# Ensuring Architectural Conformance in Message-Based Systems

Daniel Popescu
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
dpopescu@usc.edu

Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
neno@usc.edu

## Abstract

*Satisfying dependability properties such as fault-tolerance, survivability and security often requires functional solutions that are captured and analyzed in architectural models. Since the abstractions used to describe software architectures are typically different from the constructs used to implement systems, it can be difficult for a software development team to faithfully implement architectural solutions. If the system implementation does not conform to the architecture, then any dependability predictions produced through analysis of architectural models are invalid. This paper proposes an approach for establishing the conformance of an event-based system implementation to its intended architecture. We have applied our technique to several applications implemented on top of two event-driven architecture implementation frameworks. While a number of open research questions remain, the technique has shown good results to date.*

## 1. Introduction

The discipline of software architecture provides the appropriate abstractions, methods, techniques, and technologies for documenting, reasoning about, and communicating the principal design decisions in a software system. Architectural models capture these principal design decisions and serve as the starting point for analyzing non-functional properties such as dependability. Dependability properties such as fault-tolerance, survivability and security are often satisfied through functional solutions that can only be modeled and analyzed via behavioral models. For example, after a service provider fails in a system, a prescribed event sequence might promote a backup or replica to be the new service provider. This dynamism must be captured within an architectural model in order to apply a dependability analysis to it.

In principle, documenting architecturally-relevant decisions allows a software engineer to properly understand and implement a software system design. However, in general a gap exists between the software design documentation and the implementation technology. Software architecture uses constructs like components, connectors and ports, while implementation languages such as Java or C++ use classes, objects and packages. Consequently, differences arise between a systems *prescriptive architecture* (captured in an architectural model) and its *implemented architecture* (captured in the systems implementation).

Architectural decisions therefore exist in both the prescriptive architecture and the implemented architecture. Most organizations keep this information in sync by conducting manual reviews. However, time-to-market pressures often cause engineers to change the system implementation directly, leaving the architectural documentation as an afterthought. In such cases, the prescriptive and implemented architectures drift apart, and critical architectural decisions may even be invalidated over time. In turn, dependability properties that were established by analyzing architectural models may no longer accurately describe the implementation.

To prevent architectural degradation, several different strategies and technologies have been developed. Compliance checking technologies [5] try to map architectural designs and constraints onto the implementation and report differences and violations. Other approaches [2][7] "inject" architectural constructs into the implementation technology (e.g. as implementation language constructs), which can aid architectural reasoning even if architectural changes are made directly in the implementation.

We have combined both approaches to produce a semi-automatic dynamic architectural conformance testing method. Our approach helps a software architect to identify the degree to which a systems implementation is compliant with its prescriptive architecture. In particular, we have targeted event-based systems, which are frequently used to implement applications with high dependability requirements. We assess whether *sequences of events* exchanged among

the modules in a systems implementation correspond to the prescribed event sequences. Our approach is not fully automatic because, in the end, a human engineer must make judgments, if a recorded sequence of events conforms to the prescriptive sequence. While a tool can help visualizing how a trace differs from the prescriptive sequence, a human engineer has to decide wether the violations show that the implementation does not conform to the prescriptive sequence or if the violations are tolerable. For example, some messages that happen in the system might not appear in the prescriptive sequence because the software architect wanted to keep the prescriptive sequence concise focusing only on the critical messages.

To date, we have restricted our studies to systems implemented using two architectural frameworks: Prism-MW [7] and c2.fw [9]. Both frameworks support implementing systems in terms of architectural constructs, including components and messages (i.e., events). This greatly simplifies the task of filtering out architecturally unimportant messages and reduces the sizes of message traces. The process of matching the recorded message traces to prescriptive traces remains hard, because each trace may (1) contain many noise messages, (2) leave expected messages out of a sequence, or (3) change the order of messages.

To deal with all three situations, we have adapted a string-matching algorithm [11] and enhanced it with several trace preprocessing steps. In particular, by focusing only on architectural messages, filtering via causality [6], compressing message loops [11], generalizing messages, and pruning sequences to relevant excerpts, we are able to reduce the observed message trace sizes and enable a possible matching. Noise messages and errors may still exist in the trace because the implementation might have drifted already. Nevertheless, applying our technique can help a software architect to decide the degree to which an implementation is architecturally compliant. Therefore, the technique helps to establish the confidence in the dependability guarantees made based on the analysis of architectural models.

We have applied this technique on several of existing applications. While many open issues remain, our early results have been promising. We report on the technique and our results here.

## 2. Related Work

Abi-Antoun et. al [1] propose an approach for enforcing architectural intent in an implementation in *WADS 2005*. In their paper, Abi-Antoun et al. assert, as we do. that the results of an architectural dependability analysis can only be transfered to the implementation if the system's implementation conforms to its architectural models. Their approach differs from our approach because they focus on detecting *structural* differences instead of *behavioral* differences. As a modeling language, Abi-Antoun et al. rely on a general purpose architectural description language (ADL) and the ArchJava implementation technology. The ArchJava language [2] extends Java with architectural constructs, which simplify the matching of a prescriptive structural model to the implementation. ArchJava uses a type system to ensure that the implementation complies with architectural constraints.

Knodel and Popescu [5] compare three *structural* compliance checking approaches: reflexion models and two rule based systems. Reflexion models are the most prominent static compliance checking technique. As in our approach, a prescriptive architecture description is compared with the corresponding implemented architecture.

Luckham [6] discusses event causality in the context of complex event processing (CEP). In contrast to our approach, event causality rules have to be manually specified for each component or agent. Additionally, CEP uses regular expressions to match event patterns and therefore it is less tolerant to errors in the event trace. Hendrickson et al. [4] developed an approach to visualize and classify events and their causalities in message-based systems. Their approach enables exploring architectural event traces, but does not support matching or filtering.

Hamou-Lhadj and Lethbridge [3] surveyed eight trace exploration tools and techniques. All tools differ in their visualization, data collection and trace compression techniques. All presented approaches suffer from problem of intractable trace sizes, since most approaches record every program statement. Some discussed approaches try to overcome the size explosion problem by using different trace compression techniques such as simple pattern matching, trace sampling and hiding of irrelevant components.

Wendehals and Orso [12] use behavioral matching to detect design patterns. Additional static analysis helps to decide which object methods can possibly participate in a design pattern. Most methods are thereby eliminated and only a few method calls are recorded.

## 3. Approach

The overall goal of our approach is to check the compliance of a systems execution trace to its prescriptive architectural description. Figure 1 shows a conceptual overview of the approach. The approach consists of two major steps: (1) extracting the runtime data and (2) filtering the runtime data and comparing it to a prescriptive sequence of events. Section 3.1 and 3.2 describe step 1 and section 3.3 and 3.4 describe step 2 of the approach.

A stock market simulation case study helps to illustrate the approach. The stock market simulation was developed in Java on top of c2.fw. It consists of slightly over 4000 SLOC and 19 concurrent components. Figure 2 shows the

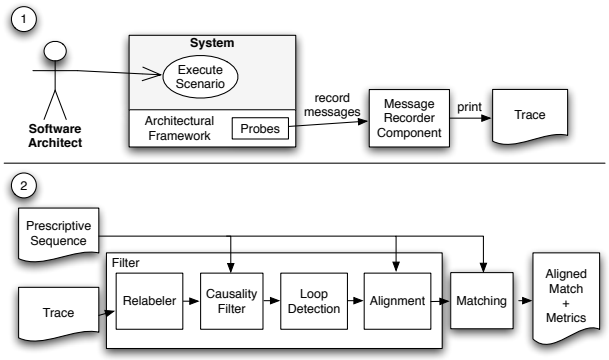


**Figure 1. Conceptual Approach Overview**

message sequence chart that is executed in this case study. The component *TPMonitor* is a transaction processing monitor, which queries the *TPQueue* component for new transaction requests. In this scenario, the queue contains a transaction and therefore the monitor updates the *Repository*. The *TPMonitor* completes the transaction by removing it from the queue, which the queue has to confirm. Executing just this one scenario resulted in over 1000 architectural events.

### 3.1. Implementation Platform Assumptions

In our research, we have focused on architectural compliance in systems implemented using architectural event-based frameworks. In particular, we have worked with systems implemented in the Prism-MW and c2.fw frameworks. Prism-MW is a middleware platform that provides implementation-level constructs for architecture elements (i.e. components, connectors, configurations, styles). Prism-MW implementations exist in Java and in C++. The c2.fw is a Java framework that enforces the C2 architectural style.

In order to record the runtime interaction between architectural components, each implemented component and its

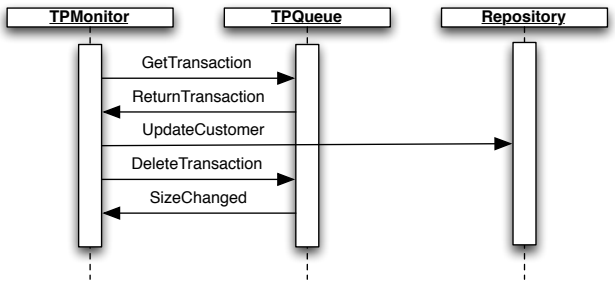


**Figure 2. Case Study - Prescriptive Sequence**

communication ports have to be identifiable in the source code. This is the case with both of the above frameworks. However, note that this is not the case with most plain OO systems: mapping the architectural constructs to the implementation constructs is difficult and potentially error prone in such languages. By using an architectural framework, we avoid the task of having to reverse engineer the components and their configuration.

While both Prism-MW and c2.fw help to identify components and the communication ports of the components, they introduce asynchronous event communication as an additional challenge for the dynamic compliance checking. In both frameworks, multiple threads allow each component to accept, process, and emit events independently. Consequently, each component can participate in multiple sequences at the same time and the control flow becomes complex and difficult to understand.

### 3.2. Trace Extraction

In order to enable a matching between a prescriptive event sequence and a recorded trace, we have to be able to record the event messages that are produced during a system execution. In our work, a message is a tuple consisting of message originator, a message name, and message receiver. Components can originate and receive event messages. A prescriptive sequence consists of messages that are temporally ordered. If a message A appears before a message B, then it is prescribed that A has to happen before B. While UML2 sequence diagrams allow further modifiers such as loops, conditions, and strict environments for the message sequence, our approach to date has used absolutely ordered message sequence charts without additional modifiers. We are planning to explore additional modifiers in the future. A trace consists of messages that were recorded while executing one use case scenario of a software system. In addition, messages in a trace can have causality relationships [6]: components often emit a message because they have received a set of specific messages. In general, causality helps to filter out unnecessary messages and this filtering will be described later (see Section 3.3).

Software probes and a message recorder component record the messages during system execution. Software probes are components that are placed at the communication ports of each component to collect each received event message. Every time a component receives an event message, the probe extracts the necessary message data and sends it to the message recorder component. The message recorder component maintains a list of all received event messages. Using this probe framework, we are able to reuse the trace-recording infrastructure every time we switch to a new component framework. Probes have to be inserted at the communication ports of a component or the component

framework, where they can record messages. Step 1 in Figure 1 shows the conceptual architecture for extracting the event messages.

To extract the causality between messages, our approach uses a simple heuristic: each outgoing event in a component is caused by the most recent incoming event. While this is certainly not always the case, our experience shows that this simple heuristic captures intended causality relationships in a large number of situations. If an architect knows a more complex causality rule for a specific component, he can add this rule into the component probe.

Although recording only the trace of architecture-level messages reduces the trace size, the recorded trace size can still become so large that reasoning about the dynamic behavior is difficult. The trace becomes even more complex because multiple concurrent sequences are often recorded in the trace. Figure 3 shows the relevant trace excerpt that was recorded while executing our case study. Each message sent during the execution is numbered. Note in the figure that the relevant messages in the trace appear after over 900 unrelated messages were sent. Furthermore, the trace is interrupted by 20 unrelated concurrent messages. Leaving all these unrelated messages in the trace would incorrectly indicate low compliance of the implementation to the architecture. Hence, a filtering technique is needed to eliminate irrelevant messages.

### 3.3. Causality Filtering

The extracted trace has to be filtered since it contains many unrelated messages. Filtering by message and/or component name may seem to be a simple choice for removing irrelevant messages. The message or component names from the prescriptive sequence could be used as filter criteria. However, this filtering leads to suboptimal results in many cases. It would remove messages that contain misspellings or new names that were updated by the programmer in the implementation. In addition, such a filter would leave messages in the trace that appear (by name) in the prescriptive sequence, even if those messages participate in a concurrent and unrelated sequence.

Our example in Figure 2 shows that only the *TPMonitor* component sends *UpdateCustomer* messages to the *Repository*. However, other components may concurrently interact with the *Repository* component, hence filtering by message name may leave some unrelated messages in the trace. Filtering by component name could remove the irrelevant *UpdateCustomer* messages, which do not have a prescribed message originator. However, filtering by component name bears the risk that important messages are removed. Figure 3 shows that the implemented *TPMonitor* component interacts with a previous unspecified *UI* component, while processing the transaction. Filtering by component name would remove this message, since the *UI* component is not specified in the prescriptive sequence. Removing this intertwined message might be allowed, but potentially could be also a security risk or architectural violation. Therefore, it is important that the filter does not remove this message. Clearly, component and message names alone are not sufficient to determine whether a message should be left in a given trace.

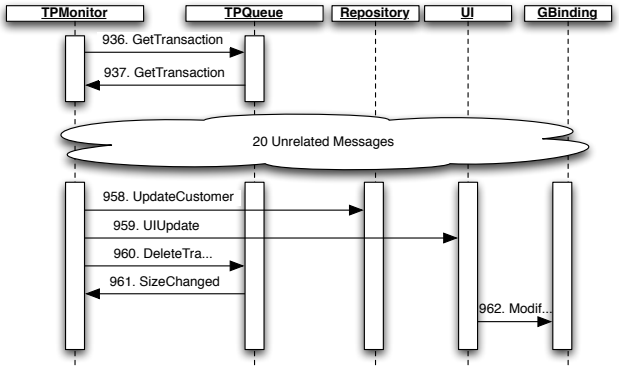


**Figure 3. Case Study - Recorded Trace Excerpt**

*Causality filtering*, used in our approach, can help to untangle intertwined sequences and remove irrelevant messages. The causality filtering uses the heuristically extracted causality relation to identify causally connected sequences and filter out unconnected messages. In the general case, prescriptive sequences describe one or more causally related chains of messages. Figure shows the relevant extracted causality tree of the stock market case study. Each component, represented by an oval, can send and receive messages, and therefore it appears multiple times in the tree. The arrow direction indicates the causality. For example, message *959 UIUpdate* causes the *UI* component to send message *962 Modifcation* to the *Gbinding* (Graphics Binding) component. Message *UIUpdate* itself was not caused by any other message.

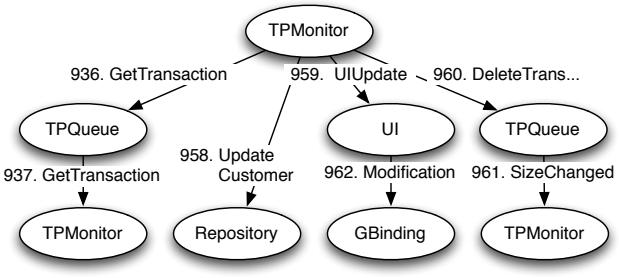


**Figure 4. Case Study - Relevant Causality Tree**

A prescriptive sequence does not capture unrelated messages. It shows a snapshot of what is happening during the execution of a system in one use case. If a message in the prescriptive sequence is not caused by its predecessor, it is in almost every case causing one successor message and therefore each prescriptive message participates most likely in a causality relationship. When all relevant causality relationships are identified, messages that are not part of the relevant relationships can be filtered out. The relevance is defined by trigger messages, which have to be defined for each prescriptive sequence and usually appear only in that prescriptive sequence. In the stock market case study, over 900 messages could be filtered out. In contrast to filtering by component name, the *UI* component is not filtered out when the causality filter is used, since *TPMonitor* sends an event to the *UI* component.

The causality filter also tries to minimize interruptions of causally related events in the trace. In the unfiltered recorded trace (Figure 3), the causality filter can move the *Modification* message before the *DeleteTransaction*, since both messages are causally unrelated and timely close. As long as no message is moved after its own caused message, the causality filter tries to move causing and caused messages closer in the sequence.

Figure 5 shows the final reduced trace after applying causality filtering. In cases where the extraction heuristic does not correctly identify the causality because the causality is more complex, a software architect can use the prescriptive scenario as a guideline to update the causality detection rules in a component's probe.

### 3.4. Matching

In order to be able to perform compliance matching between a prescriptive sequence and a trace, the length of the prescriptive sequence and the relevant excerpt of the trace should be similar. Since traces are usually large, we want to reduce the length of the trace without losing relevant information. As we have already shown, recording only architectural messages and filtering the trace by causality reduces the trace size significantly. Generalizing specific architectural messages, detecting loops, and finding an alignment of the prescriptive sequences to the trace, which we discuss next, ease the matching further.

In some cases, some messages in the prescriptive sequence are more abstract than the implemented messages. Generalization can be utilized in such cases. For example, it might be irrelevant which specific token message a parser is exchanging with its token repository; the prescriptive sequence prescribes only that a token message is exchanged. Relabeling specific architectural message to more general messages helps to have the same abstractions in the prescriptive sequence and the trace. The relabeling rules have

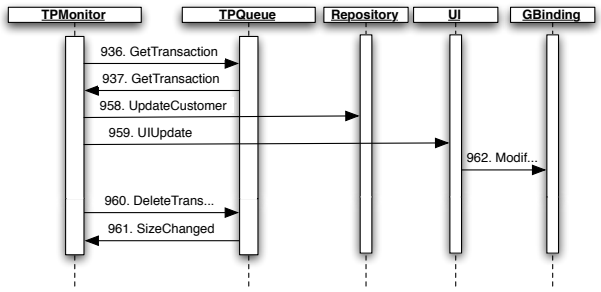


**Figure 5. Case Study - Filtered Trace**

to be specified by a software architect beforehand.

Some traces contain multiple iteration of a message loop. Since the prescriptive sequence does not contain loops or only contains a *Loop* signal keyword, the looped messages cause a suboptimal matching. Our approach uses substring repetition detection to detect concatenated loops. [11]

After performing causality filtering, generalization, and loop detection, traces still can be long and often start with initialization messages that are irrelevant for the current trace. Aligning the prescriptive sequence to the trace by using previously defined trigger messages helps to find a correct subset of the trace. In our experience, each sequence often uses one or more specific message in the beginning of the scenario. A software architect can specify such trigger messages with a cutoff distance. Preceding messages that are farther away then the specified cutoff distance are removed from the trace. This step helped to remove over 900 messages in the case study.

An approximate pattern matching algorithm performs the final matching [11]. Since many implementation-level decisions can affect the trace, an exact string matching is not useful. A software architect is more interested in the degree and the positions where the trace differs from the prescriptive sequence than in a "yes" or "no" answer. Therefore, we use an approximate pattern matching algorithm that allows a limited number of errors in the match. Specifically, our approximate pattern matching uses the *Levenshtein distance* [11] to determine the amount of errors. The Levenshtein distance describes the minimal number of insertions, deletions and substitutions to transform one string into the other. In our approach, it represents the minimal number of operations to transform the reduced trace into the prescriptive sequence. The final output of the matching is the Levenshtein distance, a prescriptive-to-length ratio (*PLR*), and the prescriptive sequence and the trace aligned to each other. The PRL metric describes the percentage of messages in the reduced trace that are prescriptive messages. Both metrics-the Levenshtein distance and the PLRshould help a software architect in assessing the degree of compliance. Figure 5 shows that the final extracted result of the case study is al-

most the same as the prescribed sequence (Figure 2). In the case study, the approximate string matching algorithm detects that the trace contains two additional *UI* messages. It also detects that the prescribed message *ReturnTransaction* is replaced in the trace by a message *GetTransaction* (Message 937).

## 4. Conclusion

The ultimate objective of this work is to help software engineers to ensure the dependability of a system by (1) demonstrating that the system's underlying architecture has desired dependability properties and (2) maintaining the desired correspondence between the architecture and the system implementation. Fault-tolerance, survivability and security solutions can only be achieved, if the architectural designs are faithfully transformed to the implementation. Current implementation technologies such as Java do not make component interactions explicit through the language and therefore continuous manual or semi-automatic conformance testing is importance for dependable systems.

We have outlined a technique that can aid software engineers in establishing the conformance of an event-based system's implementation to its intended architecture. The main contribution of this paper is therefore the behavior conformance technique itself. While pattern detection techniques also facilitate behavioral traces, they either observe only some preselected messages [12] or they suffer from the trace explosion problem [3]. The discussed approach is able to reduce the trace size explosion problem by only focusing on the architectural communication. The trace size can be further reduced by applying causality filtering, generalization, loop detection and alignment. Overall, the approach is able to reduce the trace to messages that are relevant to the prescriptive sequence. Therefore, it enables a determination of the architectural conformance of the implementation. While other techniques exist for determining the conformance of structural architecture descriptions [5], this work extends the scope of error-tolerant conformance testing to architectural behavioral descriptions. While the realization of our approach rests on several assumptions, the principal of which is its reliance on an event-based architectural implementation framework, we believe the proposed technique to be applicable more generally, so long as a mapping between a systems implementation and its architecture is available or can be obtained.

We are currently expanding this work in two ways. First, we are planning to conduct more thorough experimentations on systems that are implemented on top of event-based implementation frameworks. More results should provide a better understanding of the limitations and strengths of the current approach. Second, we are currently expanding the scope of this work by considering different implementation technologies and frameworks. For example, we plan to extend the event-based architectural implementation framework with interaction protocols [10]. We believe that interaction protocols can enable more precise conformance testing. Since many of today's software components interact through explicit invocation such as object method calls, we want to extend our work to general purpose object-oriented languages [2] [8].

## References


[1] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, and T. Tseng. Improving system dependability by enforcing architectural intent. In *Proceedings of the 2005 Workshop on Architecting Dependable Systems*, pages 1–7, 2005.

[2] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24rd International Conference on Software Engineering*, pages 187 – 197, Jan 2002.

[3] A. Hamou-Lhadj and T. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 42–55. IBM Press, 2004.

[4] S. Hendrickson, E. Dashofy, and R. Taylor. An (architecture-centric) approach for tracing, organizing, and understanding events in event-based software architectures. *Proceedings of the 13th International Workshop on Program Comprehension*, pages 227–236, May 2005.

[5] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *The Working IEEE/IFIP Conference on Software Architecture*, Jan 2007.

[6] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison Wesley, Jan 2002.

[7] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256 – 272, Mar 2005.

[8] N. Medvidovic and V. Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

[9] N. Medvidovic, N. Mehta, and M. Mikic-Rakic. A family of software architecture implementation frameworks. In *Proceedings of the IEEE/IFIP Conference on Software Architecture*, pages 221–235. Kluwer, BV Deventer, The Netherlands, The Netherlands, 2002.

[10] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: the case of interaction protocols. *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, Jan 2002.

[11] G. A. Stephen. *String Searching Algorithms*. World Scientific, Jan 1994.

[12] L. Wendehals and A. Orso. Recognizing behavioral patterns at runtime using finite automata. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, May 2006.