Towards Behavioral Reflexion Models

Christopher Ackermann, Mikael Lindvall, Rance Cleaveland
Fraunhofer USA – Center for Experimental Software Engineering
College Park, MD, USA
{cackermann, mlindvall, rcleaveland}@fc-md.umd.edu

***Abstract*— Software architecture has become essential in the struggle to manage today's increasingly large and complex systems. Software architecture views are created to capture important system characteristics on an abstract and, thus, comprehensible level. As the system is implemented and later maintained, it often deviates from the original design specification. Such deviations can have implication for the quality of the system, such as reliability, security, and maintainability. Software architecture compliance checking approaches, such as the reflexion model technique, have been proposed to address this issue by comparing the implementation to a model of the systems' architecture design. However, architecture compliance checking approaches focus solely on structural characteristics and ignore behavioral conformance. This is especially an issue in Systems-of-Systems. Systems-of-Systems (SoS) are decompositions of large systems, into smaller systems for the sake of flexibility. Deviations of the implementation to its behavioral design often reduce the reliability of the entire SoS. An approach is needed that supports the reasoning about behavioral conformance on architecture level.**

In order to address this issue, we have developed an approach for comparing the implementation of a SoS to an architecture model of its behavioral design. The approach follows the idea of reflexion models and adopts it to support the compliance checking of behaviors. In this paper, we focus on sequencing properties as they play an important role in many SoS. Sequencing deviations potentially have a severe impact on the SoS' correctness and qualities. The desired behavioral specification is defined in UML sequence diagram notation and behaviors are extracted from the SoS implementation. The behaviors are then mapped to the model of the desired behavior and the two are compared. Finally, a reflexion model is constructed that shows the deviations between behavioral design and implementation. This paper discusses the approach and shows how it can be applied to investigate reliability issues in SoS.

Software engineering; software reliability; behavioral analysis; program comprehension; behavior verification

I. Introduction

SOFTWARE architecture has become an essential part of the development and maintenance process of today's systems. The increasing size and complexity must be managed with appropriate abstractions that help focus on important system characteristics and hide implementation details. Views of software architecture not only facilitate comprehension of systems in general but have concrete benefits as they allow conducting analyses of software qualities (e.g., reliability, security, etc.), support the assessment of change impact, and guide the developer in implementing changes [14]. However, architecture documentation can only serve its purpose if it accurately reflects the state of the system. Unfortunately, architecture documentation and the implementation (i.e. the source code) coexist independently from each other. This leads to situations in which the implementation evolves, while the architecture documentation remains unaltered. The result is outdated and irrelevant architecture documentation [10]. An implementation that does not adhere to its design may not fulfill the software qualities it was designed to meet.

Architecture compliance checking has been proposed as a means to remedy that issue. The first and most influential compliance checking approach was proposed by Murphy et al. [12]. In the so-called reflexion model approach, a system implementation is compared to a high-level model of the system's structural design. The result is a new model, the reflexion model, which illustrates how the implementation deviates from the high-level model. The system architect can then re-establish conformance of the implementation to the design. The reflexion model approach has been implemented in several tools [9] and has been applied with great success to a variety of systems. A major drawback current architecture compliance checking approaches suffer is that they are limited to evaluating structural properties. At the same time, behavioral properties are crucial for a system to achieve its reliability, security, performance, etc.

This is most critical in systems-of-systems. Systems-of-Systems (SoS) are distributed systems that act to a large extent autonomously but collaborate with each other to fulfill a common task [11]. Problems in the interaction between systems can reduce the performance and reliability of the entire SoS. At the same time, few methods exist to analyze interaction behaviors and their adherence to the design specification.

This paper presents a reflexion model approach for checking the compliance of an implementation's behaviors to a model of the desired behavior. In that approach, the SoS is executed and the behaviors are monitored. The user specifies a high-level model in sequence diagram notation that expresses constraints on the sequencing of messages. The behaviors are then compared to the desired behavior and deviations are identified. Finally, reflexion models are produced that illustrate the deviations graphically. While we will focus on the evaluation of behaviors in SoS, the approach can also be applied to system-internal interaction behaviors.

The remainder of this paper is organized as follows. Section 2 introduces the basic terminology and provides background about compliance checking with reflexion models. Section 3 then discusses our approach in detail. Section 4 presents a case in which we applied our approach to a real-world SoS. Related work is addressed in section 5 before concluding the paper with section 6.

II. Background

A. Reflexion Models

The reflexion model approach was motivated by the fact that models that were used by system architects to reason about the system were not consistent with the implementation. The models are simple box and line drawings, where boxes represent components and lines dependencies between them. Each component represents one or more source code units (e.g., class, method, etc.). The dependencies indicate how the components are related. The goal of the reflexion model technique is to compare the source code dependencies to the dependencies in the high-level model and illustrate deviations between the two. Before the comparison can be conducted, source code units must be manually mapped to the component in the high-level model that represents them. The dependencies are then compared automatically and deviations are detected. A deviation is either a divergence or an absence. Absences illustrate dependencies that are present in the high-level model but are missing in the implementation. Conversely, divergences are dependencies that are in the implementation but do not exist in the high-level model.

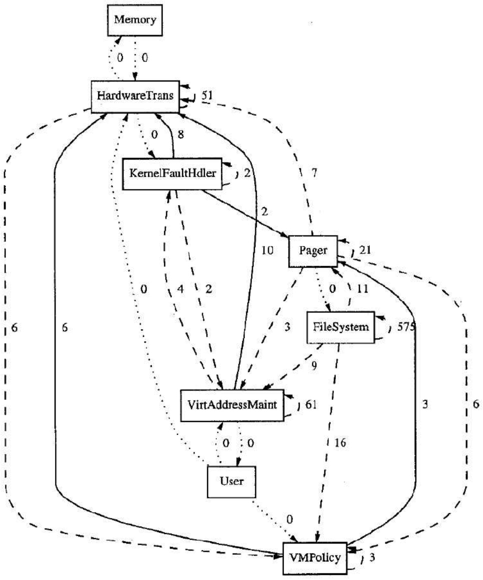


Figure 1: Structural reflexion model as illustrated by Murphy et al. [12].

In order to employ the reflexion model approach for architecture compliance checking, the architecture is specified as a high-level model and the implementation is checked against it. The graphical representation of the deviations in the reflexion model allows the system architect to reason about deviations in the context of the entire structure.

B. Behavioral Reflexion Models

We propose to adopt the reflexion model approach for the context of behavioral analysis. More specifically, we aim to check the adherence of an implementations' interaction behavior to a model that describes the desired behavior. Systems in a SoS interact by exchanging messages, which may contain data or control signals. Data messages are containers to transport data from one physical location to another. Via control messages, the sending system seeks to trigger a certain action in the receiving system.

In order for the systems to collaborate as desired, sequencing rules are specified that describe how the systems should interact with each other. Such sequencing rules can be specified in notations such as Finite State Machines, Propositional Logic, and sequence diagrams. The nature of a sequencing specification is that it describes not a single behavior but the set (or a subset) of all valid behaviors. A behavior is valid if it adheres to the specified rules. For the purpose of this work, we will focus on sequencing rules that are specified in the UML sequence diagram notation as illustrated in Figure 2. A sequence diagram illustrates the components that interact as boxes and messages that are exchanged between these components as arrows. More advanced sequencing patterns are used to describe variability in the behavior. Loops indicate the repetition of a sequencing pattern. Alternative constructs express that a behavior may exhibit different sequencing patterns. Sequence diagrams are hierarchical as a loop and alternative construct may contain other constructs.

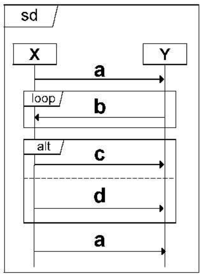


Figure 2: Example sequence diagram.

In our reflexion model approach, we are aiming to compare behaviors that are exhibited by a SoS implementation to a high-level model that describes the valid sequences in sequence diagram notation. The result of the comparison is a reflexion model that illustrates the deviations between the behavior and the high-level model graphically. The purpose of a reflexion model is not to merely indicate whether a behavior adheres to a given specification, but rather to provide information about the degree to which the two match and how they deviate. The graphical representation of the deviations in the context of the entire behavior should facilitate comprehension and support the user in resolving the issues.

III. Approach

Our approach follows the basic idea and the steps of the original reflexion model approach (see Figure 3). A high-level model is created by the user and the information about the implementation is collected. The implementation information is then mapped to the high-level model. Subsequently, the deviations between implementation and high-level model are computed. Finally, reflexion models are constructed that illustrate their deviations.

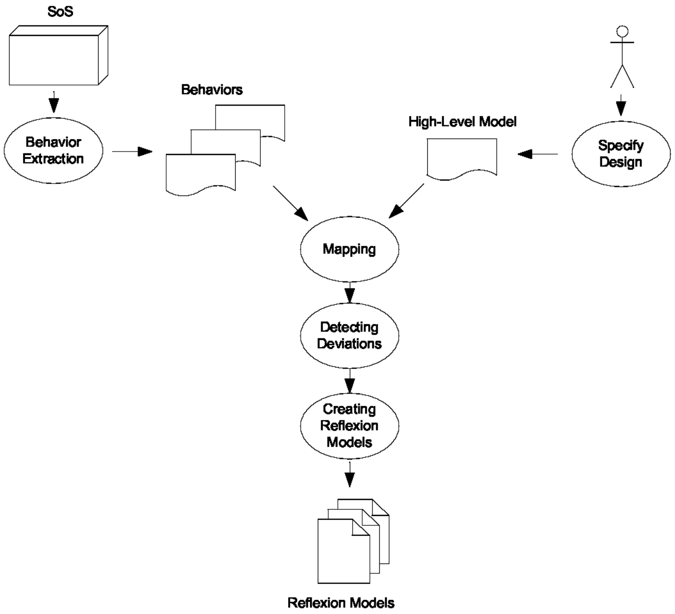


Figure 3: Process overview for computing behavioral reflexion models.

The high-level model is specified in sequence diagram notation and captures the ordering of messages. The implementation information is a set of behaviors that are observed when executing the SoS. The behaviors are individually mapped to the high-level model. More specifically, messages in the behaviors are mapped to messages in the high-level model. Deviations that indicate where and how the ordering of messages is not consistent in the behavior and the high-level model are collected. The collected deviations are then used to construct reflexion models. We distinguish three types of reflexion models that illuminate deviations of all behaviors combined and individual behaviors. Furthermore, we describe a process for employing the different reflexion models for analyses of SoS interaction behaviors.

A. Recording Behavior

The behavior is recorded by executing the SoS and monitoring its interactions. We have presented an approach in which behaviors can be retrieved by observing the physical communication channels between systems [2]. So-called observation points are placed at the endpoints of each communication channel (i.e., at each system) in order to monitor the sending and receiving of messages traveling on these channels.

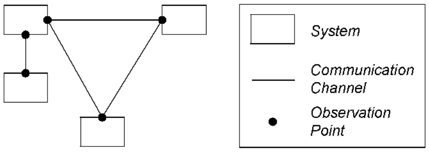


Figure 4: Conceptual view of a SoS.

Figure 4 shows the conceptual setup of a SoS architecture. The boxes represent systems, which are connected by lines indicating communication channels. The dots are the observation points through which the physical interactions can be monitored. Then, the actual behavior model (in terms of message sequences) is constructed based on the observations.

B. Mapping

The mapping establishes a connection between the information extracted from the implementation and the high-level model. In the structural reflexion model approach, source code units are mapped to high-level components. In our approach, we establish the connection by mapping messages of the observed behavior to messages in the high-level model. Hereafter we will refer to the former as *concrete message* and the latter as *high-level message*. The high-level model describes constraints on the ordering of messages in the behaviors. The mapping is used to identify what message in high-level model represents a concrete message. For instance, Figure 5 shows a high-level model (left) and a behavior (right). The first message in the behavior is of type *a*. The high-level model also has a message of type *a* in the beginning of the sequence. Thus, the first messages of the behavior 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 message that appears in a loop may have more than one concrete messages mapped to it (e.g., message *b*). A high-level message that appears in an alternative construct does not have a concrete message mapped to it for every behavior (e.g. message *d*). Also, a high-level model may specify that a message of a certain type may appear more than once (e.g. message *a*).

In the structural approach, the mapping is a manual activity in which the user selects the source code units and maps them to a high-level component. When mapping behaviors to high-level model, a manual approach is not practical as the behaviors often consist of a vast number of messages. We have, therefore, developed a method to automatically map concrete messages to high-level messages.

First, we must establish a set of mapping rules. A concrete message is always mapped to a high-level message of the same type. A high-level message may only have a single concrete message mapped to it unless it appears in a loop. In that case, multiple messages may be mapped to the high-level message (as many as there are loop iterations). A concrete message may only be mapped to a single high-level message.

A simple mapping by message type is not possible as the high-level model may specify more than one high-level messages of the same type (illustrated by message *a* in Figure 5). The context in which the message appears is taken into account as an additional mapping criterion. The context refers to where in the sequence the message appears. We can now refine the mapping rules by stating that a concrete message may only be mapped to a high-level message of the same type, which appears in the same context.

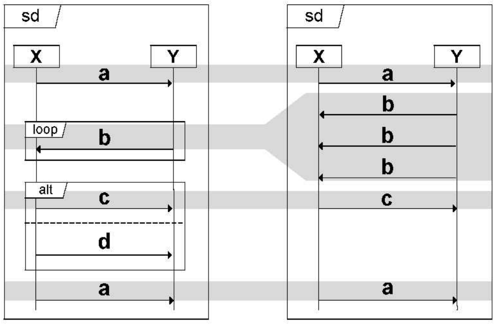


Figure 5: Concrete messages in the diagram on the right are mapped to high-level messages in the diagram on the left.

A possible mapping strategy could be to "execute" the high-level sequence diagram model with the observed behavior similar to executing a Finite State Machine (FSM). The concrete messages are consumed while traversing through the sequence diagram and the mapping is established. Such a mapping is, however, only possible if the observed behavior is "accepted" by the sequence diagram. That is, it is applicable only to sequences of concrete messages that adhere to the sequencing rules described in the high-level model. Such an assumption is, of course, not desirable as the purpose of our approach is to identify deviations in behaviors that violate the ordering specified high-level model.

1) Approximate Mapping

In order to solve the mapping issue, we turn to the area of biosequence analysis in which approximate matching methods are employed to identify DNA strands that are similar to each other [7]. DNA strands are represented as sequences of characters (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.

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

Figure 6: Example for approximate matching of two strings.

Characters in s_0 that match characters in s_1 are aligned with each other. If characters 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 characters. Assuming that matching two characters has a cost of 0, inserting a gap has a cost of 1, and mismatching characters has a cost of 2, the total cost for the example alignment amounts to 5.

An extension to the traditional approximate matching idea has been presented by Myers et al. [13]. Instead of matching two sequences, they propose an approach to match a sequence of characters to a regular expression. The algorithm follows three basic steps: model construction, cost computation, and solution discovery.

Construction. The regular expression is first converted to a Finite State Machine (FSM), which is then replicated one time more than there are characters in the string. Then, the replica FSMs are connected via transitions as specified in the construction algorithm. Figure 9 illustrates the result of that construction procedure. After the model has been constructed, it accepts strings that are not covered by the regular expression, allowing for deviations.

Cost Computation. Starting at the source (the start state), the cost for each state is computed by adding the cost of the preceding state and the cost of the transition. If there are multiple states transitioning into a state, the preceding state is chosen that results in the lowest cost for the current state. After this procedure, the cost of the sink state (i.e., the last state) represents the cost of the best alignment between string and regular expression.

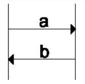
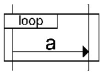
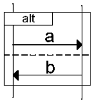
Solution Discovery. In order to identify an actual optimal alignment, the FSM model must be traced back from the sink to the source state. A trace-back procedure is not described by Myers et al. [13] but we will describe such a procedure in the next sections.

We have modified the approximate matching algorithm in order to conduct the mapping of concrete messages to high-level messages. More specifically, the high-level model is converted into a regular expression where the symbols are high-level messages and the input string is a behavior (i.e., a sequence of concrete messages). The algorithm is modified so it not only aligns each concrete message with a high-level message but maps them to each other. The following describes how the sequence diagram is converted to a regular expression and how the original algorithm is modified to conduct the mapping.

2) Model Translation

The sequence diagrams we consider for the scope of this work are limited to sequencing, loop, and alternative constructs. Sequencing expresses the succession of messages. Loops specify a repetition of a sequence pattern. Alternatives indicate that there are two alternate sequence patterns that may occur. In order to produce regular expressions from sequence diagrams, sequence diagram constructs are converted into regular expression constructs. Regular expressions put operators to our disposal that express the same semantics as the sequence diagram constructs. In a regular expression, the succession of symbols is expressed via the concatenation operator (*a.b*), loops via Kleene star (*a**), and alternatives via unions (*a+b*). Table 1 illustrates the mapping between the two notations.

Table 1: Mapping of sequence diagram (SD) constructs to elements in regular expression (RE) notation.

	SD	RE
Concatenation		a.b
Kleene star		a*
Union		a+b

Converting a sequence diagram to a regular expression is now a matter of parsing the sequence diagram and converting each sequence diagram element into the respective regular expression element. Instead of symbols, the regular expression will contain high-level messages. For instance, the characters *a* and *b* in Table 1 represent high-level messages of that type.

3) Modifications

In order to adopt the approximate matching algorithm by Myers et al. [13], several modifications must be applied to the construction procedure.

First, instead of comparing characters, the algorithm must be adjusted to compare and match concrete messages to high-level messages. We have discussed earlier that a concrete message can only be mapped to a high-level message of the same type. Consequently, the matching criterion is the type of the messages. A match occurs if the concrete message and the high-level message are of the same type.

Furthermore, the possibility in the algorithm to align unequal symbols must be removed. In traditional applications of approximate matching, it is useful to mismatch symbols. However, a mismatch in our case would violate the mapping criterion in which we stated that the concrete message and high-level message must be of the same type. We adjust the algorithm by removing (or not inserting) substation edges in which two unequal symbols are aligned with each other. Without these substitution edges, no alignment is produced that requires a mismatch.

Also, the operation of matching must be extended to a mapping operation. That is, when a concrete message is matched with a high-level message, a pointer is set from the concrete message to its high-level counterpart and the high-level message is updated with a reference to the concrete message. The references between the high-level and concrete messages are useful when using the reflexion model for analysis as will be explained later.

In addition, a cost function must be defined as Myers et al. [13] did not propose one. We defined the cost function that assigns a cost of 1 to gaps in either the high-level model or the behavior and 0 to matching alignments.

Finally, Myers et al. do not suggest a method for identifying the optimal alignment after the model has been constructed. Thus, we extended the algorithm to identify the mapping that leads to an optimal alignment of high-level and concrete messages. We adopted the approach that is used in many dynamic programming approaches in which pointers are set to the table cell from which the cost for the current cell was computed. To each state, we add a pointer and set the pointers when constructing the FSM model. In order to determine the optimal alignment, one starts at the sink state and follows the pointers of each state to the source state.

C. Sequence Checking

The goal of sequence checking is not only to provide a yes/not-sure answer regarding the adherence of the message sequence to the high-level model, but rather to produce a model that graphically illustrates the deviation: a reflexion model. A behavior may deviate from the sequencing specification of the behavioral design by *omissions* and *additions*. An omission occurs if the high-level model specifies a message that is not present in the behavior. Additions are messages that are part of the behavior but are not specified as part of the behavioral design.

Such deviations occur if the behavior does not adhere to the sequencing rules. The sequencing rules are used to describe all valid behaviors. Deviations must be described in respect to a valid solution. That is, first the solution to which to compare the invalid behavior must be identified. Then, the deviations can be determined. One could compare the behavior to any valid solution. However, depending on the similarity of the valid solution to the behavior, the number of deviations varies. A common approach is to find the solution that is closest to the invalid behavior. Closeness can be determined by the number of editing operations that are necessary to derive one behavior from the other.

During the trace-back of the mapping procedure, the best alignment between high-level model and behavior is identified. This also implicitly detects the closest valid behavior and even the deviations between the observed and the valid behavior. In the traditional approximate matching algorithm, deviations are expressed in terms of gaps that are inserted in either the high-level model or the behavior. A gap is inserted in the high-level model if a message occurs in the behavior that cannot be matched with a high-level message. This is what we have defined to be an addition. Likewise, if a gap is inserted into the behavior, an omission is detected. In order to identify omissions and additions, the approximate matching algorithm is extended once more. Instead of inserting a gap into the alignment, a deviation record is produced that captures information about the deviation:

<behvr, pos_design, pos_behvr, msg, dev_type>

A deviation record contains a reference to the behavior that contains the deviating message (*behvr*). It stores the position in the design in which the deviation occurred (*pos_design*) and the position in the behavior (*pos_behvr*). Furthermore, it contains the message that caused the deviation (*msg*) and the type of the deviation (*dev_type*). Depending on the type of deviation, the message is either a concrete or a high-level message. If the deviation is an addition, a reference to the concrete message that could not be mapped to a high-level message is recorded. Conversely, if the deviation captures an addition, the high-level message is referenced.

The following shows an example for a FSM model that is constructed based on a high-level model (Figure 7) and a behavior (Figure 8).

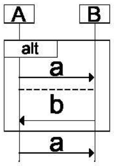


Figure 7: Example high-level model.

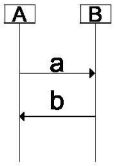


Figure 8: Example behavior.

Figure 9 illustrates the FSM model that is constructed based on the high-level model and the behavior above. The high-level model is first converted into a regular expression. The resulting expression is *(a+b)a*. Then, the regular expression is converted into a single FSM, which is subsequently replicated two times and transitions are inserted to connect the different levels of the FSM model (indicated by dashed lines).

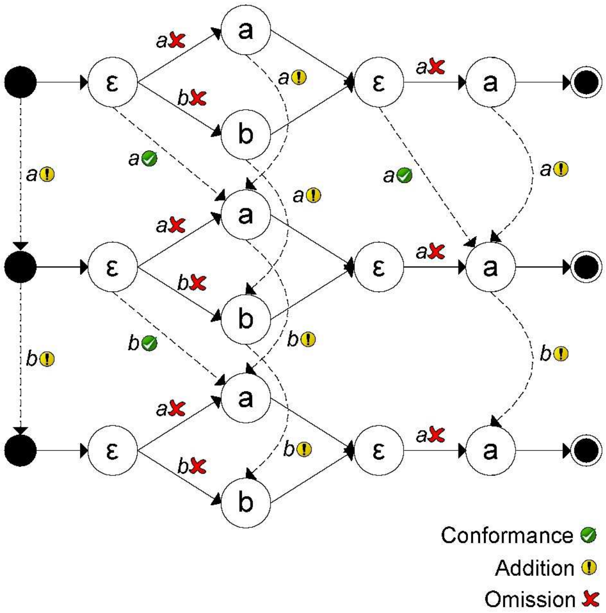


Figure 9: Example for matching the string *ab* to the regular expression *(a+b)a*.

The model in Figure 9 shows the transitions annotated with deviation types. When tracing back from the sink state to the source state, one can identify the alignment and produce the deviation records based on the deviation types that are encountered.

D. Reflexion Models

The reflexion model is a graphical representation of the deviations between the implementation and the high-level model. In the structural reflexion model approach, inconsistencies in the dependencies of the source code and the high-level model are highlighted in the high-level model (see Figure 1). While in the structural reflexion model approach, the implementation is represented by the source code, in our approach the implementation is represented by a set of observed behaviors. The following will define reflexion models to analyze individual behaviors and to explore the deviations of all behaviors combined. We will refer to the former as behavior reflexion model and the latter as high-level reflexion model. We will illustrate the different reflexion models on an example. Figure 10 shows a high-level model and Figure 11 illustrates three behaviors that were observed from executing the SoS. The following example reflexion models highlight deviations between the behaviors and the high-level model.

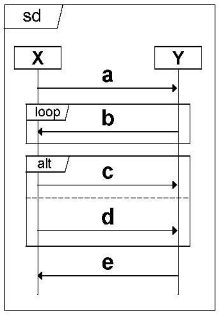


Figure 10: Example high-level model.

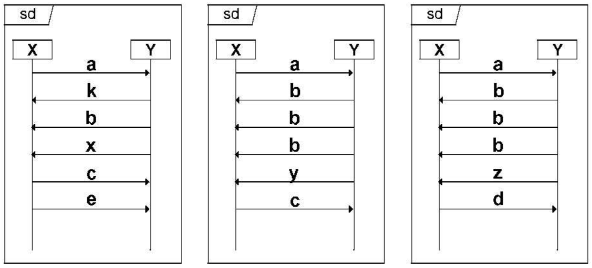


Figure 11: Example behaviors.

1) Behavior Reflexion Model

Behavior reflexion models view the conformance of a single behavior to the high-level model. To construct the reflexion model of a single behavior, the sequence diagram can be plotted to illustrate the original sequence of messages. Subsequently, the view is updated to highlight the deviations to the high-level model. To illustrate additions, the respective message is annotated with an exclamation mark and colored in red. Messages are added to the diagram to visualize omissions in the behavior. An omission is annotated with a red cross and also drawn in red color. If multiple messages were omitted in the behavior, they are drawn in the sequence in the same order as they appear in the high-level model. Figure 12 shows examples for a behavior reflexion model.

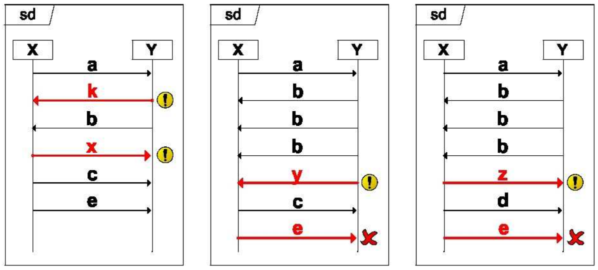


Figure 12: Basic reflexion model of each behavior.

2) *High-Level View of Behavior Reflexion Model*

The drawback of illustrating the deviations in the original sequence of messages is that it does not show the structure of the high-level model. Thus, an additional behavior reflexion model is constructed that illustrates the deviation of a single behavior from the perspective of the high-level model. The model is constructed in the same way as the high-level reflexion model, which is described next. Figure 13 illustrates examples for a high-level view of a behavior reflexion model.

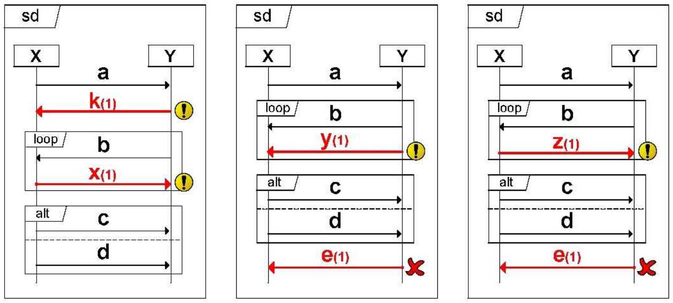


Figure 13: Reflexion model showing the behaviors from a high-level perspective.

3) *High-Level Reflexion Model*

The purpose of the high-level reflexion model is to view all combined behaviors from the perspective of the high-level model. The sequence diagram of the high-level model is first drawn without deviations and then updated to show the omissions and additions. Omissions are again rendered in red color and annotated with an exclamation mark. Additions are drawn below the high-level message that is specified in the deviation record. If multiple deviations were recorded for a high-level message, they are merged into a single message arrow, which is labeled with the types of all messages that are part of a deviation. Also, each message is annotated with the number of deviations that were merged. An example high-level reflexion model is depicted in Figure 14.

4) *Process*

The behavior reflexion model (Figure 12) illustrates the deviations of a single behavior, where the behavior is rendered as a simple sequence of messages. The behavior reflexion models shown in Figure 13 also highlight the deviations of a single behavior but illustrate them in the context of the high-level model. Finally, Figure 14 illustrates the high-level reflexion model in which the deviations of all recorded behaviors are combined.

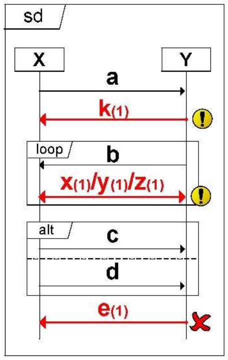


Figure 14: High-level reflexion model showing the deviations of all behaviors.

When analyzing the deviations between high-level model and implementation, one can use the high-level model for a qualitative assessment as to how well the implementation adheres to the high-level model. In order to investigate specific deviations, the user can make use of a reflexion model that shows only the deviations of a single behavior. The high-level perspective represents the deviations in a concise manner but does not reveal all details. For instance, it does not show the sequence of merged messages, etc. To derive even more details, the user can make use of the basic behavior reflexion model, which illustrates the behavior as a sequence of messages but without information about sequencing patterns (i.e. loops and alternatives) contained in the high-level model. By applying a combination of the different types of reflexion models, one can quickly derive a general understanding of the quality of the match and then reveal more details.

IV. Case Study

The following describes two case studies that illustrate the application of behavioral reflexion models for real-world SoS'. In particular, the case studies aim to answer the following questions:

- How can reflexion models be used for analyzing behaviors?
- What issues can be analyzed using reflexion models?

We have discussed the different reflexion models and a process for analyzing SoS' using these models. The case study will show concrete applications of that process. Also, the described analyses were conducted to resolve concrete issues. We will discuss the analysis results to provide insight into these issues.

A. *MOC – Client Interaction*

The Johns Hopkins University Applied Physics Laboratory (JHU/APL) Space Department develops Mission Operations Center (MOC) system software using a shared architecture called Common Ground for JHU/APL-supported NASA missions. APL's NASA missions use the CGS for spacecraft Integration and Test (I&T) and operations. The software is currently supporting I&T and operations for three deep space missions: MESSENGER (discovery.nasa.gov), STEREO (stprobes.gsfc.nasa.gov), and New Horizons (www.nasa.gov/mission_pages). The Mission Operation Center (MOC) acts as control center for satellites in orbit and as permanent storage for telemetry data that is captured in space and transmitted to ground. The MOC is also the hub for client systems that access the stored telemetry data. While the MOC is developed at JHU/APL, the client systems are developed and maintained by other organizations.

1) High-Level Model

The protocol that was used by the MOC and the clients to communicate was documented in the SoS specification.

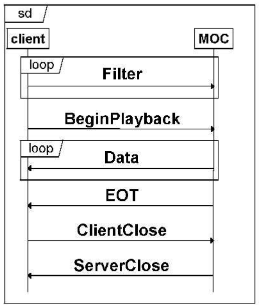


Figure 15: High-level model specifying the design of the MOC-client interaction.

2) Compliance Checking

The system architects monitored the systems via TCP ports and recorded the physical communication using UNIX Snoop at the MOC. The captured information was processed to construct 9 behavioral traces. A FSM was constructed to map each behavior to the high-level model and the deviations for all behaviors were collected.

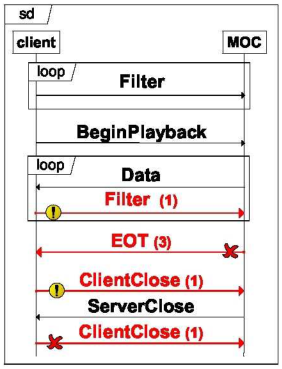


Figure 16: High-level reflexion model of MOC-client interaction.

3) Results

The high-level reflexion model that resulted from comparing all 9 behaviors to the high-level model is illustrated in Figure 16. The reflexion model shows that the behaviors deviate in two ways from the high-level model. First, a *Filter* message is exchanged after a *Data* message has already been sent. Second, the *EOT* message did not occur in all traces. The high-level reflexion model allows for a qualitative analysis of how well the behaviors match the high-level model. The diagram shows that the behaviors contain many sequencing violations. None of these deviations occur during the exchange of *Filter* and *BeginPlayback* message but towards the end of the transaction.

In order to further investigate the nature of these deviations, the behaviors can be illustrated individually through a behavior reflexion model. We want to first analyze the addition of a *Filter* message. The question that arises is whether the deviation is an isolated incident or occurs in the context of other deviations. This cannot be answered with the high-level reflexion model as it shows the deviations of all behaviors in a single diagram. Since every deviation contains a reference to the message that caused the deviation, we can determine that the unexpected *Filter* message was part of trace 2.

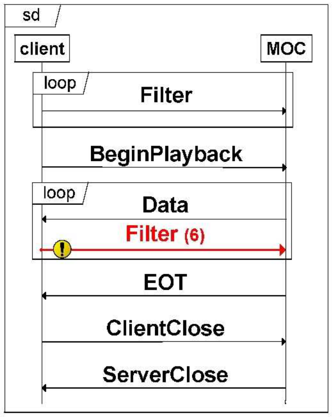


Figure 17: High-level perspective of behavior reflexion model for trace 2.

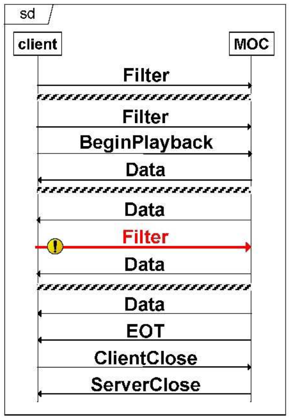


Figure 18: Behavior perspective of behavior reflexion model for trace 2.

Figure 17 shows a high-level view of the behaviors captured in trace 2 along with the deviations. The diagram illustrates that the only deviation in trace 2 is an unexpected *Filter* message following a *Data* message. The diagram does not reveal, however, where exactly the message occurs. Since it is enclosed in a loop, it could occur in between *Data* messages or in between the last *Data* and the *EOT* message. The behavior perspective of the reflexion model depicts these details. The model is illustrated in Figure 19. Due to space reasons, repetitions of *Filter* and *Data* messages were omitted as indicated by the slash marks. The model illustrates that the *Filter* message occurred in between *Data* messages. We can, therefore, conclude that that deviation was not caused by a previous problem but is due to an error in the client system.

The *EOT* message in the high-level reflexion model is marked with omissions. To investigate the omissions further, we can analyze the behavior reflexion models of the traces in which the deviations occurred. One of the three deviations that are represented by the *EOT* message arrow refers to trace 7. Figure 20 shows the high-level view of the behavior reflexion model. It illustrates that the *EOT* omission is not the only deviation in that behavior. Both the addition of the *ClientClose* and the omission of the *ServerClose* message appear in the same trace.

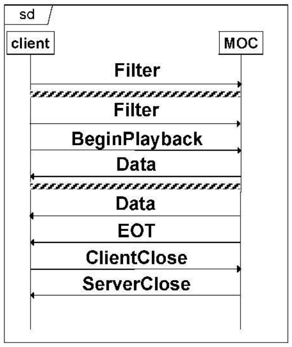


Figure 19: Behavior

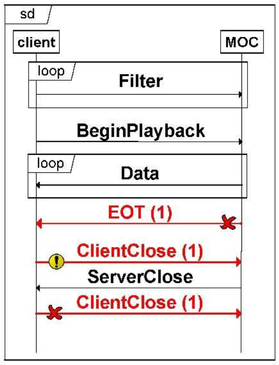


Figure 20: Behavior reflexion model.

After sending *BeginPlayback* and receiving Data messages, the client sends a *ClientClose* message, not waiting for the EOT message. The MOC reacts to this misbehavior by sending a *ServerClose* message.

B. *Satellite-MOC Interaction*

The satellite that continuously captures telemetry data, transfers the captured information to the MOC every time it is within reach. The data is transferred using the file transfer protocol CFDP [5]. Figure 21 shows the behavioral design of the protocol. The *Metadata* message specifies the size of the file, the filename and so on. The satellite then starts sending messages and an *EOT* when all data has been transmitted. The MOC responds by sending an acknowledgement message for the *EOT*. If some of the data was lost during transmission, the client can query the satellite for re-transmission (NAK) upon which the satellite must respond with at least one *FileData* messages. Once all data has arrived at the MOC, it sends a *Finished* message, which is acknowledged by the satellite.

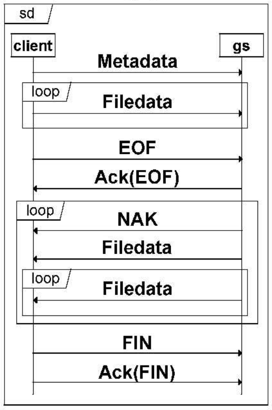


Figure 21: High-level model of CFDP protocol.

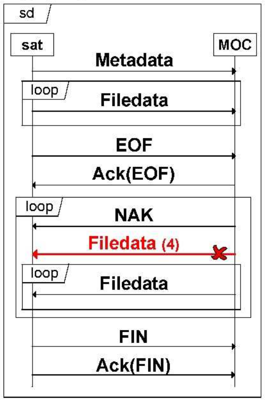


Figure 22: High-level reflexion model.

The interaction was monitored during the normal operation of the SoS. A total of 154 behavior traces were recorded (with a total of 88,309 messages) were recorded. We applied our approach to compare the behaviors to the high-level model and produce a high-level reflexion model as illustrated in Figure 22.

The majority of the behaviors adhered to the high-level model. The only deviation was that in response to a *NAK* message, the satellite would not always respond with a *FileData* message. That means the request by the MOC to re-transmit data that was lost; was not satisfied by the satellite.

A reflexion represents the behaviors of an implementation in respect to a given high-level model. One might not want to specify a complete model of the behavioral design but focus on certain scenarios. The high-level model in Figure 21 shows a complete model for data transmission via CFDP. It also includes a part in which lost data is re-transmitted.

While the re-transmission of data does not violate the protocol specification, it is undesirable as it decreases the performance of the SoS. Also, a re-transmission points out a lack of reliability of the communication medium. Re-transmissions can be analyzed by simply changing the high-level model and comparing the behaviors against it. Figure 23 shows the high-level model that specifies the nominal transmission behavior. The model assumes that all data is transmitted without problems and no re-transmission is necessary.

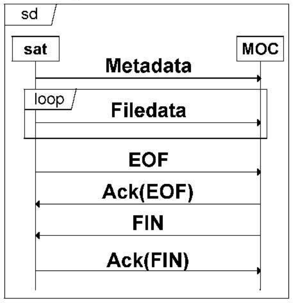


Figure 23: High-level model of the nominal interaction.

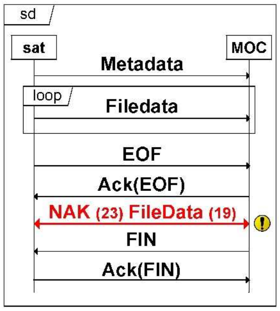


Figure 24: Reflexion model of the adherence to the nominal interaction.

By checking the behaviors against the modified high-level model, we derive a reflexion model that illustrates how often the satellite is forced to re-transmit the data. Based on the reflexion model, one can now analyze each behavior that deviates from the nominal case. The reflexion model shows that the MOC requested the re-transmission of data 23 times using a *NAK* message. This case study shows how the same behaviors can be analyzed from different viewpoints to illuminate different aspects.

V. Related Work

In the areas of reverse engineering and software testing, several works have been published that are of relevance for our research. The following elaborates on them.

A. *Process Conformance*

The study of processes and how they conform to a desired process template is often a discussed issue. Given a process specification, many approaches aim to compute metrics that express the degree to which an observed process adheres to the specification. Aalst et al. [1] have proposed an approach for comparing event traces that were captured in log files to a process specification defined via Petri Nets. The result of the comparison is a metric that expresses the fitness and appropriateness of the trace in respect to the specification. Fitness describes how many events in a set of traces are accepted by the specification. A specification is appropriate if it describes the trace in the least general terms. Computation of metrics that express the degree of conformance (or divergence) between process and its specification is also presented by Cook et al. [6]. The metrics that are computed with that technique are closely related to the computation of costs in approximate matching techniques. They express simple distances between strings (i.e. how many modification operations are necessary to produce one string from the other). While metrics are useful to quantify process conformance, they do not support the user in resolving concrete deviations.

Cook et al. [6] also provide a graphical user interface for analyzing deviations between two event sequences. However, it does not allow for comparing an observed event sequence to a high-level specification, such as sequence diagrams or finite state machines.

B. Testing

Testing approaches aim to identify concrete violations of an implementation to adhere to a specification. Protocol testing techniques are in particular concerned with detecting violations of interaction behaviors to the specification [3]. Testing approaches determine whether the sequence is violated and what that violation was. However, it is necessary to illustrate all deviations in the context of the entire behavior in order to reason about deviations and ultimately resolve them.

C. Reverse Engineering

Although the primary goal of reverse engineering approaches is not to evaluate or verify a system, they share the goal of supporting the user by representing information about systems in a comprehensible manner. Ultimately, the understanding gained through these techniques should help users to solve issues in the system. Jerding et al. [8] have presented an approach for recovering behavior traces in multiple interconnected views. The approach by Briand et al. [4] also visualizes traces but illustrates it in sequence diagram notation including advanced sequencing constructs. Briand et al. also point out the need for an approach to automatically check whether a retrieved behavior is consistent with the documentation.

VI. CONCLUSION

We have presented an approach for comparing the behaviors of a SoS implementation to a high-level model and computing reflexion models that show deviations between them. This work presents a step towards architecture centric analysis for behavioral characteristics. It is based on the reflexion model approach as it has proven to be valuable for analyzing software architectures and extends it to behavioral characteristics since they play a central role in ensuring the correctness and quality of SoS'. The case studies have shown how our approach can be employed to analyze real-world systems and identify behavioral issues.

Our current approach is limited in that it can only evaluate the adherence to a limited set of sequencing constructs. In the future, we will enhance the comparison to also handle other sequencing constructs, such as parallelism, optional, etc. Furthermore, other behavioral characteristics might have to be taken into account when analyzing certain systems. For instance, real-time systems require the timing of messages to adhere to constraints to ensure the system's correctness. We are currently working on extending our approach to additional behavior characteristics. Finally, we will evaluate the applicability of our approach to system internal behaviors.

ACKNOWLEDGMENT

The authors wish to thank Lisa Montgomery and her NASA IV&V SARP program team, as well as Sally Godfrey (NASA GSFC) for supporting this work. We also would like to thank Bill Stratton and Deane Sibol (JHU/APL) for successful collaboration and valuable feedback. Finally, we extend our gratitude to Christoph Schulze for his valuable contribution to this work.

REFERENCES

[1] W. M. P. v. d. Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance Checking of Service Behavior," ACM Trans. Internet Technology, vol. 8, nr. 3, pp. 1-30, 2008.

[2] C. Ackermann, R. Cleaveland, M. Lindvall, "Recovering Views of Inter-System Interaction Behaviors," Working Conference on Reverse Engineering (WCRE), 2009.

[3] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," International symposium on Software testing and analysis, ACM, New York, NY, USA, pp. 109-124, 1994.

[4] L. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," IEEE Transaction of Software Engineering, vol. 32, nr. 9, pp. 642-663, 2006.

[5] Consultative Committee for Space Data Systems, "CCSDS File Delivery Protocol (CFDP) Part 2: Implementers Guide," 2007.

[6] J. Cook, A. Wolf, "Balboa: A framework for event-based process data analysis," ACM Trans. Softw. Eng. Methodol., vol. 7, nr. 3, pp. 215-249, 1998.

[7] D. Gusfield, "Algorithms on Stings, Trees, and Sequences: Computer Science and Computational Biology," NY Cambridge University Press, 1997.

[8] D. Jerding, J. Stasko, T. Ball, "Visualizing Interactions in Program Executions," 19th International Conference on Software Engineering, pp. 360-370, 1997.

[9] J. Knodel, M. Lindvall, D. Muthig, and M. Naab, "Static evaluation of software architectures," 10th European Conference on Software Maintenance and Reengineering (CSMR), pp. 10, 2006.

[10] M. Lindvall, R. Tesoriero, and P. Costa, "Avoiding architectural degeneration: an evaluation process for software architecture", 8th IEEE Symposium on Software Metrics (METRICS'02), pages 77-86, 2002.

[11] M. W. Maier, "Architecting Principles for Systems-of-Systems," International Journal of Software Engineering and Knowledge Engineering, vol. 1, nr. 4, pp. 267-284, 1998.

[12] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," 3rd Symposium on Foundations of Software Engineering", ACM, New York, NY, USA, pp. 18-28, 1995.

[13] E. Myers, and W. Miller, "Approximate matching of regular expressions," Bulletin of Mathematical Biology, vol. 51, nr. 1, pp. 5-37, 1989.

[14] R. N. Taylor, N. Medvidovi, I. E. Dashofy, "Software Architecture: Foundations, Theory, and Practice," John Wiley & Sons, 2009.