation errors.

The high-level model is constructed in two main steps. First, a sequencing model is constructed that describes the basic elements of the high-level model (e.g., interactions, events, and messages) and specifies sequencing rules on interactions. In the next step, the high-level model is completed by adding timing constraints on its events and data constraints on its parameters. The following sections describe these two steps in detail.

5.3 Constructing the Sequencing Model

The sequencing model describes in what order interactions may occur. An initial model sequencing model is constructed by applying a machine learning algorithm to the set of execution traces, each of which describes a sequence of interactions. The machine-learning algorithm constructs a model that describes the implemented sequencing semantics as a Finite State Machine (FSM). That model may deviate from the intended architecture as it also describes erroneous executions. In a subsequent step, the system engineer modifies the model to remove the parts that are due to errors. A measure is computed for every transition, its *support*, that expresses a level of certainty that the transition is valid (i.e., not erroneous). The support guides the system engineer in identifying and removing erroneous transitions. Once an FSM has been constructed that correctly describes the valid order of interactions, it is converted into a sequence diagram.

5.3.1 Constructing the Initial Sequencing Model

Several approaches have been proposed in the literature to construct models that illustrate the semantics of a system from a set of event traces (e.g., [49][54]). A widely-used approach is the *k-tail algorithm* [61]. In the k-tail algorithm, event sequences are merged and a Finite State Machine (FSM) model is produced that describes the control flow or sequencing semantics of the input sequences. The algorithm has found application in program analysis and software visualization approaches as a means to compactly describe a large number of execution traces and

to identify the control flow of programs. We have adapted the k-tail algorithm to infer the sequencing semantics of execution traces that have been extracted from the implementation.

When applying the k-tail algorithm for constructing a sequencing model, each trace is first converted into a FSM representation. We will refer to a FSM describing a single execution trace as *individual sequencing model*. The individual sequencing models are then merged to form a single sequencing model that describes all execution traces.

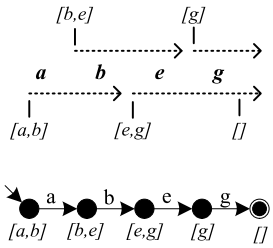


Figure 5.6: Construction of an individual sequencing model.

An execution trace describes a sequence of interactions that can be written as a string in which each symbol represents one interaction. An example is illustrated in Figure 5.3.1. The trace consists of the interaction sequence "a b e g". A sliding window of size $k = 2$ is used to identify all $k - length$ interaction sequences (k-*futures*). For each such sequence, a state is produced that is labeled with that k-future. A transition is inserted between every pair of states that is produced from consecutive sequences. The transition is labeled with the interaction of the first interaction in the k-future of the source state. The state that is labeled with the first k-future is the start state s_0 and a state with an empty k-future is an accepting

state f. The result is a sequencing model that describes the sequencing semantics of a single execution trace. Formally, a sequencing model can be described as a tuple $(I, K, S, s_0, \delta, \gamma, f)$, where

I is a set of interactions,

K is a set of k-futures,

S is a set of states,

$s_0 \subseteq S$ is an initial state,

δ is the state-transition function: $\delta : S \times I \rightarrow S$,

γ is the state labeling function: $\gamma : S \rightarrow 2^K$, and

$f \subseteq S$ are the accepting states.

Each state is labeled with a set of k-futures and each transition is labeled with an interaction label. After the individual sequencing models have been constructed, they are merged to a single model. All states (of all sequencing models) that are labeled with the same k-future are merged to one state. If, as a result, two transitions with the same label emanate from one state, they are also merged. Furthermore, if the merged transitions have different target states, the states are merged as well (i.e., the merging procedure is recursive). Finally, all states that are marked as start states are merged to a single state in order to create a unique start state. Outgoing transitions are merged according to the k-tail algorithm.

An example of the merging procedure is illustrated below. Three execution traces have been observed and converted into individual sequencing models as shown in Figure 5.7. The individual sequencing models are then merged using the k-tail algorithm, resulting in the model shown in Figure 5.8. The transitions are labeled

with the set of interactions and the states are labeled with the respective k-futures. The parameter k in this example is set to 2. The final sequencing model accepts a superset of the execution traces that were used to construct it.

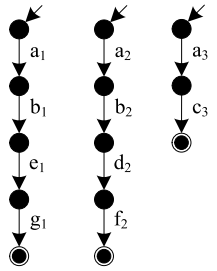


Figure 5.7: Example execution traces.

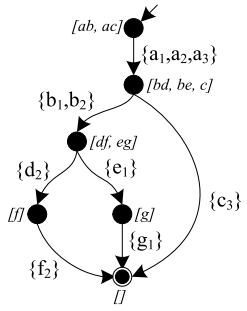


Figure 5.8: Implementation model constructed from the example traces.

By changing the k-value, one can modify the compactness and restrictiveness of the resulting FSM. In our analysis of various systems and their respective specifications, we determined a k-value of 2 to produce satisfactory results for the majority of systems. The same value has been found to produce desirable results in other contexts [61].

5.3.2 Computing the Transition Support

The sequencing model summarizes the execution traces that were extracted from the system implementation and as such illustrates the implemented architecture. In order to derive a high-level model that describes the intended architecture, parts of the sequencing model that are due to implementation errors must be identified and removed. If accurate documentation is lacking, it might be difficult for

the system engineer to manually identify erroneous parts of the model. To provide guidance to the system engineer, a measure is computed that indicates what parts of the model might be due to errors: the *support*. The support is computed for each transition and expresses the certainty that the transition is valid. Transitions with little support are potentially due to errors and the system engineer might decide to remove them.

One way to identify parts of a model that are due to errors has been presented by Raz et al. [60]. In their approach, the model consists of a set of invariants that describe the system behavior. An invariant describes relations that hold among variables throughout the system execution. The model is constructed based on a set of variable values that are observed during runtime. An invariant that is constructed based on a large number of probes is likely to be valid. Conversely, an invariant that is based on only a few observed value sets might potentially be erroneous. Raz's studies have shown that program bugs can be identified via invariants that are based on little evidence. That suggests that the frequency with which relations hold or events occur, can be used as a measure for expressing the likelihood that the event is erroneous.

A similar principle is employed by Ammons et al. [3] to discover formal specifications of protocols for programs interacting with an application program interface. A model is generated using the k-tail algorithm, which illustrates the operations on a program variable, such as read, write, and close. The "hot core" of the model is then identified by removing all transitions that are unlikely to be executed. The authors explain that the hot core, i.e., the parts of the model that are frequently

executed, represents the specification.

We adopt the notion of frequency of occurrence to compute the support for transitions. More specifically, we identify erroneous transitions by the number of traces that execute it. A transition has one or more interactions mapped to it. These interactions are part of one or more traces. Since one trace may execute a transition multiple times, more than one interaction of a trace might be mapped to the same transition. The execution trace count E_t is determined, which describes the number of traces that have interactions mapped to a transition. In other words, it expresses the number of traces that execute that part of the model. The execution trace counts for an example model which was constructed from 300 execution traces is shown in Figure 5.9.

From these absolute counts, the support for a transition is computed as the probability for that transition to be traversed by an execution trace:

$support = |E_t|/|E|$,

where E_t is the set of traces whose interactions are mapped to the transition t and E is the set of all traces. The support for the transitions of the example model of Figure 5.8 is shown in Figure 5.10. The example shows that all execution traces begin with an interaction in which a message with the label "a" is exchanged since the transition has a support of 1. That means that all of the traces (100%) execute that transition. In the majority of traces (70%), this is followed by an exchange of a message "b".

According to Raz's findings [60], parts of the model that describe a behavior, which is frequently exhibited by the implementation are likely to be valid. Con-

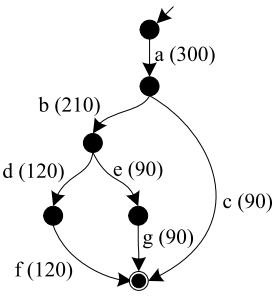


Figure 5.9: Number of traces that execute each transition.

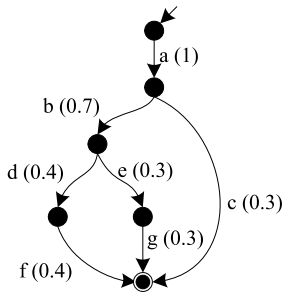


Figure 5.10: Observed frequencies of the example model.

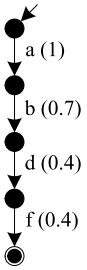


Figure 5.11: Final sequencing model.

versely, transitions that are executed by few traces might be due to errors. The system engineer can use the support as guidance when modifying the model. One approach for deriving a valid sequencing model is to remove all edges whose support is less than a certain threshold. For instance, Figure 5.11 shows the model that is produced when all transition with a support less than 0.4 are removed.

The system engineer must be aware, however, that the support ignores several factors that need to be taken into account when deriving such a model from runtime information. As Musa et al. [46] have described, not all sequences of operations are equally likely to occur in a system. The equiprobability assumption, which specifies that all transition that emanate from the same state are equally likely to occur, does not accurately describe typical systems. Hence, the support is only meant to guide the system engineer and not to replace her. The role of the system engineer is to ultimately judge whether a transition is indeed erroneous and remove it. In our work, the support has shown to be well suited for guiding the system engineer

in this task. further discusses the support in the context of several case studies.

Our notion of support differs from the probabilities used in the approach by Ammons et al. [3] in that we emphasize the number of traces executing a transition rather than the number of times overall a transition is executed. In many of the systems we have studied, the protocol includes a few interactions that are repeated many times (e.g. during transmission of data). When we applied Ammons' approach, the transitions representing the repetitive interactions received a probability of approximately 100% and all other transitions a probability of 0%. Consequently, even valid interactions were identified as anomalous.

We have also studied more complex methods for identifying anomalous transitions, which take into consideration that transitions that are part of branches are expected to be executed less frequently than those transitions that are not. In that method, a transition would receive a higher support if it was part of a branch. However, in our evaluation, we discovered that the more complex measures did not result in more accurate classifications of transitions.

5.3.3 Converting the FSM into a Sequence Diagram

After constructing a FSM sequencing model from the set of captured execution traces, the model is converted into a sequence diagram representation. A regular expression is first derived from the FSM and the regular expression is subsequently translated into a sequence diagram.

Kleene's theorem shows that FSMs and regular expressions are semantically

equivalent. That means that for every FSM, there exists a regular expression that accepts the same language. Several techniques have been proposed to convert a FSM to a regular expression [48]. Examples are the transitive closure approach, the state removal method, and Brzozowski's algebraic method [14]. The latter constructs short regular expressions and is at the same time straight-forward in its implementation. We will, therefore, apply the algebraic method for inferring a regular expression from a FSM. In the algebraic approach, a regular expression is constructed for each state in the FSM. The resulting system of regular expressions is then solved for the regular expression of the start state. When applying the approach to the sequencing model, a regular expression is produced in which each symbol represents an interaction. These regular expressions encode sequencing rules using the operators union $(a + b)$, concatenation (ab), and iteration $(a*)$. Assume that the example model in Figure 5.3.3 has been constructed. The corresponding regular expression for that model would be $ab(df + eg)$.

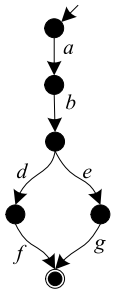
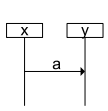
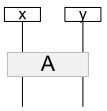
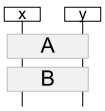
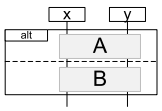
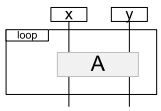


Figure 5.12: Example sequencing model

Similar to a regular expression, a sequence diagram explicitly encodes repetitions and branching behaviors. Every regular expression term can be directly

translated into a semantically equivalent sequence diagram construct as shown in Table 5.1. A single symbol in a regular expression is converted into a single interaction in the sequence diagram. The sequencing operations can be applied to any sequence of interaction (indicated by A and B). The term AB in a regular expression is expressed by ordering B immediately below A in the sequence diagram. The term $A + B$ is illustrated using an alternative operator. Finally, the regular expression term A^* is translated to a loop construct, indicating that the loop may be repeated 0, 1, or more times. The information about the sending and the receiving components can be retrieved from the interactions in the regular expression.

Table 5.1: Conversion of regular expression elements to sequence diagram constructs.

Regular Expression	a	A	AB	$(A) + (B)$	A^*
Sequence Diagram					

In order to produce a sequence diagram, the regular expression is simply parsed and each element in the regular expression is converted into an element in the sequence diagram notation. Note that the regular expression does not describe the sender and receiver of messages or their parameters. However, the symbols in the regular expression are interactions. Each interaction specifies the sender and receiver of the message. Thus, the components of the sequence diagram as well as the direction of the message arrow can be inferred from the symbols.

Figure 5.3.3 shows the semantically equivalent sequence diagram representation of the model illustrated in Figure 5.3.3. The diagram expresses that the first

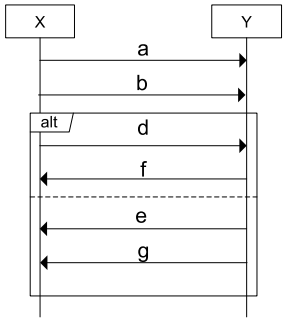


Figure 5.13: Final sequencing model in sequence diagram notation.

two messages must be of type “a” and “b”. It may then be followed by exchanging either the message sequence “d”, ”f” or “e”, ”g”.

5.4 Constructing Timing and Data Constraints

In the next step, the high-level model is completed by adding timing and data constraints. Timing constraints are defined on the events of the high-level model and data constraints are specified on its message parameters. Similar to the construction of the sequencing model, timing and data constraints are inferred semi-automatically. First, an initial set of constraints is automatically inferred. Subsequently, the system engineer adds, modifies, and removes constraints. We first discuss the basic approach on timing constraints and then extend it to data constraints.

5.4.1 Specifying Timing Constraints

Typical timing constraints describe upper and lower bounds on the time that elapses between two events [68]. Constraints are only defined on specific event pairs

that are critical in ensuring the system's proper behavior. When constructing the high-level model, the system engineer must identify those critical constraints. More specifically, the system engineer must answer the following questions:

1. What events are critical?
2. What type of constraint must they adhere to?
3. What is the constant value (i.e., upper and/or lower bound)?

To aid the system engineer in answering these questions, a first set of so-called *default constraints* is inferred automatically from the observed execution traces using machine-learning algorithms. Default constraints describe relations between a set of pre-defined events in the high-level model and have a pre-defined constraint type. Default constraints might not describe critical timing relations and are only suggestions for possible constraints. The system engineer reviews and modifies the default constraints. Additionally, the system engineer may manually add critical constraints that have not been inferred. The following describes how the events, the constraint type, and the constant values are determined for the set of default constraints.

5.4.2 Identifying Events and Constraint Types

In a distributed system, there are two factors that have an impact on the performance of the system: the response time of components and the latency of the network. Only if the components perform as expected and messages are transmitted within a reasonable amount of time can the system overall achieve the desired

performance. Based on this observation, two types of constraints can be identified: *delay constraints* and *latency constraints*. Delay constraints assess the time for a system to respond to some stimulus. Conversely, latency constraints describe timing rules on the time messages travel on the network. Delay and latency constraints are computed as part of the default timing constraint set.

Delay constraints are computed for all instances where a potential response of a component can be observed. A component may respond to a stimulus by executing some internal behavior or by emitting a message to the network. Since only the external behavior is captured, only responses in form of sent messages can be observed. For every message that is sent from a component (response), a constraint is computed that describes the delay from the last observed event at that component. That reference event, or stimulus, can be the sending or receiving of a message. Examples for delay constraints are shown in Figure 5.14. Each event of sending a message is recognized as a response (e.g., send-event of "data") and the preceding event at the same component is assumed to be the corresponding stimulus (e.g., send-event of "start").

Furthermore, latency constraints are computed for all interactions in the high-level model. That is for the time that elapses between the sending and receiving of a message. An example of a latency constraint is shown in Figure 5.15. The latency constraint of the "stop" message indicates that the time that message traveled on the network was between 10 and 200ms.

While in most cases, timing constraints describe upper bounds on the time between two events to ensure a desired system performance, other events may be

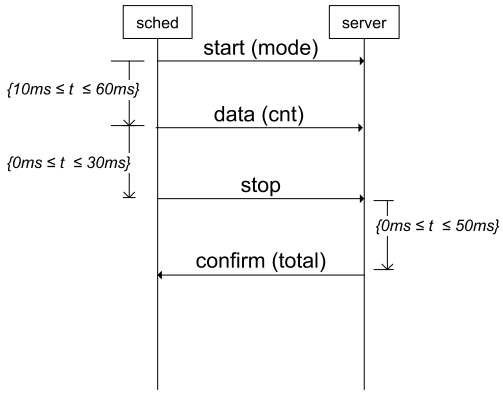


Figure 5.14: Default delay constraints.

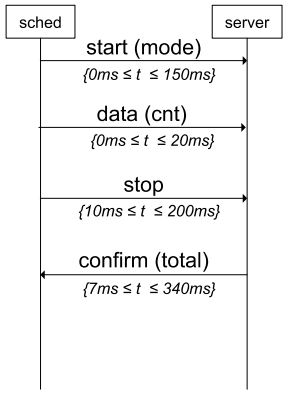


Figure 5.15: Default latency constraints.

constrained not to occur within a certain time. For instance, timeout-recovery event should not occur before a timeout has expired. In such a case, a constraint would describe a lower bound. It is difficult to determine whether the time between two events should adhere to an upper or lower bound. Thus, all latency and delay constraints that are inferred automatically describe both an upper and lower bound.

5.4.3 Constraint Values

The upper and lower bounds are inferred from the set of extracted execution traces. Each interaction in the high-level model has a set of concrete interactions mapped to it. That mapping establishes the connection between high-level model and execution traces. Figure 5.4.3 illustrates a mapping for a high-level model and three execution traces. Each interaction is represented by the message name and the time at which the message was sent. The interactions in the traces are mapped to the interactions in the high-level model.

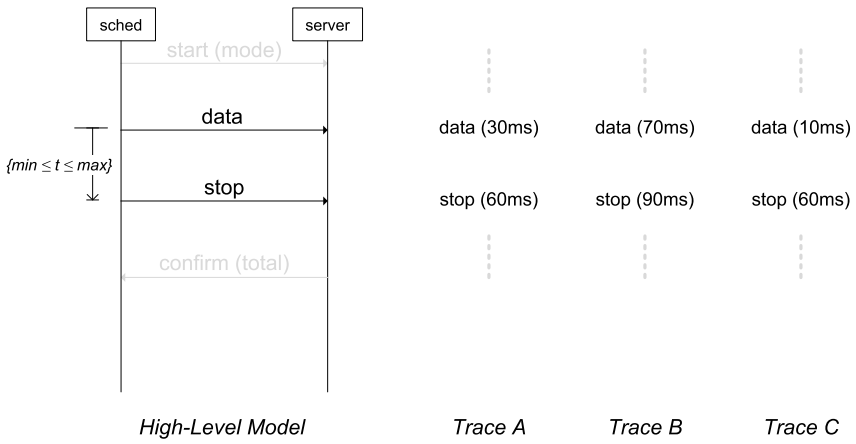


Figure 5.16: Mapping of execution traces to high-level model.

Through the mapping, the events in each execution trace that correspond to the events in the high-level model can be identified. For instance, in the example shown in Figure 5.4.3 a constraint is to be inferred between the send-event of the "data" message and the send-event of the "stop" message. The corresponding events of sending the "data" message t_{data} and of sending the "stop" message t_{stop} in *Trace A* can be identified through the mapping. Subsequently, the delay $t_{delay} = |t_{data} - t_{stop}|$ between the two events in *Trace A* is computed. By computing t_{delay} for each execution trace, a set T of delay values is determined. From that set, the constraint values are determined. The upper and lower bound for the upper/lower bound constraint can be directly obtained as the minimum and maximum value of T respectively. Suppose, the following set of values had been observed:

$$T = \{40ms, 41ms, 42ms, 43ms, 44ms, 45ms, 46ms,$$
$$47ms, 48ms, 49ms, 45ms, 41ms, 42ms, 43ms, 44ms, 45ms, 169ms\}$$

The constraint that is produced based on these values is:

$40ms \leq t \leq 169ms$.

Since the system might deviate from the intended behavior, the values that are observed from the implementation might not describe the intended architecture but rather the implemented architecture. Consequently, the range defined by the inferred constraint may include erroneous values. Invalid values are those that are below the lower bound or that exceed the upper bound of the intended constraint. According to Raz's findings [60], values that are due to errors occur infrequently. Thus, to derive a set of valid values, the values that occur infrequently and lie at the beginning or the end of the value space are removed. That process is also referred to as outlier analysis. An outlier can be described as an observation that deviates markedly from other members of the sample in which it occurs [32]. Outlier analysis has been applied in a wide array of data mining approaches in order to prepare datasets prior to constructing models thereof [8]. Observations that are due to noise or error are removed in order to construct an accurate model. We pursue a similar goal in the construction of timing constraints. A variety of approaches are available to identify outliers, which differ in the type of datasets they perform well on and the types of observations they identify as anomalous.

In order to identify timing values that are due to errors, we employ a technique that has found applicability in the area of Statistical Process Control (SPC) [51]. In that field, it is common to specify the boundaries of valid values using the so-called *three sigma rule*. That rule expresses that the majority of values (99.9%) in a normal distribution lie between $\mu \pm 3\sigma$, where μ is the mean and σ the standard deviation of a value set. Using that rule, a threshold can be identified for the smallest and the

largest valid value in the set of timing values T. By applying the three sigma rule to the example above, we can compute a minimum threshold of -49ms and a maximum threshold of 157.93ms. All values that are lower than the minimum threshold or larger than the maximum threshold are considered outliers and removed from the value set. The only value that is removed in the example set is 169ms. After the outliers have been removed, a new constraint can be constructed:

$40ms \leq t \leq 49ms.$

Since the potentially erroneous values have been removed from the value set, the constraint reflects the valid set of values, i.e., the intended architecture. Similar to the sequencing model, the default constraints represents only suggestions to the user for possible constraints and serve to aid comprehension. The minimum and maximum thresholds identified by the three-sigma rule might exclude values that are in fact valid. Hence, the system engineer must inspect the constraints and modify them if necessary. Also, the system engineer might add additional constraints besides delay and latency constraints that are inferred automatically.

Similar to the support in the sequencing model, we also compute a measure for timing constraints to guide the user in modifying the high-level model with the goal to derive the intended architecture. The measure we compute is called the *strength*. The strength of a constraint expresses the fraction of timing values from the original set T that is included in the constraint:

$strength = |T_{valid}|/|T|,$

where T_{valid} is the set of values that are classified as valid by constraint c, and T is the set of all values that are checked against constraint c. In the constraint

shown above, 17 values had been collected. Only one value is excluded from the final constraint (169ms). Thus, the strength of the constraint is $16/17 = 94\%$. That value indicates that few erroneous values have been detected. We have found that constraints with a high strength to be more accurate than constraints in which many values have been identified as erroneous. One of the reasons might be that the three sigma rule only performs well on normally distributed data. If the timing values are not normally distributed, the constraint that is computed might exclude many valid values. The strength illustrates whether a constraint might need to be manually adjusted. See Chapter 7 for a discussion on the quality of the constraints that are inferred using our approach.

5.4.4 Data Constraints

Default data constraints aim to describe a valid set of values for each parameter. Protocol specifications often describe such a set of allowed values. It is essential that components adhere to the values in order to ensure proper system behavior. Default parameters are constructed to identify and describe discrete and continuous sets of allowed values. Discrete values are described by element-of constraints while continuous value sets are described by lower/upper bound constraints.

The inference of data constraint values is similar to the inference of timing values. A set of values V is determined that contains all values that were observed for that parameter in all of the execution traces. A constraint is then inferred in several steps. First, an element-of constraint is inferred where possible. More specifically, if

the number of distinct values in V is less than a certain threshold, the parameter is identified as discrete valued and an element-of constraint is produced. This applies to both numerical and textual parameters. If the number of distinct values exceeds the threshold for a numerical parameter it is identified as having continuous values and a lower/upper bound constraint is constructed. No constraint is produced for textual parameters whose distinct values exceed the threshold. Examples of data constraints where the threshold is set to 3 are shown below. A parameter w carries numerical values. The values for that parameter that can be observed from the extracted execution traces contain only three distinct values 2, 3, and 4. Since that does not exceed the threshold, an element-of constraint is produced. Similarly, a textual parameter x carries few distinct values and an element-of constraint is produced for it as well. Parameter y, however, exceeds that threshold, which is why a lower/upper bound constraint is constructed. Similarly, the textual parameter z has more than three distinct values and no constraint is constructed.

$w : \{2, 3, 2, 3, 2, 2, 3, 4, 3, 2, 2, 3\} \rightarrow w \in \{2, 3, 4\}$

$x : \{'c', 'c', 'a', 'c', 'c', 'c', 'c', 'b', 'c', 'c', 'c', 'c', 'a', 'c'\} \rightarrow x \in \{'d', 'b', 'c'\}$

$y : \{40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 41, 42, 43, 44, 45, 169\} \rightarrow 40 \leq y \leq 169$

$z : \{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n'\} \rightarrow$ no constraint

Similar to the initial set of timing constraints, the data constraints reflect the implemented architecture as they describe the values that were observed in the implementation. The constraints must be modified to remove outliers and reflect the intended architecture. For discrete parameters, all values that occur less than

a certain threshold are removed. Consider the example of parameter y above. The original constraint is $x \in \{'d', 'b', 'c'\}$. The value 'a' occurs 2 times, the value 'b' occurs 1 time, and the value 'c' occurs 11 times. We have identified a threshold of $0.02 * |V|$ to result in satisfactory constraints. With that threshold, the values 'a' and 'b' are identified as outliers and removed as they are potentially due to implementation errors. The resulting constrains is $x \in \{'c'\}$. One can vary that threshold to achieve different results. For a discussion of the impact, the factor has on the constraints, see Chapter 7. For lower/upper bound constraints, the outliers are removed using the technique described for timing constraints.

With this step, we conclude the construction of the high-level model, which describes sequencing, timing, and data rules. The automatic inference of constraints only serves to support the system engineer in constructing the high-level model. The system engineer must review the constructed constraints and modify or remove them if necessary. She may also add additional constraints that have not been inferred. The automatically inferred high-level model may support the system engineer in that task since it provides insight into the system semantics.

5.5 Summary

In this chapter we have introduced the high-level model, which describes the intended architecture using a set of sequencing, timing, and data rules. We introduced the sequence diagram notation used to describe the high-level model and precisely specified the types of rules that we will support in the context of this re-

search. Since it is often difficult for system engineers to identify a concise model of the intended architecture, we developed an approach for supporting the process of constructing the high-level model. In that process, an initial model is inferred using a set of machine learning algorithms. That model contains information that supports the user in modifying the rules to ultimately derive a high-level model that accurately reflects the intended architecture. While we have described the individual steps of constructing the high-level model in detail in this chapter, discusses how these steps are applied in practice.

Chapter 6

Compliance Checking

In the previous chapters, we have described how execution traces are extracted from the system implementation. Furthermore, we have elaborated on a method for eliciting a high-level model in a semi-automated fashion. In this chapter, we will discuss how the execution traces are compared to the high-level model and reflexion models are computed. The compliance checking procedure consists of three steps. In the first step, the interactions of an execution trace are mapped to the interaction in the high-level model. In the second step, the execution trace is checked for violations of sequencing, timing, and data rules that are specified in the high-level model. Finally, in the third step, a reflexion models is computed that graphically illustrates the detected violations. The following describes each of these three steps in detail.

6.1 Mapping

The high-level model describes a set of interactions that occur between a set of components. Rules are specified on these interactions describing the valid system semantics. In order to be able to check the conformance of an execution trace to these rules, it must be determined what interaction in the high-level model (*high-level interaction*) corresponds to what interaction in the execution trace (*concrete*

interaction). In other words, concrete interactions must be mapped to high-level interactions. In Chapter 5, we have described how a high-level model is constructed from the set of execution traces. In the inferred model, all concrete interactions are already mapped to high-level interactions. If the model that is initially inferred remains unchanged, the mapping already exists. However, several situations may occur in which executions traces must be re-mapped to the high-level model. For instance, the high-level model might be specified a-priori without automated support. Also, the initial model is often modified to remove parts that are due to implementation errors. In that process, the mapping is partially removed. The following describes how the mapping between execution traces and high-level model is established.

Such a mapping is also established between the source model and the high-level model in the structural reflexion model approach. While in our approach the mapping is established via interactions, in the structural reflexion model approach source code units are mapped to components in the high-level model. Based on that mapping, inconsistencies between source code dependencies and high-level model dependencies can be identified. The mapping in the structural reflexion model approach is a manual activity. The system engineer identifies sets of source code units that are mapped to each component in the high-level model.

In the behavioral reflexion model approach, a manual mapping is not practical. Each execution trace that is extracted from the implementation often consists of a large number of interactions. Manually assigning interactions in the execution trace to the interactions in the high-level model is, thus, time consuming. Furthermore,

not only a single but often a large number of execution traces is extracted from the system implementation. Each of these traces must be individually mapped to the high-level model, adding an additional burden for the system engineer. In this chapter, we are describing a method for automatically mapping execution traces to a high-level model. We will first describe the basic mapping rules and then introduce an approach from the domain of process validation that we have adapted to automate the mapping process.

6.1.1 Background

In the mapping procedure, each execution trace is individually mapped to the high-level model. The basic mapping principles are illustrated in Figure 6.1. An example execution trace is shown on the left and a high-level model is shown on the right. Only the sequences of interactions are illustrated; information about timing and data are omitted. The first interaction of the execution trace is labeled a. The high-level model also has an interaction with label a in the beginning of the sequence. Thus, the first interaction of the execution trace and the high-level model can be mapped to each other as indicated by the gray shading. The example also illustrates that a high-level interaction that appears in a loop may have more than one concrete interaction mapped to it (e.g., interaction b). A high-level interaction that appears in an alternative construct does not have a concrete interaction mapped to it for every execution trace (e.g. interaction d). Also, a high-level model may specify that an interaction of a certain type may appear more than once (e.g. interaction a).

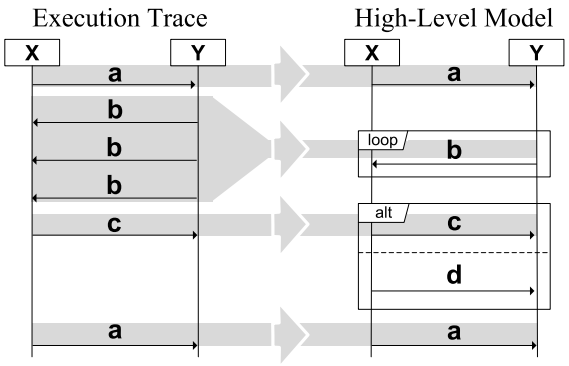


Figure 6.1: Mapping of execution trace to high-level model.

What the example illustrates informally can be expressed in a few mapping rules. The basis of the mapping is the interaction label. Therefore, a concrete interaction is always mapped to a high-level interaction with the same label. A high-level interaction may only have a single concrete interaction mapped to it unless it appears in a loop. In that case, multiple concrete interactions may be mapped to the high-level interactions (as many as there are loop iterations). A concrete interaction may only be mapped to a single high-level interaction.

A simple mapping by interaction label is not possible as the high-level model may specify more than one high-level interaction with the same label. For instance, interaction a in Figure 6.1 appears in the first and the last position of the high-level model. The context in which the interaction appears is taken into account as an additional mapping criterion. The context refers to where in the sequence the interaction appears. We can now refine the mapping rules by stating that a concrete interaction may only be mapped to a high-level interaction with the same label, which appears in the same context.

A possible mapping strategy could be to "execute" the high-level sequence

diagram model with the observed execution trace similar to executing a Finite State Machine (FSM). The concrete interactions are consumed while traversing through the sequence diagram and the mapping is established. Such a mapping is, however, only possible if the observed execution trace is "accepted" by the sequence diagram. That is, it is applicable only to sequences of concrete interactions that adhere to the sequencing rules described in the high-level model. Such an assumption is, of course, not desirable as the purpose of the behavioral reflexion model approach is to identify interactions in execution traces that violate the ordering specified high-level model.

6.1.2 Approximate Matching

In order to solve the mapping issue, we turn to the area of bio-sequence analysis in which approximate matching methods are employed to identify DNA strands that are similar to each other [29]. DNA strands are represented as sequences of symbols (i.e., strings). The traditional approximate matching algorithm tries to find the best alignment between two strings such that the cost of the alignment is minimal. The cost is generally computed via a cost function. A simple cost function is the edit-distance that simply counts the number of edits that are necessary to align the sequences. More elaborate cost functions can assign specific costs for certain matching operations. An example for the approximate matching of two strings is illustrated in Figure 6.2.

Symbols in s_0 that match symbols in s_1 are aligned with each other. If sym-

s_0	q	a	c	_	d	b	d
s_1	q	a	w	x	_	b	_

Figure 6.2: Example for approximate matching of two strings.

bols cannot be aligned, the algorithm uses one of three operations: inserting a gap (indicated by "_") in s_0, inserting a gap in s_1, or mismatching the symbols. Assuming that matching two symbols has a cost of 0, inserting a gap has a cost of 1, and mismatching symbols has a cost of 2, the total cost for the example alignment amounts to 5.

The basic approximate matching algorithm is not sufficient for aligning an execution trace with a high-level model as the former only aligns two sequences of symbols. While a trace can be represented as a sequence of symbols, the high-level model specifies more complex sequencing rules.

Several approaches have been proposed for aligning event sequences to more complex models. Meyers et al. [47] proposes an algorithm for aligning a sequence of events with a regular expression. The regular expression is first converted into a FSM, based on which an *alignment model* is constructed. The alignment model extends the FSM to allow for insertions and deletions. The event sequence is then parsed with the alignment model and the optimal alignment is determined. Another method is presented by Cook et al. [21], in which an execution event stream is aligned with a FSM model. The matching algorithm is based on a best-first search through an alignment tree that is constructed by matching up events in the execution stream to transitions in the FSM. In order to optimize the performance of the algorithm, different pruning methods are illustrated, which can be controlled

by several parameters. This makes it possible to adjust the method to different applications.

The algorithm by Meyers et al. [47] operates in $O(MN)$ time for simple operations, where M is the length of the input string and N the length of the regular expression from which the FSM is produced. However, the runtime of the algorithm increases significantly when aligning multi-symbol gaps [21]. In our research, we have encountered many of such situations. Often times, execution traces are captured that do not show the beginning or the end of a scenario. This is not due to software bugs but to problems in capturing the execution traces. When this occurs, the algorithm must identify a sometimes large sequence of missing interactions as such. Meyers' algorithm has shown to perform poorly in these situations. To solve this problem, we adapted the approach by Cook et al. [21] to match an execution trace to a high-level model. Even with multi-symbol gaps, the algorithms performs in $O(MN)$ time in practice. The runtime increases, however, if the execution trace deviates largely from the FSM. An additional advantage of Cooks approach is that it allows for optimizing the performance using different pruning parameters, which has shown to be useful when applying the behavioral reflexion model approach to different types of systems.

To adapt the approach by Cook et al. for mapping in our approach, a high-level model given in sequence diagram notation must first be converted into a FSM. The algorithm by Cook et al. is then applied to the FSM model. In the following sections we provide an overview of the approximate matching approach by Cook et al. and highlight details that are specific to the application in our mapping procedure. We

then describe how the high-level model is converted into a FSM representation.

6.1.3 Cook's Approach

The approach by Cook et al. aims to identify flaws in the conformance of an *execution event stream* to a *process model*. The process model describes the valid or expected sequence of events, while the execution event stream describes an observed event sequence. The process model is given as a deterministic FSM, where transitions are labeled with events. The goal of the approach is to identify where and how the execution event stream deviates from the process model. Two types of deviations or violations of the process model can be identified: *insertions* and *deletions*. An insertion is an event that was not described by the process model but occurred in the execution event stream. Conversely, a deletion is an event that was expected according to the process model but did not occur in the execution event stream. The algorithm aligns the execution event stream with the process model in a way so as to minimize the number of insertions and deletions. In other words, it aims to make the cost of the alignment as small as possible.

The optimal (or lowest-cost) alignment is determined by constructing an alignment tree that describes the possible alignments of the execution event stream with the process model. Each node in the alignment tree denotes a matching step, i.e., an insertion, deletion, or match. A node in the alignment tree specifies the current state in the process model, the current index in the execution event stream, and the last matching operation. An example for an alignment tree that is constructed for

aligning the event sequence “a b a” with the process model in Figure 6.3 is shown in Figure 6.4. The root node represents the beginning of the alignment when no event has been matched with the process model.

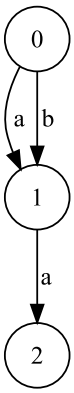


Figure 6.3: Example process model.

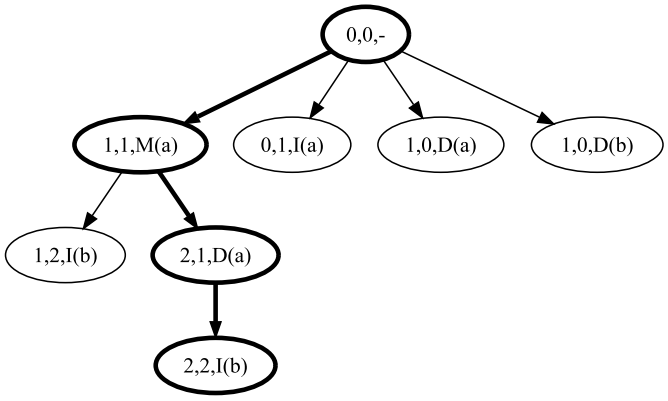


Figure 6.4: Example alignment tree.

After the root node has been created, new nodes are added by evaluating the nodes in the alignment tree in a specific order. When evaluating an alignment node, the algorithm identifies alternatives for aligning the next event in the execution event stream with the transitions emanating from the current process state. For each alternative, a new node in the alignment tree is generated. For instance, the children of the root node in Figure 6.4 describe the different alternatives for aligning the first event in the execution event stream a with the two transitions a and b emanating from the current state 0.

The order in which nodes are evaluated is determined using a best-first search. Each time a new alignment node is generated that describes an alignment alternative, it is not only added to the alignment tree but also to a priority queue. In that

queue, the nodes are ordered by their cost. After one node has been processed, the next node that is evaluated is the one with the lowest cost in the priority queue. Cook et al. describe several ways of computing the cost. For our purposes, we define the cost of a node in the alignment tree to be the sum of all insertions and deletions in the alignment described by the path that leads up to that node. For instance, the alignment that leads to node "*2,2,M(a)*" has a total cost of 1 as it includes only one misalignment (insertion) and two nodes at which events were matched.

The algorithm terminates when all events in the execution event stream have been aligned. An example for an optimal alignment that was discovered for the example is indicated by bold nodes and transitions in the alignment tree shown in Figure 6.4. The algorithm determined a lowest-cost alignment in which the event b is inserted at state 1 and all other events are matched.

Constructing the alignment tree might be resource consuming especially if the execution event stream deviates greatly from the process model. In order to reduce the computational effort of finding the optimal solution, nodes that seem unpromising are removed from the alignment tree. Cook et al. suggest two pruning methods: *cost pruning* and *position pruning*. In cost pruning, alignment nodes are removed whose cost is higher than some threshold relative to the cost of the currently processed node. Position pruning is the removal of nodes that are a certain number of symbols behind the current node in the execution event stream. The threshold parameters can be adjusted to reduce the runtime of the algorithm. However, removing too many nodes may cause the algorithm to find a less than lowest-cost solution. The goal is to find a sufficient trade-off between reduction of runtime and

the likelihood that a lowest-cost path will be detected. With these optimizations, the alignment algorithm has a running time of *O(NM)* in practice, where N is the number of transitions in the high-level model and M is the number of interactions in the execution trace. We have studied the performance of the algorithm and describe the results in Chapter 7.

While the goal of the algorithm presented by Cook et al. is to identify events in the execution event stream that deviate from the process model, we are less concerned with the deviations than with the correct mapping of concrete interactions to high-level interactions. After the alignment node has been discovered that describes an optimal alignment, the path to that node is traversed and the respective interactions are mapped to each other. A concrete interaction that has been identified to be an insertion cannot be mapped to the high-level as no high-level interaction exists that represents it. Likewise, a high-level interaction might not have a concrete interaction mapped to it if a deletion occurred at the respective point in the execution trace.

6.1.4 Converting Sequence Diagrams to Finite State Machines

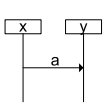
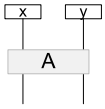
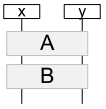
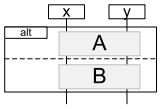
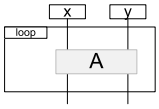
The high-level model may either be specified a-priori or may be automatically inferred (see Chapter 5). If the model is inferred, a FSM representation of the high-level model already exists and no conversion is necessary. However, the system engineer may also give the high-level model in sequence diagram notation. In order to apply Cooks approach to such a high-level model, it must be converted into a

FSM.

The following describes how a sequence diagram is converted into a FSM. We have discussed the reverse conversion from FSM to sequence diagram in Section ref-section:fsmtoseq. This conversion follows the same basic steps in reverse order. The sequence diagram is first converted into a regular expression, which is subsequently transformed into a FSM.

Similar to a regular expression, a sequence diagram explicitly encodes repetitions and branching behaviors. Every sequencing construct in the sequence diagram can be directly translated into a semantically equivalent regular expression term as shown in Table 6.1. A single interaction is converted into a single symbol in the regular expression. The sequencing operations can be applied to any sequence of interaction (indicated by A and B). A set of interactions A that is followed by a set of interactions B is written as AB in the regular expression. A set of interactions A that are an alternative to another set of interactions B is written as $A+B$. Finally, a set of interactions A that may be repeated 0, 1, or more times is written as A^*. The information about the sending and the receiving components cannot be expressed in the regular expression and is, thus, lost in the conversion.

Table 6.1: Conversion of sequence diagram constructs to regular expression elements.

Sequence Diagram					
Regular Expression	a	A	AB	$(A)+(B)$	A^*

In order to derive a regular expression from a sequence diagram, the sequence

diagram is parsed and every element of the diagram is converted into a regular expression construct. An example for a high-level model in sequence diagram representation is shown in Figure 6.5. The regular expression that is derived from that model is $(a + b)a$.

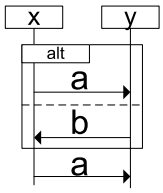


Figure 6.5: Example high-level model.

After a regular expression has been derived from the sequence diagram, it is translated in a deterministic FSM using the method based on the derivatives or regular expression presented by Brzozowski [14]. The final FSM of the example model shown in Figure 6.5 was illustrated in the alignment example (see Figure 6.3).

6.2 Compliance Checking

After an execution trace has been mapped to the high-level model it is compared against rules of the high-level model and violations are recorded. These violations will later be used to construct a graphical model of the behavioral violation: the reflexion model. In this section, we describe the compliance checking process in detail.

An execution trace describes a sequence of interactions that have been observed during system execution. An interaction denotes the exchange of a single message between two components. The high-level model describes the allowed se-

quence of interactions using a set of sequencing operators. Furthermore, it specifies rules on the timing of events and the data carried by parameters via constraints. In the first step of the compliance checking process, the interactions in the execution trace (concrete interactions) are checked for consistency with the sequencing rules described in the high-level model. Subsequently, the timestamps observed for the send- and receive-events of interactions are checked for compliance with timing constraints. Simultaneously, parameter values are checked for violations of data constraints. In the following sections, we describe first how execution traces are checked for compliance with sequencing rules and then for compliance with timing and data constraints.

6.2.1 Checking for Compliance with Sequencing Rules

The high-level model describes rules on the ordering of interactions via constructs specified in the sequence diagram notation, which is defined as part of the Unified Modeling Language (UML) [10]. For our research, we consider a limited set of three sequencing constructs: concatenation, loops, and branches. For details about the sequencing constructs and the notation used to specify the high-level model see Chapter 5.

Sequencing rules are typically described as part of protocol specifications in distributed systems. In order for two or more component to be able to interact reliably and efficiently, they must correctly implement the protocol and adhere to its sequencing rules. However, implementation errors are inevitably introduced during

the development process, which may cause a component to deviate from the sequencing rules and to exhibit invalid behaviors. Rather than causing a component to exhibit an entirely unexpected interaction sequence, implementation errors often trigger only slight deviation from the valid system semantics. More specifically, they cause a component to exhibit *insertions* and *deletions*. Insertions are unexpected interactions that occur in the execution trace but are not described as part of the high-level model. Conversely, deletions are interactions that were expected according to the high-level model but do not occur in the execution trace. The goal of checking the compliance of an execution trace to the sequencing rules described in the high-level model is to identify such insertions and deletions.

Previously, we have discussed the approximate matching approach by Cook et al. [21], which we have adopted for the mapping of execution traces to the high-level model. The algorithm creates a mapping that minimizes the total number of deviations. In the mapping procedure, we only focused on the alignment of interactions but disregarded the information about detected violations. In order to check the compliance of an execution trace to the sequencing rules, we extend the mapping process described previously slightly. Instead of disregarding insertions and deletions, the violations are collected to be used in a later step of the behavioral reflexion model approach when constructing the reflexion model (for details see Section 6.3). In other words, the mapping and sequencing checking are conducted in the same step.

6.2.2 Checking for Compliance with Timing and Data Rules

In addition to sequencing rules, the high-level model specifies rules on the timing of events and the parameters carried by messages. These rules are described via constraints, which are Boolean expressions that evaluate to true if they are satisfied and to false if they are violated. If the variable values cause the expression to evaluate to false, a violation of the constraint is encountered. Conversely, if the variable values evaluate the expression to true, the values satisfy the constraint.

In order to check the compliance of an execution trace with a high-level model, the timestamps of events must be checked for compliance with timing constraints. Likewise, the values of parameters must be checked for compliance with data constraints. Such a checking is also conducted in conventional testing or runtime verification procedures. In such procedures, the system is executed and the values of variables are monitored. If the values cause a constraint to evaluate to false, an error is reported. Rosenblum [63] describes an approach for inserting assertions into a program, which are checked during runtime. Assertions are similar to constraints specified in the high-level model as they specify Boolean expressions. The developer specifies assertions at specific points in the program's source code. When the program is executed, the assertions are checked for violations. In case a violation is detected, the program enters an error handling routine and outputs runtime information to support debugging.

The compliance checking of captured execution traces is similar to the testing or runtime verification of programs. Each trace is parsed and timestamps of events

as well as values of parameters are read. If a constraint refers to the respective event or parameter, it is evaluated with the updated values and a violation is reported if the constraint evaluates to false. Upon encountering a constraint violation, a *violation record* describing the instance of the violation is produced. A violation record contains references to the event or parameter that caused the violation as well as a reference to the violated constraint. After all traces have been checked, the violation records are used to construct a reflexion model. Details about the construction of reflexion models are provided in Section 6.3.

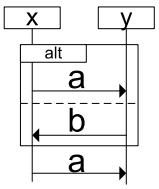


Figure 6.6: Constraint checking example.

An example for the checking of timing and data constraints is illustrated in Figure 6.6. A high-level model specifies three data constraints and one timing constraint. A behavior is observed, whose events have concrete timestamps and their parameters carry concrete values. The execution trace is parsed and the timestamps of events as well as the parameter values are monitored. The message *start* has a parameter *req_mode* with the value "medium". According to one of the data constraints, the parameter *req_mode* may only carry the values "slow" and "fast". Thus, the observed parameter value violated that constraint. A violation record is produced indicated by the gray box, which captures information about the instance of the violation. Similarly, violation records are produced for every data message as

the values of their *res_mode* parameters are not consistent with the value specified by *req_mode*. Finally, a timing constraint violation is encountered as the time between the receive-event of *stop* and the send-event of *confirm* is greater than 30ms.

6.3 Constructing Reflexion Models

A key component of the behavioral reflexion model approach is to represent deviations between an implementation and high-level model in a way that allows the system engineer to reason about them. Deviations are not always due to implementation errors but may also stem from incorrect rules in the high-level model. The system engineer must be able to identify the source of the deviation to be able to make the necessary modifications. Furthermore, when an implementation violates the intended architecture, it often causes the system to exhibit an unexpected behavior and, thus, leads to additional deviations from the high-level model. The system engineer must be able to identify the root cause of such erroneous behaviors in order to systematically debug the implementation. In this section, we describe the reflexion model, which illustrates deviations between high-level model and extracted execution traces in a graphical manner. With such a graphical presentation of deviations, the system engineer can derive an overview of the discrepancies, reason about them, and adjust the high-level model and the implementation, respectively.

In the structural reflexion model approach [45], the high-level model describes the components and dependencies of a system. A model of the source code structure, the *source model*, is extracted from the system implementation and compared to the

high-level model. The resulting reflexion model illustrates inconsistencies between high-level dependencies and source dependencies.

An example for a structural reflexion model is shown in Figure 6.7. The boxes represent components and the lines illustrate dependencies. The reflexion model shows three types of dependencies. *Convergences* illustrate an agreement between high-level model and source model, and are drawn with a solid line. *Absences* illustrate dependencies that are present in the high-level model but are missing in the implementation (drawn with a dotted line). *Divergences* are dependencies that are in the implementation but do not exist in the high-level model (drawn with a dashed line).

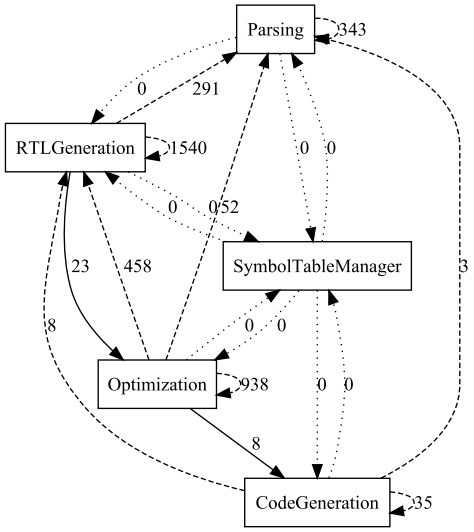


Figure 6.7: Structural reflexion model [45].

In the behavioral reflexion model approach, the reflexion model shows deviations between the extracted execution traces and the high-level model. The high-level model describes rules on the sequencing of interactions, the timing of

events, and on the message parameters. The behavioral reflexion model illustrates violations of these rules in a graphical manner.

A challenge in constructing a behavioral reflexion model is the potentially large number of execution traces and the large size of individual traces that are captured from the implementation. A reflexion model must be sufficiently compact to allow the system engineer to quickly identify violations and reason about them on a high level. At the same time, a reflexion model must provide enough details to allow for investigating concrete problems. We address this issue by computing two types of reflexion models, which differ in the amount of information they depict. The *behavior reflexion model* shows a single execution trace and how it deviates from the high-level model. This allows for a detailed analysis of the deviations. However, it is cumbersome to analyze a large number of traces. Thus, the *high-level reflexion model* illustrates a high-level model in which the deviations of all traces are highlighted. Finally, we offer an alternative representation of the high-level reflexion model in Finite State Machine (FSM) notation: the *FSM reflexion model*. That representation has shown to be useful especially when analyzing more complex high-level models. The following describes the three types of reflexion models in detail.

6.3.1 High-Level Reflexion Model

The goal of the high-level reflexion model is to provide the system engineer with an abstract view of all detected violations. The high-level reflexion model is

constructed by updating the high-level model with the detected deviations. If no violations were detected, the high-level reflexion model is equivalent to the high-level model.

First, sequencing violations are highlighted. Interactions in the high-level model that did not occur in one or more execution traces, i.e., deletions, are simply marked as such. That is, the respective interaction arrow is drawn with a dotted line and annotated with a cross symbol. Since more than one execution trace might omit the interaction, the number of deletions is indicated in parentheses. An insertion represents an interaction in an execution trace that is not specified in the high-level model. In order to illustrate an insertion, an arrow is inserted immediately after an interaction in the high-level model after which the insertion occurred in the trace. For instance, see the example shown in Figure 6.8. In one of the execution traces, an insertion "reset" occurs after interaction "data". Hence, an arrow is inserted after interaction "data" in the high-level model. The multiple labels of the arrow indicate that more than one insertions occurring at the same position in the high-level model were detected. All interactions that occur at the same position in the high-level model are aggregated and the arrow in the high-level model is annotated with the label of all inserted interactions.

In addition to sequencing violations, the high-level model is updated with timing and data constraint violations. To illustrate timing constraint violations, the violated constraint is annotated with a clock symbol. Conversely, data constraint violations are indicated by annotating the violated constraint with a lightning symbol. Since a constraint may be violated multiple times, a constraint is also annotated

with the number of violations in parentheses.

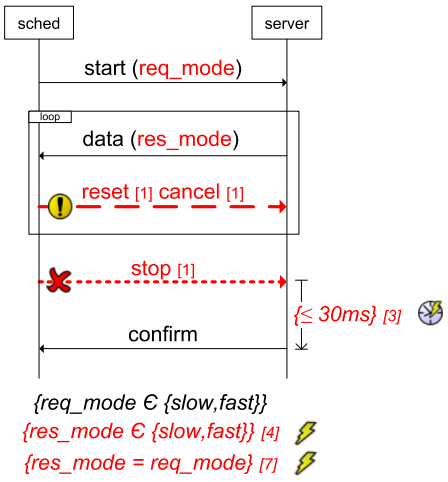


Figure 6.8: High-level reflexion model.

The high-level reflexion model provides an abstract view of sequencing, timing, and data violations. It allows for deriving a sense of the overall number and types of violations. Especially with a large number of execution traces, such an overview is important. However, by aggregating violations, many details are lost. For instance, one cannot identify where exactly in the interaction sequence the insertion "reset" occurred. It could be either in between "data" interactions or between interactions "data" and "stop". To further investigate violations and resolve them, such details are needed. To support this level of analysis, we construct a *behavioral reflexion model*.

6.3.2 Behavior Reflexion Model

A behavior reflexion model shows a single execution trace (describing a single behavior) and highlights its violations of the rules specified in the high-level model.

The execution trace is shown in sequence diagram notation (see Figure 6.9). Components are illustrated by labeled lifelines and interactions are represented by labeled arrows. The arrow label indicates the type of the message that is exchanged. Both ends are annotated with timestamps of the send- and receive-event respectively. The sequence diagram showing the execution trace is then updated with sequence, timing, and data violations. First, insertions and deletions are marked in the interaction sequence. Since deletions are interactions that did not occur in the behavior, they must be inserted into the interaction sequence. The arrow of a deletion is drawn with a dotted line and annotated with a cross symbol. An interaction that has been identified as an insertion is simply marked as such. That is, it is drawn with a dashed line and annotated with an exclamation mark symbol. Furthermore, timing and data constraint violation are shown in the diagram. An interaction is annotated with all constraints that it violated. A violated data constraint is annotated with a lightning symbol and a violated timing constraint is annotated with a clock symbol. An example for a behavior reflexion model is shown in Figure 6.9.

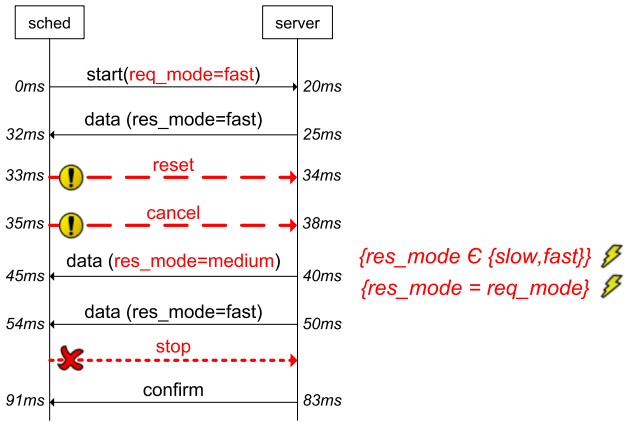


Figure 6.9: Behavior reflexion model.

While the behavior reflexion model shows only a single execution trace, it provides more details about the violations detected for that trace. More specifically, it illustrates the exact position of insertions and identifies the concrete interactions that caused constraint violations. In our research, we have found that the high-level and behavior reflexion models can be used at different stages in the analysis process. Initially, the system engineer often seeks a general idea of what types of rules are violated and how many violations have been detected. The high-level reflexion model, thus, is well suited for an initial analysis. Using the high-level as starting point of debugging activities, the system engineer then analyzes the reflexion models of individual execution traces to locate the source of the problem and resolve it. Chapter 7 provides a detailed discussion on the use of reflexion model for verification and maintenance tasks.

6.3.3 FSM Reflexion Model

In Chapter 5, we have described the use of a Finite State Machine (FSM) model for reconstructing a high-level model that describes the intended architecture. While all supported sequencing rules can be expressed with both a sequence diagram and a FSM model, the notations differ in the way they illustrate these rules. A sequence diagram is useful to make sequencing constructs, such as loops and alternatives explicit. However, in our studies of various systems, we have identified that sequence diagrams are limited in illustrating complex sequencing rules. The FSM represents a suitable alternative for analyzing complex sequencing rules

and illustrating deviations from complex models. We, thus, also construct a FSM reflexion model.

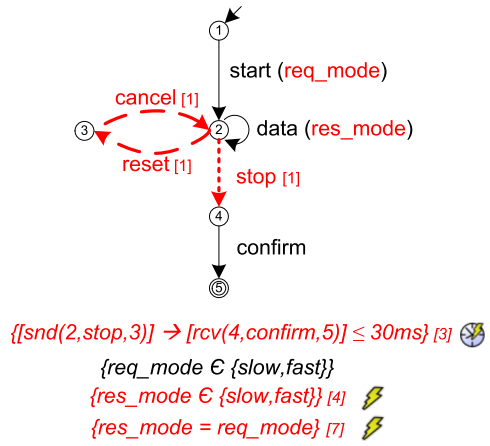


Figure 6.10: FSM reflexion model.

Similar to the high-level reflexion model, the FSM reflexion model illustrates the violations of all execution traces in a single model. More specifically, it describes the sequence, timing, data rules, and illustrates violations thereof. The major difference between the high-level reflexion model and the FSM reflexion model is the way in which sequencing violations, i.e., insertions and deletions are illustrated. Deletions are highlighted by drawing the respective transition with a dotted line. Since insertions are absent from the high-level model they must be added to the FSM. Insertions are illustrated by inserting transitions at the state at which the violation occurred. For instance, at *State 2* after the "start" interaction in Figure 6.10, one of the execution traces has an unexpected "reset" and then "cancel" interaction. Insertion transitions are added starting and ending at *State 2*. If in an execution more than one interaction is inserted at a point in the high-level model, multiple transitions are added, originating and ending at the state at which the insertion oc-

curred. States must be added in order to connect the transitions. More specifically one state less than the number of insertions is added. The resulting FSM accepts (at least) the same language as the high-level model. In other words, it accepts all valid executions. In addition, it accepts all invalid interaction sequences.

The sequence diagram provides a convenient notation for specifying timing constraints by annotating the respective events. However, in an FSM, events cannot be annotated in the same manner. Thus, timing constraints are, similar to data constraints, added as independent elements. The event annotations are replaced by event identifiers. An event is specified by the transition and the event type. For instance, the event "snd(2,stop,3)" refers to the send-event of the interaction represented by the transition from *State 2* to *State 3*, which is labeled "stop". Finally, data constraints in a FSM reflexion model are illustrated in the same way as data constraints in the high-level reflexion model by annotating them with a lightning symbol and the number of violations. A parameter in a data constraint refers to all parameters with that name in the FSM reflexion model.

6.4 Summary

In this chapter we have described the checking of execution traces for violations of the high-level model and how these violations are presented in a reflexion model. The compliance checking process consists of three steps, similar to the structural reflexion model approach by Murphy et al. [45].

First, we discussed the issue of mapping execution traces that have been ex-

tracted from a system implementation to a high-level model that describes the intended architecture. Due to the large amount of data that is collected from the system implementation, only an automated mapping approach is feasible in practice. We have described a method for automating the mapping based on an approximate matching algorithm that was presented by Cook et al. [21]. We adapted the approach and introduced an additional step in which the sequence diagram representation of a high-level model is converted into a FSM representation. The result is an automated mapping procedure in which execution traces are mapped to a high-level model despite sequencing errors. The mapping is conducted in a way so as to minimize the number of sequencing errors.

Furthermore, we have described how execution traces are checked for the compliance with sequencing, timing, and data rules that are specified in the high-level model. Violations of sequencing rules have already been identified during the mapping of the execution trace to the high-level model. Only a small extension of the mapping algorithm is necessary to record the detected insertions and deletions. Checking the compliance of timing and data constraints is similar to the testing or runtime-verification of programs. The interactions of a trace are parsed and the timestamps of events as well as the parameter values are monitored. If a value causes a constraint to evaluate to false, a violation is recorded. The result of the compliance checking process is a set of sequencing, timing, and data violations.

Finally, we have explained how the detected violations are presented in reflexion models. More specifically, we have presented three different types of reflexion models that help to illustrate the violation on both an abstract and detailed level.

Chapter 7

Evaluation

In the previous chapters, we have presented the behavioral reflexion model approach in detail. We first elaborated on a method for extracting behavioral information from a system implementation. Subsequently, we described a semi-automated approach for constructing a high-level model. Finally, we illustrated how reflexion models are computed that graphically illustrate deviations between execution traces and high-level model. In this chapter, we describe the application of our approach and evaluate its capabilities for identifying inconsistencies between architecture representations. As the results are difficult to quantify, the evaluation is qualitative: we discuss two case studies, in which we applied our approach to two space mission systems.

7.1 Research Questions

The evaluation of our approach consists of several parts. The goal of the behavioral reflexion model approach is to support the system engineer in identifying and resolving discrepancies among architecture representations. More specifically, we aim to identify discrepancies between documented and intended architecture. Furthermore, we aim to detect inconsistencies between implementation and intended architecture. We can formulate these objectives in the following two research ques-

tions:

1. Can our approach be used to align the architecture documentation with the intended architecture?

2. Can our approach be used to align the implementation with the intended architecture?

The purpose of identifying inconsistencies among architecture representations is to support maintenance activities by producing accurate architecture design documentation that facilitates understanding. Furthermore, by aligning the implemented architecture with the intended architecture, flaws in the reliability of system that are caused by unintended behaviors can be resolved. Based on these goals, we can formulate two additional research questions:

1. Do the results of our approach facilitate maintenance?

2. Does the application of our approach lead to increased system reliability?

Finally, we aim to answer questions regarding the application and performance of the behavioral reflexion model approach. In Chapter 5, we have discussed several machine-learning algorithms that are used to infer a high-level model from execution traces. One parameter that was used to adjust the sequencing model was the transition support. So far, we have only discussed how the model is computed. In this chapter, we describe what values produce satisfactory results in practice. Furthermore, we will provide an evaluation of the performance of our approach. When constructing the high-level model, several machine-learning approaches are applied

to infer information from execution traces. These traces may potentially be large and many of these traces might be collected. We, thus, discuss the scalability of different parts of our approach. We formulate these objectives in the two last research questions:

1. What parameter values yield satisfactory results in producing the high-level model?

2. Is our approach computationally tractable?

7.2 Subject System

We have chosen two case studies that differ in the types of problems that need to be addressed. Both case studies describe experiences in applying the behavioral reflexion approach to a space mission system developed and operated by Johns Hopkins University's Applied Physics Laboratory (JHU/APL) under contract of the National Aeronautics and Space Administration (NASA). The system was deployed in 2004 as part of the Mercury Surface, Space Environment, Geochemistry, and Ranging (MESSENGER) mission [38]. In that mission, a spacecraft in orbit captures telemetry data of the planet Mercury for the analysis by scientist on earth. Figure 7.1 shows a simplified system setup that includes only the components that we will discuss in the case studies below.

The spacecraft component is controlled by the Mission Operation Center (MOC) on earth. In orbit, the spacecraft captures telemetry data, such as images. Every time the spacecraft is within reach of the MOC, it establishes a connection

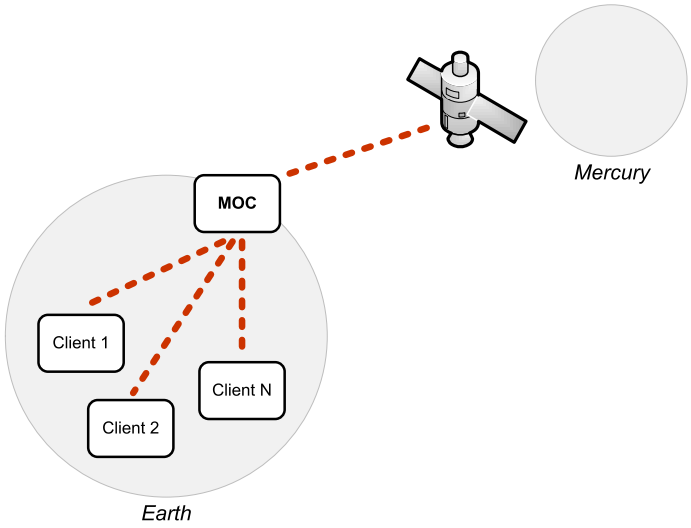


Figure 7.1: Setup of system used in case studies.

and downloads files containing the telemetry data. Upon reception, the MOC stores all data. Furthermore, the MOC serves as an interface for client components to access the data. The client systems, which are developed and operated by external organizations, are used by scientists to query and analyze the telemetry data. The client system formulates a query, requesting specific data from the MOC. The MOC then accesses its storage and returns the desired data to the client.

The process of downloading telemetry files from the spacecraft to the MOC and the process of serving that data to client systems are independent. In this chapter, we describe a case study for both processes in which the behavioral reflexion approach is employed for solving two different types of problems.

In the first study, we describe the reverse engineering of an accurate model of the intended architecture for a spacecraft/MOC communication. The task at hand was to identify the characteristics of a system in order to be able to compare the

behavior of a new system against it. Ambiguous documentation made it difficult to identify the behavior of a particular configuration. At the same time, the new system should seamlessly replace the existing system. We discuss the reverse engineering of an accurate model from an existing implementation.

The second case study describes the ground communication part of a space mission system. A MOC server communicates with external clients that seek to access data that was captured in space. During the operation of these components, problems have been reported with the reliability and availability of the MOC. The task of the MOC system engineers was to identify the interaction errors that led to these problems and locate their source.

7.3 Experimental Setup

We have implemented a series of tools to apply the behavioral reflexion model approach in the case studies. The tool chain is illustrated in Figure 7.2. White rectangles illustrate steps in the tool-chain. Gray shaded boxes illustrate individual tools. In order to extract execution traces, we go through two or three steps. First, the network traffic is captured using a network sniffer. In our work, the system engineers have used several different network sniffers depending on their availability and the engineer's preferences: UNIX Snoop, Wireshark [52], and bro [55]. Each of these tools connects to the network interface of a system and records all the communication traffic that passes through it. The information is then saved in the space-efficient *pcap* format. We then apply a tool that we have developed to

extract application-level information from the network traffic. The tool uses the NetPDL library [62] to dissect network packets and convert them into application-level messages (see details in Chapter 4). The application-level execution traces are saved in Comma Separated Value (CSV) files. In some cases, the traces must be further processed to ensure that each execution is stored in an individual trace file. We describe that manual processing in more detail for each case study.

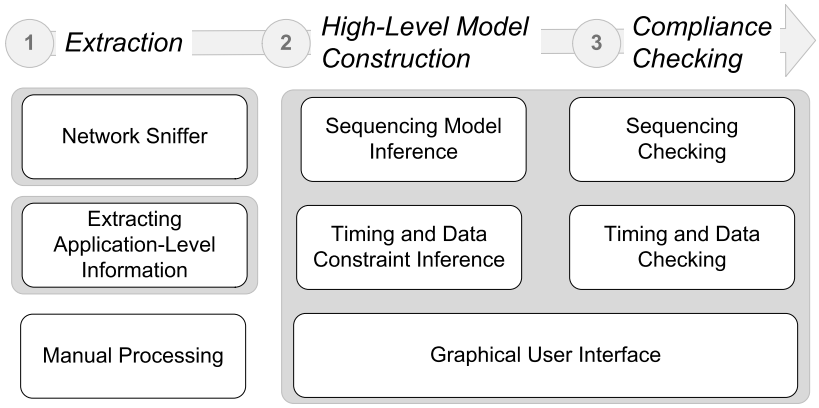


Figure 7.2: Tool chain implementing the behavioral reflexion model approach.

All remaining steps of the behavioral reflexion model approach are implemented in a third tool. Besides implementing the steps of the reflexion model approach, the tool provides a graphical user interface that displays the reflexion models and allows the system engineer to modify the high-level model. After importing the captured execution traces, the system engineer typically constructs a high-level model. In one component of the tool, we implemented the machine learning algorithms discussed in Chapter 5 for inferring an initial sequencing model as well as the set of default constraints. These component also computes transition support and constraint probabilities, respectively. The resulting high-level model

is then illustrated graphically in the tool's user interface. The system engineer can modify the high-level model to match the intended architecture. Once a satisfactory high-level model has been constructed, the execution traces can be checked against it. We implemented a tool component for identifying sequencing violations, which also constructs the reflexion model. Likewise, we implemented a component for checking the execution traces for violations of timing and data constraints.

Figure 7.1 only illustrates the tool components that we have developed for applying the behavioral reflexion model approach. In order to be able to apply an approach, a process must be specified of how these components used. We provide a discussion of that process in Section 7.4.3 and Section 7.5.4, respectively.

7.4 Case Study I: Software Maintenance

7.4.1 Task

In the MESSENGER mission, the spacecraft downloads all telemetry data to the MOC on earth. To conduct the file transfer, the MOC and the spacecraft use a reusable communication component. That communication component was developed at JHU/APL based on the CFDP protocol [26] specified by the Consultative Committee for Space Data Systems (CCSDS). After the component had been in operation for several years, it should be replaced by a new component that implements the same protocol but is smaller and less complex than the legacy component. The task of the MOC system engineers was to replace the legacy component with the new communication component. While the new component had passed testing

procedures, the system engineers wanted to ensure that that it behaves equivalently to the legacy component. One of the main reasons for the need of an additional confirmation was the large number of component configurations. While a system may be free of bugs, it might deviate from the desired behavior if the configuration settings have not been set properly.

The goal of applying the behavioral reflexion model approach was to fully yet abstractly characterize the behavior of the legacy component. More specifically, documentation should be constructed that accurately describes the specification of the system and the particular configuration used.

7.4.2 Application

The application of the behavioral reflexion model approach was conducted in two stages. First, the system engineers collected behavioral information from the system implementation on site. In the second stage, the we processed the traces and constructed the high-level model.

7.4.2.1 Trace Extraction

A simulation environment for the MESSENGER mission exists at JHU/APL that allows system engineers to test systems in an environment that closely resembles real-world conditions. The system engineers executed the system and collected network traffic by monitoring the network interface of the MOC. While no particular coverage technique was employed to ensure that a representative sample of execu-

tion traces is recorded, the system engineers executed the systems under different conditions and collected a large number of execution traces to gain confidence in the sample.

The subsequent steps of constructing a high-level model were conducted off-site by us. After receiving the execution traces, we first extracted application-level traces from the low-level network traces. The message formats, which must be specified before the application-level information can be extracted, was described in the CFDP protocol specification [26]. We translated the information in the CFDP specification document into NetPDL format and extracted the application-level information using the NetPDL library [62]. The result of this conversion step was a single file in Comma Separated Values (CSV) format, which described the inter-component interactions on application level.

In the next step, we further processed the trace file. The single trace file that was produced in the previous step included executions that occurred in parallel. More specifically, multiple files were transferred simultaneously from the spacecraft to the MOC. In our approach, we require each execution to be stored in a separate file. Furthermore, concurrent executions must be split into non-concurrent ones before the machine-learning algorithms can be applied. Splitting up the trace file into files that only describe individual executions was rather straight-forward. In order for the communication system to be able to handle concurrent file transfers, each message that is exchanged carries a *transaction sequence number*, which assigns a unique identifier to each file transfer procedure. We divided the messages in the trace file by that transaction sequence number and saved the resulting sets in

individual files.

7.4.2.2 Constructing a Sequencing Model

After that initial effort, we started constructing a high-level model. We started with constructing a model of the sequencing semantics. Since the high-level model should describe the semantics of the legacy component, we used the execution traces that were extracted from that component to infer an initial high-level model. More specifically, we applied the machine-learning methods discussed in Chapter 5 to distill a FSM model describing the implemented sequencing semantics.

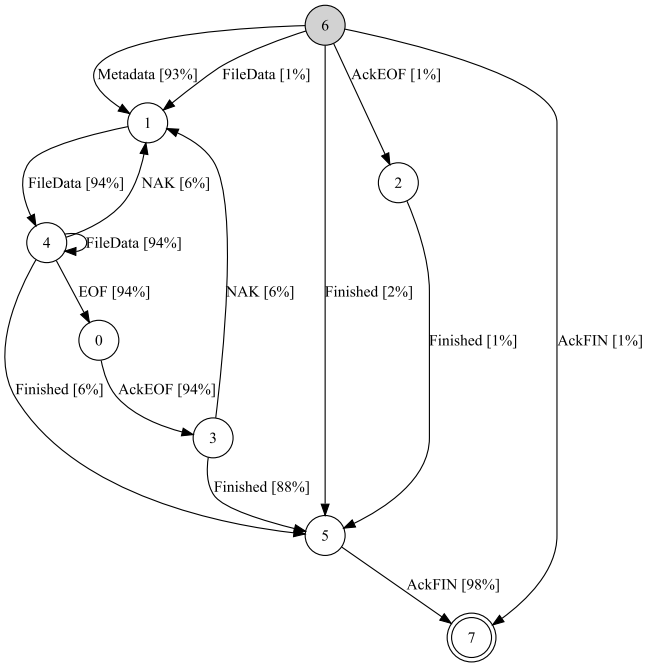


Figure 7.3: Inferred sequencing model of the MOC/spacecraft communication.

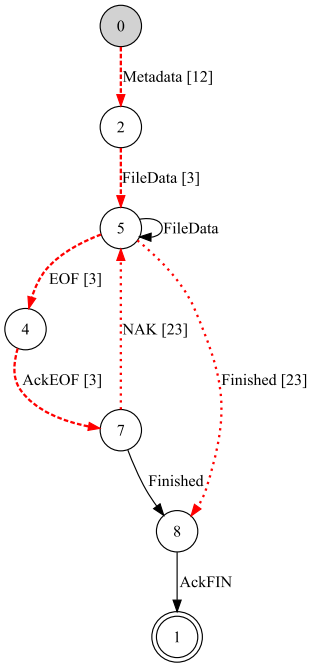


Figure 7.4: Reflexion model of the MOC/spacecraft communication.

Figure 7.3 shows the initial sequencing model that was inferred from the extracted execution traces. The initial sequencing model was rather complex and did not resemble the sequencing rules that we expected to find in the execution traces. The complexity of the model typically stems from behavioral anomalies. That means that many of the transitions are invalid and not part of the intended architecture. In the next step of our analysis, we modified the model by removing invalid edges in order to derive a model that resembles the intended architecture.

In a previous chapter, we have discussed the measure of *support* as a means to identify invalid transitions. The support expresses the fraction of behaviors that execute a given transition. Each transition of the model in Figure 7.3 is annotated with its support. In order to derive a model that resembles the intended architecture, we removed transitions with a low support, or low-probability transitions. More specifically, we removed transitions whose support is below a certain threshold. In this sequencing model, we could clearly identify low-probability and high-probability transitions. Several transitions in the model received only a support of less than 10%, while the support of the rest was 88% or more. We removed all transition with a support less than 10%, resulting in a model, which we believed represented the intended architecture.

After constructing the initial sequencing model, we computed a reflexion model showing how and where the extracted execution traces deviate from the sequencing model. The reflexion model is shown in Figure 7.4. In the reflexion model, four of the transitions are marked as deletions and two insertions are highlighted. We investigated the deletions further by computing reflexion models of individual

executions, in which these violations occurred. Our analysis determined that the deletions are noise in the execution traces. The trace capture was started while the system was operating. As a result, the beginning of some file transfer procedures were not recorded and, thus, marked as deletions in the reflexion model. While the noise disturbed the inferred model significantly (see Figure 7.3), the reflexion model is able to illustrate the desired semantics and represent the noise by highlighting the respective transitions. Since the deletions were no reason for concern, we simply ignored them.

The reflexion model in also shows two insertions labeled "NAK" and "Finished". The behavior reflexion model revealed that these interactions occurred in the same execution trace. The system engineers explained that these interactions are part of an error handling routine that is built into the CFDP protocol. Whenever the MOC detects that part of a file did not arrive, it issues "NAK" messages (Negative Acknowledgement) requesting the spacecraft to retransmit the missing data. We removed these transitions from the initial high-level model due to their low support. However, as shown in this case, a transition can receive a low support even if it is valid since not all paths are executed equally often. Error handling routines are a typical example for valid but infrequently executed paths.

This shows that the support only serves as an indicator as to what transition could potentially be invalid but is not always a reliable classifier. The reflexion model can be used to reason about whether the high-level model is correct, i.e., reflects the intended architecture. Based on the graphical representation of sequencing deviations, the system engineer can identify whether they are due to implementation

errors, issues in the high-level model, or noise in the captured execution traces.

7.4.2.3 Constructing Timing Constraints

After anomalous transitions have been removed, it is less difficult to reason about timing and data constraints as the sequencing model is less complex and reflects the expected system semantics. In the subsequent step, we inferred timing constraints to reason about the timing behavior of the system. The following discusses selected timing constraints.

Since the system engineers were only able to capture the communication on the network interface of the MOC and not at the spacecraft, only partial timing information was available. The inferred timing constraint, thus, described only the timing behavior at the MOC. However, even without all the timing information, it is possible to indirectly reason about the delays in the spacecraft. For instance, consider the constraint

$\{[snd(4, AckEOF, 7)] \rightarrow [rcv(7, Finished, 8)] \leq 300ms\}$.

The message "AckEOF" is emitted by the spacecraft to acknowledge the receipt of the "EOF" message. The spacecraft must then immediately send a "Finished" message. The constraint expresses the time that elapses between receiving the "AckEOF" and the "Finished" message at the MOC. In other words, it illustrates that the delay of the spacecraft when sending these two messages does not exceed 300ms, assuming that both messages travel with an equal speed through the network. Although we must make several assumptions, the partial trace information

allows us to reason about the delay of the spacecraft that is located far from the MOC. That information is useful when implementing the replacement component as it illustrates the delays that can be expected between receiving messages and helps to determine after what delay a connection issue is detected and an error handling routine should intervene.

It is also possible to reason about the network latency despite partial timing information. The constraint

$$\{35s \leq [snd(4, Finished, 7)] \rightarrow [rcv(7, AckFIN, 8)] \leq 36s\}$$

expresses the time between which the MOC emits a "Finished" message and receives a response in form of a "AckFIN" message. The response time varies between 35 and 36 seconds. Assuming that the delay of the spacecraft is similar to the one identified previously (less than 300 ms) and assuming that the upload and download latency is similar, a message travels around 17 seconds on the network. Although such a large latency would be unusual for ground communication, it is expected in this setup as the spacecraft is located at the planet Mercury and messages must travel a large distance between the components. The timing constraint illustrates the latency that must be expected by the new communication component. The design of the new component must take such latencies into account.

Constructing the initial high-level model and its constraints is a fully automated process. Machine-learning algorithms described in Chapter 5 are employed to judge what observed timing values are valid and what values could potentially be due to errors. The machine-learning algorithms are not always successful in separating valid from invalid values and might construct constraints that include

invalid values (false positives) or exclude valid values (false negatives). Our goal was to construct constraints that reflect the intended architecture and avoid false positives and false negatives. In order to check the timing constraints for accuracy, we computed a reflexion model showing timing constraint violations. The high-level reflexion model illustrated that several timing constraints were violated. To identify false negatives we evaluated all concrete timing values that caused these violations. One of the constraints we analyzed was

$$\{[rcv(0, Metadata, 2)] \rightarrow [rcv(2, FileData, 5)] \leq 300ms\}.$$

After computing the reflexion models, we could see what the concrete values were that violate that constraint. A total of 15 violations were identified. 14 of these were caused by a delay of 344ms. Since that delay was only slightly smaller than the previous upper bound of 300 milliseconds, we decided that the constraint was too restrictive and manually adjusted it to include these values. After adjusting the constraint, we computed the reflexion model again. Only one value, a delay of 48 seconds and 17ms, violated the new constraint. That value was significantly larger than all other values, which presumably presented a problem in the operation of the system. A high delay during the data transmission might lead to the loss of data that cannot be downloaded in the remaining time. We took note of this issue for discussion with the MOC system engineers.

7.4.2.4 Constructing Data Constraints

After analyzing the sequencing model and timing constraints, we focused on the parameters of the messages that are exchanged between the MOC and the spacecraft. More specifically, we used the machine-learning algorithms described in Chapter 5 to infer data constraint that provide information about the parameter values. An example for an inferred data constraint is

$\{236306 \leq FileSize \leq 1095706\}$.

The constraint describes the minimum and maximum values that were observed for the message parameter "File Size". The parameter is submitted as part of the "Metadata" message in the beginning of the file transfer procedure. It informs the MOC about the size of the file in bytes that is about to be transmitted. The constraint illustrates that the size of the files that were transmitted during the observed executions was between 200 and 1000 kilo bytes. The files are not transmitted at once but split up into several elements, which are sent in separate packages, or Protocol Data Units (PDUs). Another data constraint that was inferred provided information about the payload of the individual PDUs:

$\{981 \leq PDUDataFieldLength \leq 1031\}$

The constraint illustrates that the payload of a PDU is around 1 kilo byte. That means that the spacecraft would have to transfer around 1000 PDUs to transmit a large file. Similar to timing constraints, we also evaluated the inferred data constraints for false positives and negatives. When evaluating the values for the parameter "PDU Data Field Length" that were classified as violations by the above

constraint, we identified several false negatives. Some of the PDUs transmitted a payload that was less than the lower bound of 981 bytes. We further investigated the values by computing behavior reflexion models details of the behaviors that violated the constraint. The reflexion models revealed that all violations were caused by the last "FileData" message in an execution before ending the transmission. The reason for the low payload was that the last "FileData" message contains not a full payload but only the part of the file that could not be transferred with the previous message. Since these are not violations, we modified the lower bound of the constraint to include these values. After adjusting the constraints, we again computed the reflexion model to check for false negatives and false positives. We repeated that process until we derived constraints that do not misclassify parameter values. Some of the parameter values appeared to be erroneous. These potentially erroneous values could be due to an error in the implementation or inaccuracies in the data collection. We recorded those to discuss them with the system engineers.

7.4.3 Process Discussion

In this case study, we have described the application of the behavioral reflexion model approach to construct a model of the intended architecture that describes the interactions between the spacecraft and the MOC of the MESENGER mission. The goal of constructing a high-level model was to study the characteristics of a legacy component a replacement component should exhibit the same behavior. The process of inferring an accurate high-level model was an explorative one due to the

limited knowledge about the system behavior. The diagram in Figure 7.5 depicts the process of constructing a high-level model that accurately reflects the intended architecture. Each rectangle illustrates an activity, which are executed according to the numbering.

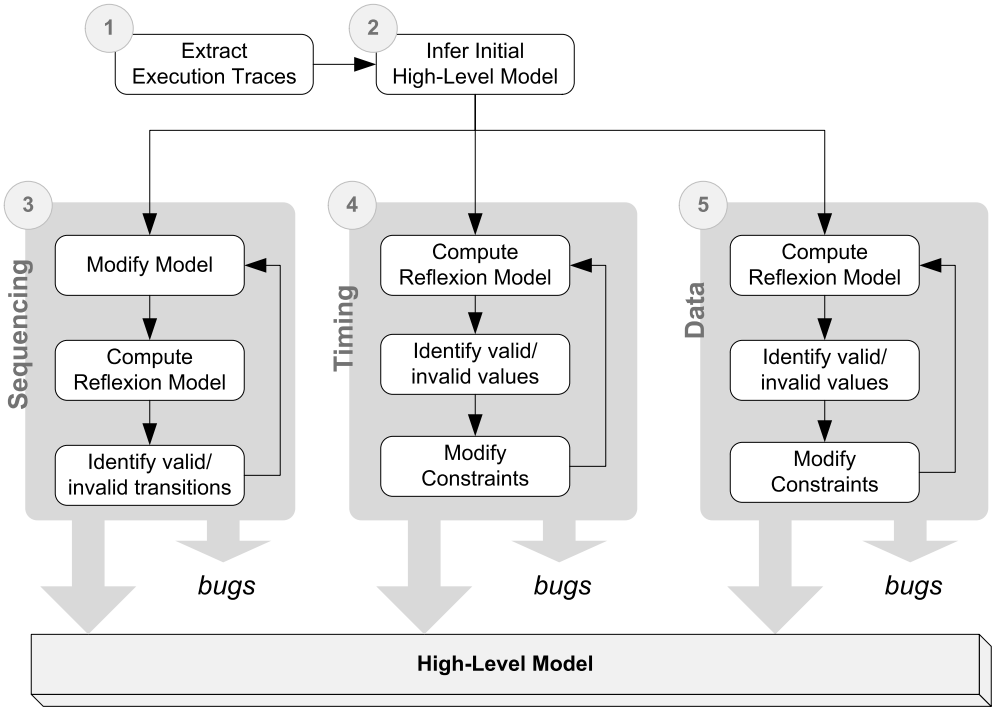


Figure 7.5: Process of applying the behavioral reflexion model approach to spacecraft/MOC communication.

After execution traces have been captured and extracted, an initial high-level model is automatically inferred using the machine-learning algorithms described in the previous chapters. In this, as in many other application of the behavioral reflexion model approach, we have found it useful to first focus our attention the sequencing model. The reason is that the sequencing model organizes the interactions whose events and parameters are referenced by timing and data constraints respectively.

The sequencing model that was inferred automatically was rather complex. Hence, our first goal was to remove the transitions that we believed to be erroneous and derive an accurate model of the interaction sequences. We removed low-probability transitions as we suspected them to be erroneous. However, after computing the reflexion model, we realized that valid transitions had been classified as anomalies. We adjusted the sequencing model to include the valid transition and recomputed the reflexion model. In that reflexion model, only anomalous transitions were classified as insertions or deletions. Thus, we were confident that the high-level model is accurate in respect to the intended architecture. The transitions that were identified as violations by the reflexion models were reported to the MOC system engineers.

Once we constructed an accurate sequencing model, we focused on analyzing the timing characteristics. With the initial high-level model, a set of timing constraints were automatically inferred. Our task was to identify whether the constraints accurately reflect the intended architecture. To do this, we considered the values that are excluded by the constraints and, thus, classified as anomalous. In order to identify these anomalies, we computed the reflexion model, which highlight constraint violations graphically. We were able to identify false negatives, which are valid values that are classified an anomalous. Such values are typically close to other valid values and occur in larger numbers. We modified the constraints to include these values. This is an iterative process in which the constraint is incrementally adjusted to include valid values. The process terminates when the constraint violations that are shown in the reflexion model are in fact anomalies. At that point, the constraints classify all values correctly and, thus, represent the intended

architecture.

The process of constructing accurate data constraints is similar to the timing constraint construction. Using the feedback of the reflexion model, we identified the false negatives and adjusted the constraints to include those. Upon termination, we had identified a set of accurate data constraints as well as values that we believed are due to errors in the implementation. We reported these to the MOC system engineers. This process illustrates how with a combination of inferring constraints and the feedback from the reflexion model, we can identify a high-level model that accurately reflects the intended architecture, at least to a limited extent.

7.4.4 Performance

One of the main challenges in behavioral analysis is the vast amount of information that must be processed. When collecting information via dynamic analysis, a large number of executions is often captured. Each of these executions often consists of a large number of interactions. In order to assess the scalability of the algorithms employed in the behavioral reflexion model approach, we measured the runtime of these algorithms with a varying number of interactions. We organized our performance analysis according to the components of our research tool shown in Figure 7.2. More specifically, we measured the performance of the following steps:

- Sequence Construction: The sequencing model is constructed using the k-tail algorithm. A FSM is constructed by merging executions traces. See Section 5.3.1 for details.

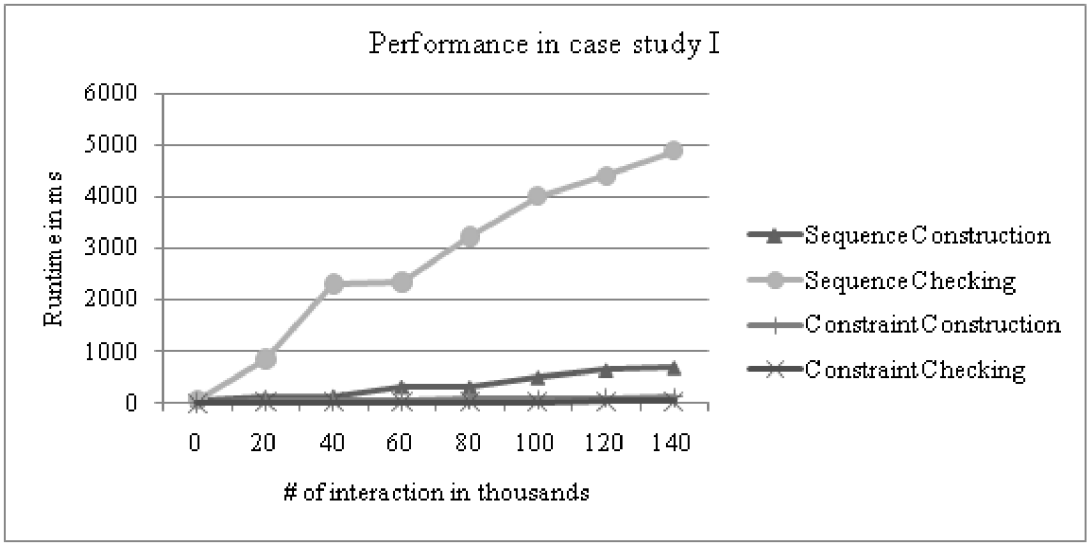
- Sequence Checking: Application of Cook's algorithms for the detection of sequencing violations. In that process, an alignment tree is constructed and a best-first search is conducted to identify an optimal alignment. Details are describes in Section 6.2.1.

- Constraint Construction: Timing and data constraints are constructed as described in Section 5.4. That includes collecting the values of each parameter and event-pair as well as inferring the constraint using outlier detection algorithms.

- Constraint Checking: All execution traces are parsed and the parameter as well as the timing values are checked for violations of a constraint. Details of this process are explained in Section 6.2.2.

Each of these algorithms involves the analysis of the execution traces. Hence, the runtime depends on the number of interactions. We executed the algorithms on all 276 execution traces, with a total of 141,223 interactions. The performance analysis was conducted on an Intel Core Duo CPU 2.20GHz processor with 4 GB RAM running a 32 Bit Microsoft Windows 7 operating system. The results are shown in the diagram below.

The graph shows that the checking of execution traces for sequencing deviations takes significantly more time than any other algorithm. At a maximum of 141,223 interactions, the runtime of the sequence checking algorithm is 5 seconds. In comparison, the maximum runtime of the other algorithms was as follows: (1) sequencing construction: 700ms, (2) constraint construction: 110ms, (3) constraint



checking: 36ms. As the reflexion model shown in Figure 7.4 indicates, the execution traces deviated significantly from the high-level model. Despite these discrepancies, the sequence checking algorithm was able to identify the optimal alignments of all execution traces in a reasonable amount of time. Most importantly, the graph illustrates that runtime of all algorithms linearly increase with the number of interaction.

7.4.5 Research Questions

In this case study, we have applied the reflexion model approach to construct an accurate model of the intended architecture from the system implementation with the goal to allow the system engineers to replace the legacy communication component with a new component. In the discussion above, we have addressed research questions 1, 3, 5, and 6.

Research question 1 aimed to evaluate whether the behavioral reflexion model approach can be used to align the documented architecture with the intended architecture. We have shown that we were able to reconstruct a high-level model from the

set of captured execution traces. Using machine-learning algorithms, we inferred an initial high-level model from the execution traces. Subsequently, with the feedback provided by the reflexion model, we modified and refined the high-level model to eventually derive a model that we believed reflects the intended architecture. The high-level model can be used to update existing documentation and, thus, align the documented architecture with the intended architecture.

In research questions 3 and 4, our goal was to evaluate whether the behavioral reflexion model approach has a positive impact on system qualities. In this case study, we addressed question 3, which is asking for improvements of the system maintenance. The task of the system engineers was to replace a legacy communication component with a new component. It was critical that the new component is compliant with the behavior of the legacy component. In order to verify compliance, an accurate model of the behavior exhibited by the legacy component was needed. We were able to produce such a model using our approach.

Research questions 5 aims to provide insight into what parameters yield a satisfactory high-level model. We discussed the measure of support as a means to guide the system engineer in modifying the sequencing model and derive a model that reflects the intended architecture. In this case study we have seen that two classes of transitions could be clearly identified. The support of low-probability transitions was less than 10%, while the support of high-probability transitions was more than 88%. Ideally, the first class would represent invalid transitions and the second class invalid transitions. However, two valid transitions were wrongly classified as invalid based on their support. The case study shows that a threshold

can be clearly identified but that the system engineer cannot rely entirely on the support for classifying transitions.

Finally, in research question 6, we aimed to address the performance of the behavioral reflexion model approach in the case studies. We have presented a performance evaluation, which showed that the runtime of all algorithms combined on all 276 execution traces was slightly more than 5 seconds. Furthermore, the analysis revealed that the runtime of the algorithms increases linearly with the number of interactions. We can, thus, report that our approach does scale in practice.

7.4.6 Observations

In this case study, we made several interesting observations. First, we have seen that the behavioral reflexion model approach can be used even if only partial behavioral information is available. The system engineers were only able to monitor the network interface of the MOC but not of the spacecraft. This only has a minor impact on the construction of the high-level model and the compliance checking. The inference of the sequencing model and the data constraints are entirely unaffected by the missing behavioral data. It is not possible to infer latency constraint that describe how long message travel on the network but we have shown that one can reason about the network latency using the available timing information.

Another issue related to the collection and extraction of execution traces is that some traces are incomplete as the monitoring was started or stopped while the execution was in progress. That means that interactions in the beginning of some

executions and the end of other executions are missing. A sequencing model that is inferred from such incomplete traces contains noise, which increases its complexity. Since interaction from few initial and last execution traces, the respective transitions in the sequencing model receive only a low support. Hence, they are removed when deleting low-probability transitions. Handling such noise in the data is crucial when applying the behavioral reflexion model approach in a real-world setting.

In addition to distilling a high-level model, we were also able to identify anomalous interactions sequences, events, and parameter values. While the sequencing anomalies were all due to noise in the execution trace sample, timing anomalies seemed to be caused by a problem in the implementation. We have identified several instances in which a component showed poor responsiveness, which decreases the performance of the system.

We also uncovered limitations of our approach. For instance, our approach is only able to check for certain types of errors, which might not be sufficient for all systems. In this case study, we were not able to detect problems in the transfer of files. The CFDP protocol transmits files in chunks. The communication system implements a mechanism for managing these chunks and ensuring that the file arrives entirely at the ground station. The CFDP protocol specifies rules about how these chunks must be re-transmitted in case of a data loss. Unfortunately, we were not able to identify violations of these rules as it requires the identification of each chunk and set operations in order to check for which chunks are missing and whether the files has been completely transmitted. In our approach, we have focused on checking properties that are common in behavioral specification and cause problems

if violated. The chunk management is specific to the CFDP protocol and is not addressed by our approach.

Prior to applying the behavioral reflexion model approach, the system engineers did not have means to identify the actual semantics of the system. Testing was applied to ensure that the system adheres to its functional requirements. However, by applying wrong configuration settings, the system could exhibit a behavior that is consistent with the requirements specification but might deviate from the desired behavior of a particular configuration. With the behavioral reflexion model approach, it is possible to reconstruct a model that describes the semantics of the legacy communication component. The model describes the behaviors that are exhibited with the current configuration. As such, it can be used to verify that the new component is consistent with the semantics of the legacy component. A high-level model may also serve to expose the actual behavior of a system and may be used by the system engineer to confirm consistency with the expected behavior.

7.5 Case Study II: Failure Analysis

In this case study, we focus on the ground communication of the space mission described above. Client systems that are developed and operated by external organizations are used to access telemetry data that is stored at the MOC. The protocol was specified and documented by the MOC system engineers. All clients must adhere to the protocol to ensure a reliable and efficient transfer of data.

7.5.1 Task

During the operation of the mission, the MOC system engineers received reports from the operators of client systems indicating a failure in the transmission of data. In one instance, an operator reported that the MOC was frequently unavailable. While the MOC has only a limited number of interfaces for clients to connect, it has been scaled to match the expected workload. A client should, therefore, not have difficulties to connect to the MOC under normal circumstances. In another instance, the MOC was claimed to unexpectedly terminate the connection to the client system at a specific point in the protocol. In general, the problems regarded issues of a lack of availability, connection problems, and transfer of ill-formatted data.

It was the task of the MOC system engineers to investigate what errors caused the system to fail. Furthermore, the system engineers had to identify what component in the system caused the failure. The system engineers approached the failure analysis by first recording the network communication between the client systems and the MOC. A system engineer then analyzed the unprocessed network traces manually. A network trace consisted of a series of blocks, each representing a packet, in hexadecimal formal. Only with thorough knowledge of the message format specification was the system engineer able to identify message types and parameters. The system engineer inspected each packet individually in an attempt to identify the errors that caused the system failures. While the system engineer was able to identify some of the failure causes, the analysis was cumbersome and time

consuming. In collaboration with the system engineers, we applied the behavioral reflexion model approach to analyze the captured communication that had been analyzed manually.

7.5.2 Application

The system engineers provided us with the protocol specification, which describes the message formats, rules on sequencing of interactions, and constraints on message parameters. In addition, the system engineers formulated constraints on the expected timing of events. That information should support us in constructing the high-level model.

7.5.2.1 Trace Extraction

During the system operation, the system engineers captured network traffic by monitoring the network interface of the MOC. However, it was not possible to monitor the network interface of client components. Hence, we had to conduct our analysis with only partial information. The monitored network traffic was stored in a set of files, each containing the traffic observed during a single system execution. We converted the network traffic into application-level execution traces.

7.5.2.2 Constructing a High-Level Model

The first task of our analysis was to construct a high-level model that describes the intended architecture. We used the previously described machine-learning algo-

rithms to infer an initial high-level model from the set of execution traces. We found that the model that is inferred from the execution traces is useful in comprehending the observed system execution as is presents a summary thereof. It is, thus, often beneficial to infer a model even if an accurate specification is available.

Figure 7.6 shows the sequencing model that was inferred from the captured execution traces. Each transition is annotated with a message type. Due to space reasons, parameter information is omitted. In addition, each transition is annotated with the support, indicating the fraction of execution traces that takes that transition.

We recognized parts of the model that resembled the rules describe in the protocol specification. However, many of the transitions were unexpected. For instance, the model shows that the first interaction in an execution is the exchange of a "Filter" message or a "Close" message. A "Close" message in the beginning of an interaction sequence was not defined in the protocol specification.

Such unexpected transitions are due to errors in the implementation that cause the system to deviate from the desired behavior. We have argued previously that erroneous transitions can often be identified using the support, i.e., the fraction of behaviors executing a certain transition. A transition that is executed only sporadically might be caused by an implementation error, while transitions with a high support are likely to be valid. In order to rid the model of invalid transitions, we removed those with a low-probability. While the probabilities provide guidance in modifying sequencing model, the system engineer must identify the threshold that divides low-probability and high-probability transitions. The threshold must be cho-

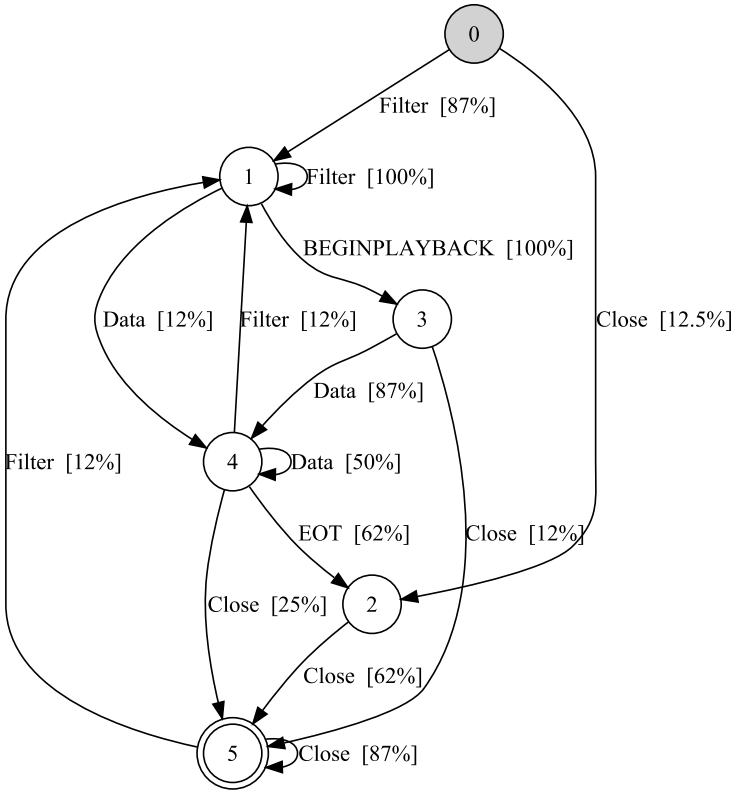
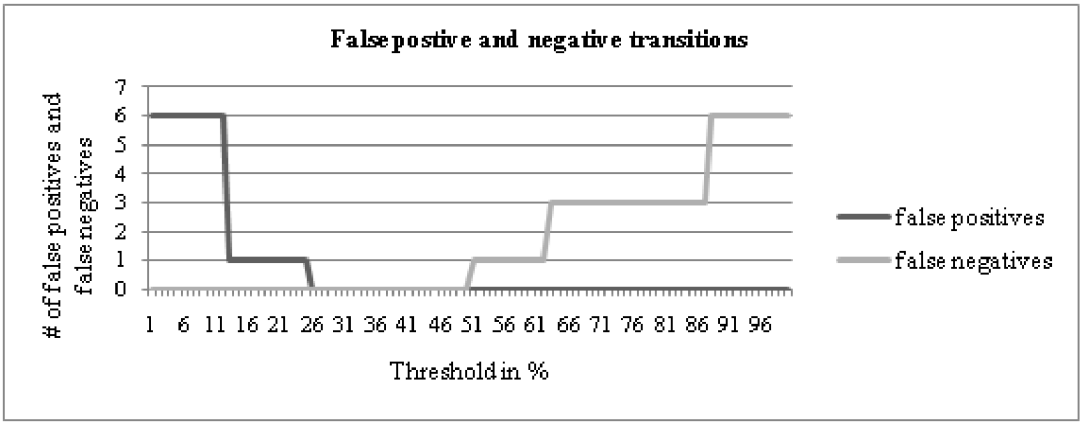


Figure 7.6: Inferred sequencing model for the interaction between the MOC and the client components.

sen so as to reduce the number of false positives and negatives. A false positive is an invalid transition that is retained in the model. A false negative is a valid transition that is removed from the model. The diagram below illustrates the number of false positives and negatives for the sequencing model in Figure 7.6 with increasing support threshold.



The diagram shows that the number of false positives decreases and the number of false negatives increases with a rising threshold. When setting the threshold to 0%, no transitions are removed from the model. Consequently, all invalid transitions are retained, resulting in a high number of false positives. As we increase the threshold, invalid transitions are removed until the model is free of false positives at a threshold of 27%. Only if the threshold exceeds 51% do we remove valid transitions and increase the number of false negatives. The diagram illustrates that there is a large range (27% - 51%) in which the number of ill-classified transitions is zero. In this sequencing model, the support is successful in separating valid from invalid transitions.

After deriving an accurate sequencing model, we focused on timing and data constraints that were inferred with the initial high-level model. First, we converted the model into sequence diagram representation as it allows for describing timing constraints in an intuitive manner. Figure 7.7 shows the sequence diagram expressing the modified sequencing rules and the timing constraints that were automatically inferred. Similar to the sequencing model, we adjusted the constraints to reflect the

indented architecture, supported by the protocol specification. For instance, one of the timing constraints expresses that the time that elapses between sending the "EOT" and receiving the "ClientClose" message must not exceed 15 seconds and 453 milliseconds. However, the system engineers specified the upper bound to be 1 second. A data constraint that was not consistent with the protocol specification was "reqType={STF,STP, TP}". According to the specification, the parameter "reqType" was only allowed to take the values "STP" and "TP". We adjusted the constraints to establish consistency of the high-level model with the protocol specification.

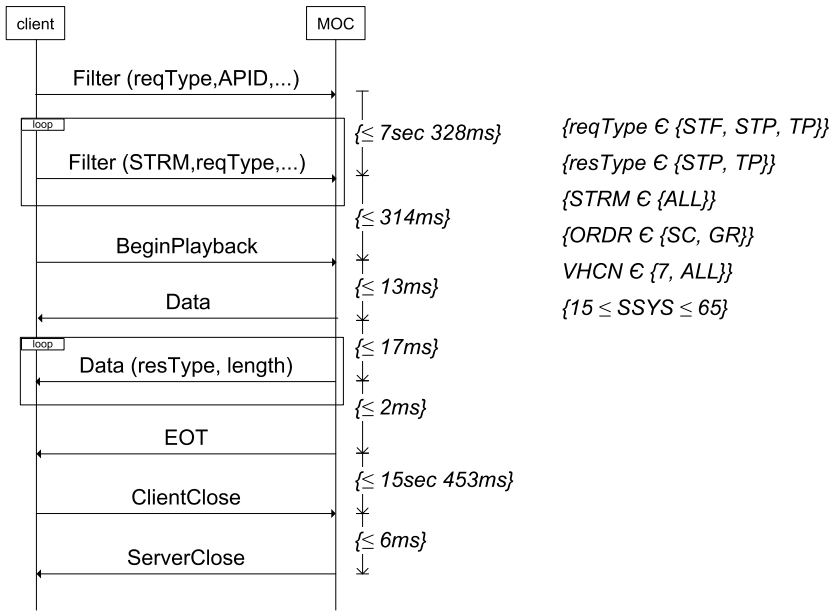


Figure 7.7: High-level model of the MOC/client communication.

We also added additional constraints expressing rules that were crucial for the correct behavior of the system according to the MOC system engineers. One of the rules expresses a relationship between the data type requested by the client and the

type of the data sent in response by the MOC:

$\{reqType = resType\}$

The constraint is needed to check that the MOC satisfies the request of the client. More specifically, it ensures that the type of the telemetry data sent by the MOC is consistent with the type requested by the client component. By adding missing constraint, we finished the construction of the high-level model. In this case study it was not necessary to check the high-level model for accuracy as we received the protocol specification describing the intended architecture from the system engineers.

7.5.2.3 Computing a Reflexion Model

With an accurate high-level model we were able to analyze the observed execution traces for violations. We computed the reflexion model showing the deviations betweens the execution traces and the sequencing, timing, and data rules. Te resulting reflexion model is shown in Figure 7.8.

At first glance, the reflexion model illustrates in general how well the executions that were observed from the system implementation match the high-level model that describes the intended architecture. The reflexion model above shows that several violations of sequencing, timing, and data rules were detected. In the next step, we analyzed the violations in more detail to identify and reason about the concrete errors that caused the system failures. Each violation refers to one or more behaviors in which the violation occurred. In order to investigate concrete vi-

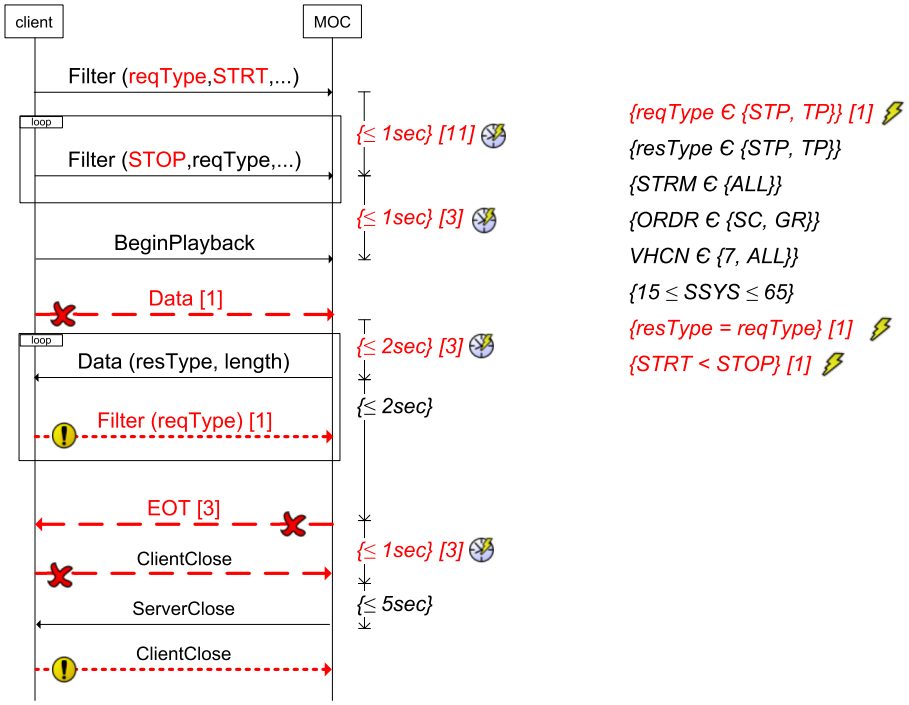


Figure 7.8: High-level reflexion model of the MOC/client communication.

olations, we constructed behavior reflexion models that illustrate the details about the violations. More specifically, they show the concrete situation or context in which the violation occurred and the values that caused it. With that information, we hoped to identify if relationships exist between violations in order to ultimately identify the original source of the problem.

7.5.2.4 Investigating Sequencing Violation

In one instance, we focus on the deletion of the "EOT" interaction. From the high-level model, we can observe that there are 3 deletions of the "EOT" interaction in all of the executions. To analyze a particular deletion, we compute a behavior reflexion model that shows only the behavior in which the violation occurs. That

reflexion model is shown in Figure 7.9.

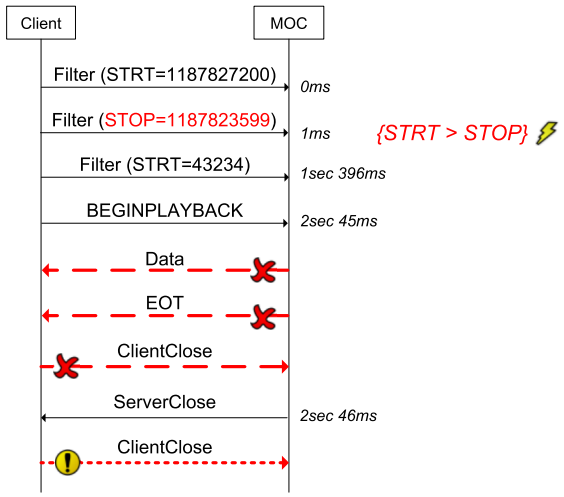


Figure 7.9: Behavior reflexion model showing a sequencing violation.

The behavior reflexion model reveals the concrete context in which the violation occurs. It shows that this particular system execution violates several sequencing and data rules. Not only is the "EOT" interaction missing but also the preceding "Data" interaction and the succeeding "ClientClose" interaction. In the interaction sequence described by the high-level model, the MOC sends a series of "Data" messages upon receiving a "BEGINPLAYBACK" message and a then sends a "EOT" message to indicate the end of transmission. After receiving the "EOT" message, the client closes the connection first and the MOC follows. However, in this scenario, the MOC did not send a "Data" or "EOT" message. I also did not wait for the client to close the connection. Instead, it abruptly closed the connection after receiving the "BEGINPLAYBACK" message. The client follows by also closing the connection. When discussing this problem with the system engineers, they pointed out that the unexpected behavior of the MOC can be explained by

the violation of the data constraint in the beginning of the execution. When formulating a query using "Filter" messages, the client defines a time frame for which it requests telemetry data. The time frame is described by a lower and upper bound as specified in the parameters "STRT" and "STOP" respectively. In order for the time frame to be valid, it is necessary that the beginning of the time frame is before the end of the time frame. That is expressed with the constraint "STRT<STOP". In this execution, the client violates that rule and provides invalid parameter values. The system engineer explained that the MOC was not able to process the invalid request and reacted by terminating the connection as soon as the client requested to start sending data. The error in the client system must be removed in order to resolve the data and sequencing violation. This shows that interdependencies do not only exist among errors of the same type but across different dimension. The example also illustrates that the reflexion models can be used to identify the source of the problem even if it is not immediately obvious.

7.5.2.5 Investigating Timing Violation

In addition to sequencing violations, the reflexion model shows a significant number of violations of timing constraints. For instance, several violations were detected of the timing constraint that specifies an upper bound on the time that elapses between the "EOT" interaction and the "ClientClose" interaction. We again computed the behavior reflexion model of the behavior that violated that constraints. The reflexion model is shown in Figure 7.10.

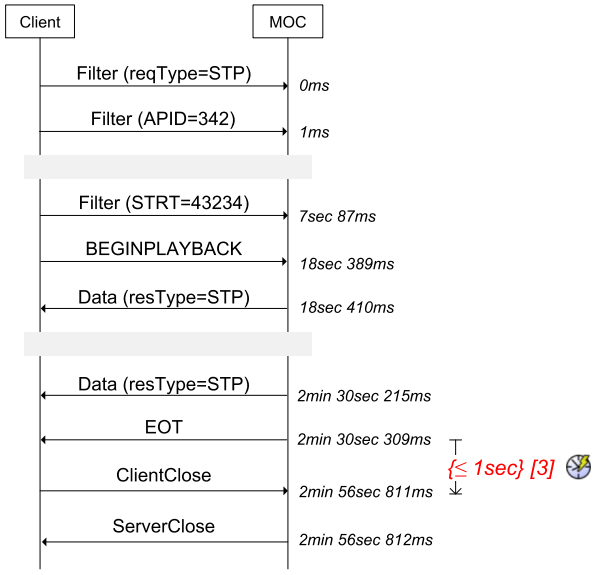


Figure 7.10: Behavior reflexion model showing the timing constraint violations.

The behavior reflexion model shows that the violation of the timing constraint is the only violation in that execution. From the model we can see that the "Client-Close" message was received 26 seconds and 502 milliseconds after the "EOT" message was sent, which violates the specified timing constraint. In between the two events, two messages are transmitted across the network and one message is processed by the client component. Thus, the violation could be due to a poor response time of the client component after receiving the "EOT" message or could be caused by a high latency of the network. Even with a poor latency of the network, such a delay is unusual. We suspected that the response time of the client system is at least partially responsible for the timing constraint violations. The timing constraints were defined to ensure that client would not unnecessarily occupy the MOC as it has only a limited number of access points for clients. In order to ensure a sufficient availability of the MOC, the violations had to be further analyzed in collaboration

with the system engineers of the respective client systems.

7.5.2.6 Investigating Data Violations

Finally, by comparing the execution traces to the high-level model, several violations of data constraints were detected as well. To further investigate concrete violations, we again compute the reflexion model of an execution trace in which a data constraint was violated. The constraint, we focused on was *reqType=resType*, which was added manually to the high-level model as it is crucial that the type of the data sent by the MOC is consistent with the data type requested by the client. Figure 7.11 shows the behavior reflexion model for the execution trace in which the constraint was violated.

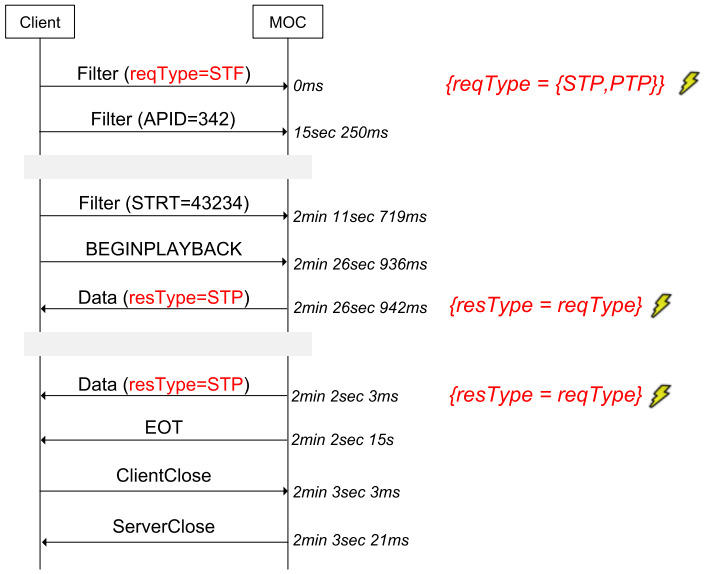


Figure 7.11: Behavior reflexion model showing the data constraint violation.

The behavior reflexion model reveals details about the violation, such as the value of the parameters. The model shows that the value of "reqType" is "STF",

while the value of “resType” is “STP”. Furthermore, the model shows the violation of another constraint in the same behavior. The constraint “reqType={TP,STP}” specifies the valid values of that parameter. The value that was observed for reqType in this particular behavior was not specified as part of this constraint and is, thus, considered invalid. We discussed this violation with the MOC system engineers. They explained that the data type “STF” was used in the past but is no longer supported by the MOC. When the client requested data of that type, the MOC ignored the invalid value and used the default data type ”STP” instead. The MOC system engineers pointed out that that is an undesired behavior. The MOC should have terminated that file transfer immediately upon receiving an invalid “Filter” message.

7.5.3 Process Discussion

In this case study, we have described the application of the behavioral reflexion model approach for the purpose of investigating system failures. The case study was motivated by issue reports that were submitted to the MOC system engineers by operators of client systems. The intended architecture was described in the protocol specification. The main challenge in this application was to identify the errors that led to the failures and their source. The application focused not on the construction of the high-level model but on the evaluation of errors that are identified in the reflexion model. Figure 7.12 illustrates that process.

After the traces had been extracted, we used the machine-learning algorithms

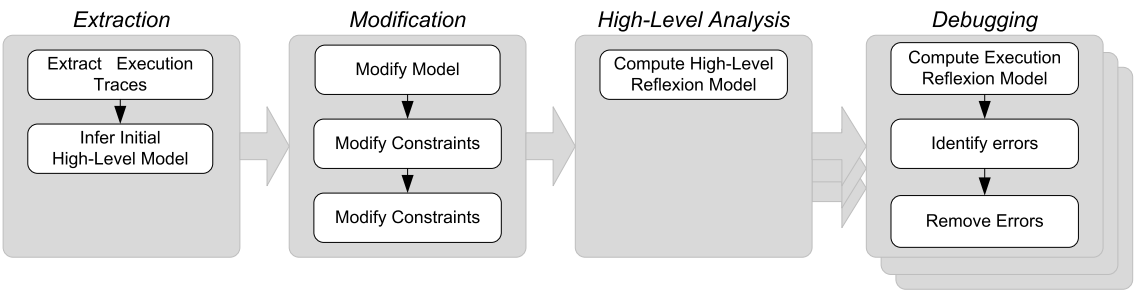


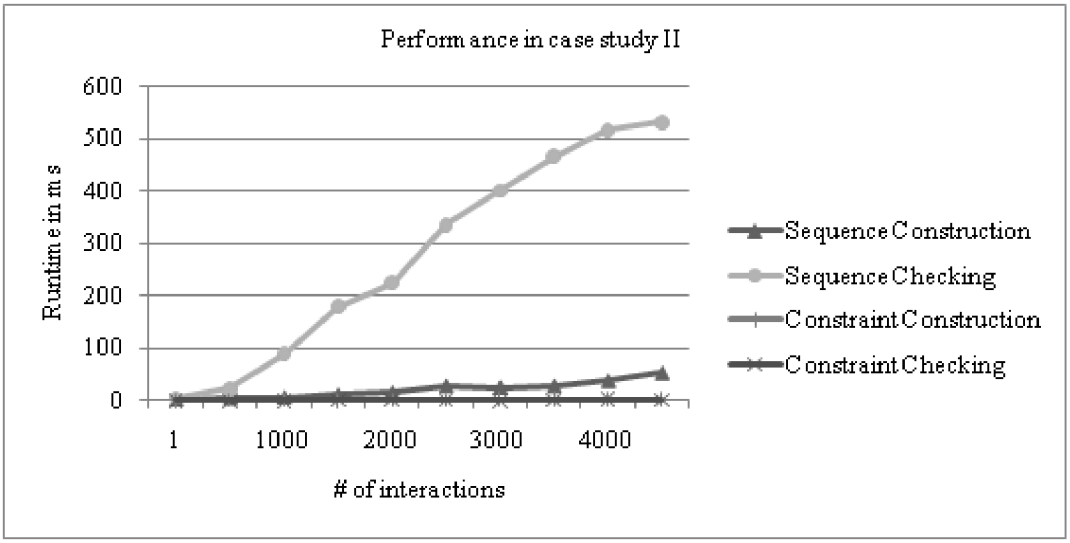
Figure 7.12: Process of applying the behavioral reflexion model approach to the MOC/spacecraft communication.

to infer an initial high-level model. Some of the rules in the model were not accurate. We used the protocol specification to adjust the high-level model and align it with the intended architecture. Subsequently, we computed a high-level reflexion model to gain insight in the types and number of deviations. In order to resolve the implementation errors, we analyzed each of the deviations further. More specifically, we computed a behavior reflexion model for each execution trace that violated the sequencing model or a constraint. Using the details presented in the behavior reflexion model, the system engineers were able to reason about the errors and relate them to failures that had been reported by client systems.

7.5.4 Performance

Previously, we have presented the performance analysis when applying the reflexion model approach to the execution traces of case study I. The graph below illustrates the same performance measures for the execution traces of case study II. In that study, we applied our approach to 8 execution traces and 4451 interactions. As in the previous performance analysis, we assessed the runtime of the sequence

construction, sequence checking, constraint construction, and constraint checking algorithms. For details about each of these steps see Section 7.4.4.



While the number of total interaction is significantly smaller than in case study I, we can observe a similar behavior when applying our approach to the execution traces in case study II. In particular, the checking for sequencing deviations consumes the largest fraction of the overall running time (533ms). The runtime of all other algorithms is significantly smaller. It is interesting to note that all of the execution traces deviated from the high-level model. Despite these discrepancies, the sequence checking algorithm performed well. Also, the runtime of all algorithms increased linearly with the number of interactions.

7.5.5 Research Questions

In this case study, we applied the behavioral reflexion model approach to identify potential deviations of the system implementation from the intended architecture. During the operation of the system, the system engineers received notice

of reliability and availability problems. Pour approach should help identify problems that caused these issues. With this case study, we aimed to address research questions 2 and 4.

In research question 2, we asked whether our approach can be used to align the implemented architecture with the intended architecture. In this case study, we have described how we constructed a model of the intended architecture using the captured execution traces as well as the protocol specification. We then computed a high-level reflexion model that provided a summary of all deviations between implemented architecture (represented by the execution traces) and intended architecture (described by the high-level model). Hence, our approach is able to identify discrepancies between these two architecture representations. After computing the high-level reflexion model, we further investigated deviations using behavior reflexion models, which provided more details of the deviations. Based on that additional information, the system engineers were able to reason about deviations, identify interdependencies between them, and ultimately identify the source of the problem. With that information, the MOC system engineers could take appropriate actions to resolve the problem and, thus, to align the implemented architecture with the intended architecture.

We aimed to evaluate the impact on the system in research question 4. There we asked whether the application of the behavioral reflexion model approach leads to increased reliability. This case study was motivated by reliability and availability issues that were reported to the MOC system engineers. By applying the behavioral reflexion model approach, we were able to identify discrepancies between

implemented and intended architecture. These discrepancies were identified by the MOC system engineers to be the cause of the reported issues. By resolving the discrepancies, the system's availability and reliability will be increased.

Furthermore, we have provided a detailed discussion to address research question 5, which asks what parameter values must be chosen to produce a satisfactory high-level model. We have illustrated how the number of falsely classified transitions changes as the threshold for the support is modified. By setting the threshold between 27% and 51%, we were able to divide low-probability from high-probability transitions. In this case study, the support showed to be successful in distinguishing between valid and invalid transitions. In other words, all low-probability transitions were in fact invalid and all high-probability transitions were valid.

Finally, we have discussed the performance of the behavioral reflexion model approach for increasing number of interactions. While only few traces were collected, the results showed that the runtime of the algorithms increases linearly. This confirms the performance results of case study I and provides us with additional evidence that our approach is scalable.

7.5.6 Observations

Using the behavioral reflexion model approach, we were able to identify violations of the rules formulated in the protocol specification and, thus, uncover invalid system executions. Using the high-level reflexion model as starting point, we were able to systematically investigate concrete violations. The behavior reflexion models

allowed us to analyze concrete violations in detail and reason about them. Using the information presented in the reflexion models, the system engineers were able to explain why the errors occurred, what component caused the original error, and what failure the errors resulted in. This knowledge is crucial when debugging a system.

The behavioral reflexion model approach significantly enhances the process that was previously in place. In that process, the system engineers manually investigated traces of captured network traffic. Such an analysis requires in-depth knowledge of the protocol as the system engineer must interpret the hexadecimal codes of the network packets. Viewing the interaction behavior on application-level allows the system engineer unfamiliar with the specifics of the protocol to reason about it. Furthermore, the behavioral reflexion model approach is more reliable and less time consuming that the state of the practice. Since only information is illustrated that is needed to reason about errors, the system engineer is less prone to oversee undesired behaviors. Furthermore, using the systematic analysis starting with the high-level reflexion model and then the behavioral reflexion models, the system engineer can more quickly navigate through the traces, focusing on the points of interest.

A limitation was that only the events at the communication interface of the MOC could be observed. This allowed us to recover a partial model and also check specific constraint violation. However, the case study shows that it is possible to not only recover the majority of constraints but it also illustrated that many of the critical violations can be detected. This is important if the approach is to be applied

in a real-world environment in which not all required data can be collected.

A limitation of our approach in the context of this case study was the inability to check for data constraints on more complex data types. Two of the parameters specify a time. Rules should have been formulated describing relationships between the two parameters. Instead of using a more complex data type, we expressed the time as a UNIX timestamp and treated it as a numeric data constraint. The disadvantage of this approach is that the representation of time to the system engineer is less intuitive and requires knowledge about the UNIX timestamp format.

7.6 Summary

In this chapter, we have described the application of the behavioral reflexion model approach on two case studies. In both studies, we applied the approach to a space mission system, which are currently in operation. We provided detailed descriptions of concrete applications of the behavioral reflexion model approach. Furthermore, we addressed issues, such as incomplete data and noise, which occur in real-world applications.

The goal of our approach is to support the system engineer in identifying and resolving inconsistencies between architecture representations. In the first case study, we illustrated how we can use the automatic inference combined with the feedback provided by reflexion models to construct a high-level model that describes the valid system semantics. The high-level model can served as documentation, which is consistent with the intended architecture. The second case study illustrated

how deviations of an implemented architecture from an intended architecture can be identified. In addition to illustrating the deviations, reflexion models also allowed the system engineer to reason about interdependencies among errors and identify the source of the problem.

Depending on the task at hand, the application of the behavior reflexion model approach may differ. The case studies have shown two strategies for applying our approach. When constructing a high-level model with little knowledge about the system, the machine-learning algorithms are used to construct an initial model and the reflexion models are used to evaluate the inferred sequencing, timing, and data rules. That is, the feedback from the reflexion models is used to refine the high-level model. When we used the behavioral reflexion model approach to investigate failures, we constructed a high-level model and then used the reflexion model to investigate problems in the implementation. More specifically, we reasoned about interdependencies of errors and identified their source. Reflexion models allow the system engineer not only to identify discrepancies between intended, documented, and implemented architecture. They also play a central part in supporting the system engineer in resolving them.

Chapter 8

Conclusions and Future Work

In this dissertation we have presented the behavioral reflexion model approach with the goal to resolve inconsistencies among architecture representations describing behavioral system characteristics.

In the behavioral reflexion model approach, a high-level model is specified defining rules on the sequencing of interactions, the timing of events, and the values of message parameters. Then, execution traces are extracted from the system implementation via dynamic analysis. The execution traces are subsequently checked for compliance with the rules specified in the high-level model. Violations of these rules are finally presented graphically in a reflexion model.

8.1 Contributions

With this research we have developed a set of theories and tools that allows the system engineer to apply the behavioral reflexion model approach in practice.

In chapter 3, we have described an approach that we have developed to extract execution traces from distributed systems, which each illustrate a single execution of a system on an abstract level. While an array of method was available for extracting structural and behavioral information from centralized systems, there was a lack of methods for extracting models that describe the interaction behavior of an entire

distributed system on application level. In developing our extraction approach, we have introduced the notion of distributed observations points, which capture raw runtime information. In a series of post processing steps, the individually captured data is consolidated to a single model. In that process, issues of clock drift and logical interaction ordering are addressed. With this method, we were able to apply our approach to distributed systems, which are particularly in need for support of resolving issues in their interaction behavior. We hope that the extraction method will find application in other areas and, thus, contribute to enriching the set of tools and techniques for the tackling reliability, integrability, and maintainability issues that plague many distributed systems.

Similar to verification methods in general, an input to the behavioral reflexion model approach is a description of the desired system state, or in other words, the specification. We have discussed the difficulty of describing the desired system characteristics if accurate documentation is lacking and experienced developers have left the team. In chapter 5, we have presented an approach for inferring a model of the behavioral specification (i.e., a high-level model) from the execution traces that have been extracted from the implementation. In the first step, a model is constructed that describes the implemented sequencing, timing, and data rules that constitute a high-level model. The automatic inference of a high-level model is similar to traditional reverse engineering methods, which extract abstract models from system implementations. However, in a second step, parts of the high-level model that appear anomalous are removed and measures are computed that indicate the confidence in different parts of the model. These measures should guide the

system engineer in refining the high-level model manually.

The main contribution emerging from this research is the introduction of behavioral reflexion models as a means of making system engineers aware of inconsistencies among behavioral system descriptions. The idea of reflexion models was introduced by Murphy et al. [mur95] for the purpose of identifying structural deviations of the implementation from an expected system state. Especially with the widespread deployment of distributed systems and the array of reliability and maintainability issues that plague them, a method was needed that allows system engineers to identify behavioral problems. We, thus, adapted the idea of reflexion models for highlighting discrepancies in the behavior of the implementation and the expected behavior. In order to support different stages of the analysis, we specified a high-level reflexion model, which summarizes the deviations found in all execution traces and a trace reflexion model for illustrating more detailed information about the violation caused by a single execution trace. Furthermore, we have specified the FSM reflexion model to concisely describe more complex sequence semantics. Reflexion models serve a versatile role in resolving inconsistencies among behavioral architecture descriptions. By representing discrepancies in the context of both the high-level model and the execution trace, the system engineer can reason about them, identify relationships between violations, and identify the source of the problem. This insight is helpful in identifying inaccuracies in the high-level model as well as detecting undesired behaviors in the implementation. By resolving these deviations, the system engineer aligns the intended, implemented, and documented architectures and ultimately establishes consistency among them.

We have applied the behavioral reflexion model approach to a variety of systems for solving different objectives. In this dissertation, we have described two case studies in which we applied our approach to two space mission systems. The goal of the first case study was to reconstruct a high-level model of the intended system semantics. In the second case study, we employed the behavioral reflexion model approach to resolve reliability problems. More specifically, we used reflexion models to identify behaviors that deviated from the expected behaviors. We then utilized the graphical representations of the deviations to reason about relationships between violations. We showed that it is possible to use our approach to not only detect undesired behaviors but also provide the system engineer with sufficient information for resolving them.

8.2 Future Work

In the application of our approach to different types of systems, we have also identified shortcomings of our approach that may be the subject of future research. One limitation that we encountered is the set of rules that can be specified in the high-level model and whose violations can be identified. Our goal was to describe a set of rules that is applicable to many systems and which describe aspects that are essential to ensure the correctness and the quality of a wide range of systems (see Chapter 5 for more details). However, just as each system is unique, so are the types of rules that one would specify in the high-level model. In a concrete case, a component was transmitting data files, which were divided up in several messages.

The objective was to formulate rules that specify that a file was completely transferred and no messages were missing. This would have required more advanced rules than the ones currently supported. One solution could be to introduce a notion of sets and operations on sets. Furthermore, a mechanism would be needed to identify what messages belong to a specific set. Checking for the complete transfer of a file could then be translated into a check that the set of messages that have been submitted and the set of expected messages are equal. However, since this problem arose only for one specific system, we did not support it in our approach. In the future, one may decide to add support for additional types of rules.

In our research and in the design of the behavioral reflexion model approach we have focused primarily on distributed systems. This is mainly due to the high frequency of problems in distributed systems, which can be linked to behavioral issues. Hence, the need for a method that aids in resolving such undesired behaviors seemed greatest for distributed systems. This is not to say that there is no need for such an approach in centralized systems. We believe that the basic idea of the reflexion model approach as well as the rules specified in the high-level model applies to centralized systems. However, in our research we have found that intra-system behaviors are often more complex than inter-system behaviors. This may at least partially be attributed to the fact that transmitting messages across a network in a distributed system is resource intensive. Protocols, thus, typically aim to reduce the number of interactions between components. In centralized systems, components interact more often and in a less well-defined manner. We believe that this poses a challenge when inferring an initial high-level model. The parameters of the employed

machine learning algorithms may have to be adjusted or even replaced by more suitable techniques. In the future, the applicability of our current approach to centralized systems could be evaluated and the approach could be optimized for that type of system.

One of the greatest challenges in comprehending and verifying systems is handling concurrent behaviors. When we applied our approach to different systems, we split up concurrent behaviors into individual execution traces, resulting in a set of non-concurrent traces. This process was done manually and may be time consuming for some systems. In the future, additional mechanisms might be introduced that allow for analyzing concurrent behaviors. More specifically, current algorithms for inferring an initial high-level model could be replaced with those that can handle concurrency. This particularly pertains to the inference of sequencing rules (Chapter 5), which, we believe, will benefit most from such an enhancement. Furthermore, the mapping procedure that is part of the compliance checking method would need to be adapted. Finally, the algorithm for identifying sequencing violations would need to be replaced with a suitable alternative.

Various algorithms are employed for inferring a high-level model and computing reflexion models. While these algorithms have shown to perform well for the systems that we analyzed as part of this research, their performance and accuracy might have to be improved when applying the behavioral reflexion model approach in a different context or to different types of systems. We could see several parts of our approach that could benefit from such an improvement. For instance, we employ a approximate matching approach for mapping execution traces to a high-level

model that only takes into account the sequence of interactions (see Chapter 6 for details). An improvement could be to also consider the impact of timing and data values on the alignment. More specifically, instead of only reducing the number of sequencing violation in the mapping, one could try to reduce the total number of sequencing, timing, and data violations. Additionally, the confidence measures for the inferred high-level model could be improved (see Chapter 5). For example, only the number of traces executing a transition is used to compute the support of a transition in the inferred FSM. The transition support could be computed based on the probability of traces traversing certain paths rather than only focusing on individual transition. That strategy might more accurately identify invalid transitions. With these and other modifications, the behavioral reflexion model approach could be optimized to perform well not only on different types of software systems but possibly also in related areas of research, such as process conformance.

While this research presents a significant step towards addressing behavioral issues in today's increasingly large and complex software systems, we hope it sparks many more initiatives to explore opportunities for improving the quality of both distributed and centralized software systems.

Bibliography

[1] Christopher Ackermann and Mikael Lindvall. Understanding change requests to predict software impact. In *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 66–75, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, New York, NY, USA, 2002. ACM.

[4] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[5] Olivier Barais, Anne-Franoise Le Meur, Laurence Duchien, and Julia Lawall. Software architecture evolution. 2008.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[7] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on UPPAAL. *Formal Methods for the Design of Real-time Systems*, 2004.

[8] Irad Ben-gal. Outlier detection. 2005.

[9] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, New York, NY, USA, 1994. ACM.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[11] Borland. *Borland Together*.

[12] L.C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, pages 642–663, 2006.

[13] LC Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of uml sequence diagrams. *Proceedings. 10th Working Conference on Reverse Engineering (WCRE) 2003*, pages 57–66, 2003.

[14] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[15] David Budgen. *Software Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[16] G. Canfora and M. Di Penta. New frontiers of reverse engineering. *Future of Software Engineering*, 2007.

[17] Lawrence Chung and Julio Cesar Prado Leite. On non-functional requirements in software engineering. pages 363–379, 2009.

[18] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *WORDS '97: Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems - (WORDS '97)*, page 199, Washington, DC, USA, 1997. IEEE Computer Society.

[19] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with brutus. *ACM Trans. Softw. Eng. Methodol.*, 9(4):443–487, 2000.

[20] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[21] Jonathan E. Cook and Alexander L. Wolf. Software process validation: quantitatively measuring the correspondence of a process to a model. *ACM Trans. Softw. Eng. Methodol.*, 8(2):147–176, 1999.

[22] George F. Coulouris and Jean Dollimore. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[23] D. Drusinsky and M.T. Shing. Monitoring temporal logic specifications combined with time series constraints. *Journal of Universal Computer Science*, 9(11):1261–1276, 2003.

[24] Doron Drusinsky. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[25] Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke, Ursula Goltz, and Technische Universitt Braunschweig. Timed sequence diagrams and tool-based analysis a case study. In *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins*, pages 645–660. Springer, 1999.

[26] Consultative Committee for Space Data Systems. Ccsds file delivery protocol (cfdp) part 2: Implementers guide. 2007.

[27] D. Ganesan, T. Keuler, and Y. Nishimura. Architecture compliance checking at runtime: An industry experience report. *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pages 347–356, Aug. 2008.

[28] Mohamed G. Gouda. Protocol verification made simple: a tutorial. *Comput. Netw. ISDN Syst.*, 25(9):969–980, 1993.

[29] Dan Gusfield. Algorithms on strings, trees, and sequences: Computer science and computational biology. *SIGACT News*, 28(4):41–60, 1997.

[30] Ahmed E. Hassan and Richard C. Holt. Architecture recovery of web applications. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 349–359, New York, NY, USA, 2002. ACM.

[31] R. Hofman and U. Hilgers. Theory and tool for estimating global time in parallel and distributed systems. pages 173–179, Jan 1998.

[32] R. A. Johnson and D. W. Wichern, editors. *Applied multivariate statistical analysis*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[33] A. Knapp and S. Merz. Model checking and code generation for uml state machines and collaborations. In *Proceedings of the 5th Workshop for Tools for System Design and Verification*, pages 59–64, 2002.

[34] J. Knodel, M. Lindvall, D. Muthig, and M. Naab. Static evaluation of software architectures. *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, 00:10 pp.–, March 2006.

[35] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, pages 12–12, Jan. 2007.

[36] R. Koschke and D. Simon. Hierarchical reflexion models. *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 36–45, Nov. 2003.

[37] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[38] Johns Hopkins Applied Physics Laboratory. *Mercury Surface, Space Environment, Geochemistry, and Ranging (MESSENGER)*, 2009.

[39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[40] Mikael Lindvall, Roseanne Tesoriero, and Patricia Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 77, Washington, DC, USA, 2002. IEEE Computer Society.

[41] Quan Long, Zhiming Liu, Xiaoshan Li, and He Jifeng. Consistent code generation from uml models. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 23–30, Washington, DC, USA, 2005. IEEE Computer Society.

[42] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, Jan 2000.

[43] Katharina Mehner. Javis: A uml-based visualization and debugging environment for concurrent java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 163–175, London, UK, 2002. Springer-Verlag.

[44] J. Moc and D.A. Carr. Understanding distributed systems via execution trace data. *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 60–67, 2001.

[45] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, New York, NY, USA, 1995. ACM.

[46] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, 1993.

[47] E.W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[48] C. Neumann. Converting Deterministic Finite Automata to Regular Expressions. 6. 2005.

[49] C. G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, New Zealand, May 1996.

[50] G. Oliver O. Rainer and S. Markus. Visusniff: A tool for the visualization of network traffic. *Proceedings of the Second Program Visualization Workshop*, pages 118–124, 2002.

[51] J. S. Oakland. *Statistical Process Control*. Butterworth-Heinemann, Woburn, MA, 1996.

[52] A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[53] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[54] Alessandro Orso, James Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed software. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 67–ff, New York, NY, USA, 2003. ACM.

[55] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: a yacc for writing application protocol parsers. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 289–300, New York, NY, USA, 2006. ACM.

[56] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[57] A. Postma. A method for module architecture verification and its application on a large component-based system. *Information and Software Technology*, 45(4):171–194, 2003.

[58] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, 1996.

[59] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington, and Carol Britton. Uml class diagram syntax: an empirical study of comprehension. In *APVis '01: Proceedings of the 2001 Asia-Pacific symposium on Information visualisation*, pages 113–120, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[60] O. Raz, R. Buchheit, M. Shaw, P. Koopman, and C. Faloutsos. Detecting Semantic Anomalies in Truck Weigh-in-Motion Traffic Data Using Data Mining. *Journal of Computing in Civil Engineering*, 18:291, 2004.

[61] S.P. Reiss and M. Renieris. Encoding program executions. *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 221–230, May 2001.

[62] Fulvio Risso and Mario Baldi. Netpdl: An extensible xml-based language for packet header description. *Computer Networks*, 50(5):688 – 706, 2006.

[63] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[64] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE Softw.*, 23(2):31–39, 2006.

[65] D.P. Sidhu and T.-K. Leung. Formal methods for protocol testing: a detailed study. *Software Engineering, IEEE Transactions on*, 15(4):413–426, Apr 1989.

[66] W.C. Stratton, D.E. Sibol, M. Lindvall, and P. Costa. Technology infusion of save into the ground software development process for nasa missions at jhu/apl. *Aerospace Conference, 2007 IEEE*, pages 1–15, March 2007.

[67] R. N. Taylor, Nenad Medvidovi, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, January 2009.

[68] J.J.P. Tsai and T. Weigert. A logic-based requirements language for the specification and analysis of real-time systems. *Object-Oriented Real-Time Dependable Systems,1996. Proceedings of WORDS '96., Second Workshop on*, pages 8–16, Feb 1996.

[69] Octavian Udrea, Cristian Lumezanu, and Jeffrey S. Foster. Rule-based static analysis of network protocol implementations. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[70] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119, 2002.

[71] Mahadevan Venkatraman and Munindar P. Singh. Verifying compliance with commitment protocols. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, 1999.

[72] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. *SIGPLAN Not.*, 33(10):271–283, 1998.

[73] CH West. General Technique for Communications Protocol Validation. *IBM Journal of Research and Development*, 22(4):393, 1978.

[74] Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, page 304, Washington, DC, USA, 2001. IEEE Computer Society.

[75] Mike Wooldridge and Michael Fisher. Agent-based software engineering. 1994.

[76] Jie Wu. *Distributed System Design.* CRC Press, Inc., Boca Raton, FL, USA, 1998.

[77] S. S. Yau, J. S. Collofello, and T. M. MacGregor. Ripple effect analysis of software maintenance. pages 71–82, 1993.