Enabling More Precise Dependency Analysis in Event-Based Systems

Daniel Popescu, Joshua Garcia, and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{dpopescu, joshuaga, neno}@usc.edu

Abstract

Maintenance engineers need to understand component dependencies in a system to successfully modify component implementations. Interaction dependencies are especially hard to understand in event-based systems, since transfer of control between components typically happens implicitly and asynchronously. We present a framework for event-based systems that guarantees that unspecified event-based dependencies do not occur. Consequently, the framework enables automated computation of architectural dependencies that are more precise than those possible in other event-based frameworks.

1. Introduction

In a study by Microsoft Research [4], a majority of participating software engineers stated that understanding the impact their code has on other code is a serious problem. Although event-based systems provide benefits such as enabling adaptability and low coupling, they exacerbate these dependency-based understandability issues because they utilize implicit transfer of control [1] [2]. This paper's research contribution is an approach that retains the benefits of adaptability and low coupling found in many event-based middleware platforms, while at the same time facilitating dependency analysis by enabling querying of a system for inter-component (i.e., dependencies between an outgoing event of one component and an incoming event of another) and intra-component dependencies (i.e., dependencies between incoming and outgoing events within a component).

2. Understandability Challenges

Event-based systems introduce understandability challenges because communication dependencies exist only implicitly, making the impact of component modifications on a system unclear. Here we discuss three of these challenges.

Ambiguous Interfaces. Ambiguous Interfaces [1] are interfaces that both offer a single, general entry or exit point to or from a component (e.g., a single "send" or "receive" method) and accept a generic event as a parameter, even though the component processes only some specific events and discards the remaining events. Consequently, a component offering only an Ambiguous Interface seems to handle more events that it does, and a static analysis of all component interfaces over-generalizes system dependencies.

Black-Box Interfaces. Black-box interfaces are interfaces that only contain subscriptions and advertisements, but do not specify any possible dependencies between incoming subscriptions and outgoing advertisements. The intra-component dependencies are hidden within the component's implementation. This decreases understandability, since every incoming event of a component could potentially impact every advertised event of the same component.

Component State. Intra-component dependencies can also be caused by dependencies on a component's internal state. An incoming event that modifies the state of a component could potentially impact all outgoing events because future incoming events may trigger a service that publishes outgoing events based on the modified state.

3. Approach

Our approach aims to enforce a component model in a system's implementation in order to achieve the following three goals: (1) extractability of more precise dependencies from an event-based system by eliminating black-box and ambiguous interfaces, (2) preservation of the completeness of an event-based system's dependency model and (3) retainment of the adaptability benefits of event-based systems.

Static specifications and dynamic compliance checking allow us to achieve these three goals. The static specifications require capturing information that is a subset of that typically provided in architectural descriptions [3]: for each component an engineer specifies subscriptions (i.e., incoming events), advertisements (i.e., outgoing events), and potential dependencies between them; moreover, the engineer specifies a system topology by describing how components and connectors are attached to each other. These specifi-

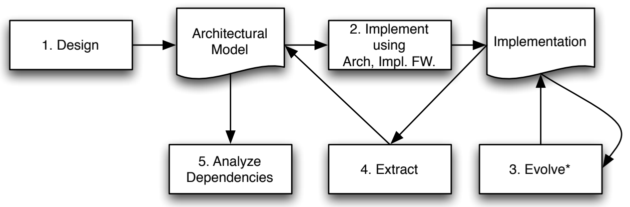

Figure 1. Overview of the approach.

cations result in a statically recoverable event-dependency model that is more precise than the analogous model comprising ambiguous interfaces. The dynamic compliance checking compares each component's runtime behavior to its static specification and reports runtime violations.

Figure 1 illustrates the activities and artifacts of our approach. The design (activity 1) results in an architectural model which is implemented using the architectural implementation framework (activity 2). As the implementation evolves (activity 3), the mapping given by the framework ensures consistency between the model and the implementation. This model can then be extracted (activity 4) so that dependency analyses can be performed on the model.

The structural model of our framework can be described as follows: An event-based system has an architecture which contains components and connectors. Components can have incoming subscription ports and outgoing advertisement ports. Components can only exchange events with connectors through their ports. Finally, a component contains state objects and *dependency rules* (described below).

Addressing Ambiguous Interfaces. The ports in our model help to address the ambiguous interfaces. Every component can only receive or publish events through ports. An event can only pass through a port if the type of the event is registered at the port. Therefore, all component ports can be queried for a component's advertisements and subscriptions. Since the system topology is explicitly defined, ports enable the analysis of inter-component dependencies.

Addressing Black-Box Interfaces. *Dependency rules* allow an engineer to specify intra-component dependencies between subscriptions and advertisements for a component.

Compliance of the implemented system to the specified *dependency rules* is checked in the following manner. After a component receives an event through a subscription port, its component thread marks all *dependency rules*—those that specify the same event types as the type of the incoming event—as active. As the next step, the component starts the service that processes this event. If this triggered service tries to publish an event, that event is only sent if it matches an outgoing event of an active *dependency rule*. This guarantees that no unspecified events are ever published.

Addressing Component State. In the general case, static analysis cannot determine whether a component's state will be modified by an incoming event. Instead of statically checking whether an event modifies the state of the component, we track the occurrences of write operations during the execution of an event-triggered service.

An engineer can specify whether or not a *dependency rule* allows writing to the state of a component. If writing is allowed for a *dependency rule*, we assume that the specified incoming event can potentially impact all advertised events because in a (black-box) component-based system one generally does not know which services are dependent on the component's state. If a component's *dependency rule* disallows any state modification, we know that the component's incoming event can only influence outgoing events specified according to *dependency rules*. A compliance warning is issued when a service modifies a component's state whose active *dependency rule* prohibits any state modifications.

Preserving completeness of the dependency model. The completeness of the dependency model is preserved by covering all the different cases of control-flow and data-flow dependencies. Our *dependency rules* cover all legal control-flow dependencies, since they monitor and filter incoming and outgoing events. To cover all data-dependencies, we make the conservative assumption that each event-triggered service that writes to a component's state influences all advertised events of the same component.

Maintaining the Adaptability Benefits. Our framework retains the same degree of adaptability because components communicate via explicit event-based connectors and maintain no direct reference to each other.

4. Conclusion

This paper introduced an approach that increases understandability and analyzability of event-based systems through the use of static specifications and dynamic compliance checking. As part of our validation, we have adapted an implementation of the arcade game KLAX [3]. After addressing ambiguous interfaces, the original KLAX implementation consisted of 16 black-box components that each had on average 24 potential intra-component event dependencies. After utilizing our approach, the intra-component dependencies could be reduced to 17 in average.

We are currently augmenting our approach to allow more accurate checking of complex component state, which further improves the precision of the extracted dependencies.

References

[1] J. Garcia et al. Identifying architectural bad smells. In *13th Eur. Conf. on Softw. Maintenance and Reengineering*, 2009.

[2] G. Mühl et al. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2006.

[3] R. Taylor et al. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2008.

[4] G. Venolia et al. Software development at Microsoft observed. Technical Report MSR-TR-2005-140, Microsoft Research, 2005.