Architectural Analysis of Systems based on the Publisher-Subscriber Style

Dharmalingam Ganesan, Mikael Lindvall
Fraunhofer CESE, College Park, Maryland
{dganesan, mlindvall}@fc-md.umd.edu

Lamont Ruley, Robert Wiegand, Vuong Ly,
Tina Tsui
NASA Goddard Space Flight Center,
Greenbelt, Maryland
{ lamont.t.ruley, robert.e.wiegand,
Vuong.T.Ly, Tina.Tsui}@nasa.gov

Abstract

Architectural styles impose constraints on both the topology and the interaction behavior of involved parties. In this paper, we propose an approach for analyzing implemented systems based on the publisher-subscriber architectural style. From the style definition, we derive a set of reusable questions and show that some of them can be answered statically whereas others are best answered using dynamic analysis. The paper explains how the results of static analysis can be used to orchestrate dynamic analysis. The proposed method was successfully applied on the NASA's Goddard Mission Services Evolution Center (GMSEC) software product line. The results show that the GMSEC has a) a novel reusable vendor-independent middleware abstraction layer that allows the NASA's missions to configure the middleware of interest without changing the publishers' or subscribers' source code, and b) some high-priority bugs due to behavioral discrepancies, which were eluded during testing and code reviews, among different implementations of the same APIs for different vendors.

Keywords: Architectural Styles, Middleware, Vendors, Static and Dynamic Analysis, Component-Connector Views, Colored Petri Nets.

1 Introduction

Architectural styles are abstract "high-level" concepts offering reusable solutions to recurring design problems. Equivalently, architectural styles define the roles, the topology, and the interaction behavior of involved components [20]. The publisher-subscriber architectural style is one of the most prominent styles, in which different components communicate in an indirect fashion by publishing and subscribing to messages managed by an intermediate communication bus (or broker) [3]. This indirect communication makes the architecture flexible because it facilitates adding and removing components. This style is thus an attractive option, for example, when one wants to develop a family of similar products in a disciplined way. For example, in a previous case study [6], we reported that the NASA's flight software product line was developed in a flexible way by adding or removing publisher/subscribers, based on the needs of missions.

However, this increased flexibility of the publisher-subscriber architectural style is also a curse because it makes it difficult to predict the emerging behavior and to prove the correctness of the integrated system even though each individual component passed testing and was demonstrated to be correct on its own. Thus, while flexibility increased, analyzability decreased. One example of problems that cannot be detected by analyzing individual components is inter-component oriented timing issues. Such issues emerge only when several components are using the bus.

Several researchers have analyzed the publisher-subscriber architectural style at an early stage (i.e., before the implementation) (e.g. [8, 9]). They construct rigorous formal models of the publisher-subscriber style and prove correctness using model-checking techniques. Unfortunately, many existing systems were developed without such a rigorous architectural phase. Also, even if there was such a phase, experience reminds us that the implementation could deviate from the specified architecture (e.g., [16, 7]). Therefore, it is instructive to reverse engineer the implemented architecture and analyze the behavioral properties and constraints of the publisher-subscriber style.

We developed a practical approach for analyzing implementations based on the publisher-subscriber style. Our key activity is to derive a set of reusable questions from the definition of the style. These questions drive the analysis and the answers to them constitute evidence regarding the compliance of the implementation to the style constraints and overall design quality. We will show that some questions are possible to answer statically, whereas others are better answered by monitoring the running system. In the static analysis phase, we bridge the gap between "high-level" concepts such as publish, subscribe, unsubscribe, and communication bus, and the source code concepts. That is, we locate key interfaces, methods, and data structures used for implementing the publisher-subscriber style. Our static analysis is guided by analyzing dependencies to external entities (e.g., Middleware vendors' APIs and programming language libraries) which are stored in our experience repository, based on more than 10 years of analysis of several commercial systems at Fraunhofer.

In order to conduct complementary dynamic analysis, we use the results of the static analysis to a) define and automatically insert probes at the right location for collecting run-time events (e.g., call events), and b) store and/or forward run-time events to various tools that we use to analyze data. Our technical set-up for dynamic analysis takes advantage of the publisher-subscriber style by introducing a run-time event collector component (RECO) into the software architecture. Our probes emit run-time events as messages into the communication bus, which deliver them to the RECO. In a sense, dynamic analysis is seamlessly integrated into the publisher-subscriber style. By monitoring the running system, we interpret the run-time events for discovering a) component-connector (C-C) views of the publisher-subscriber style including components, ports and connections between ports as well as b) sequence diagrams including messages exchanged between different components indirectly using the intermediate bus or directly between components with the bus hidden. We also discuss how we detect violations of behavioral style properties. Our dynamic analysis is facilitated by the construction of Colored Petri Nets (CP-nets) [4]. CP-nets are useful for recognizing pre-planned patterns in interleaved events. We used CP-nets because run-time events of a publisher-subscriber architectural style are highly interleaved. For example, when one component is emitting subscribe events the other component could be in the middle of creating a connection to the communication bus.

The system under study is NASA's GMSEC system, whose team has developed a software product line based on the publisher-subscriber architectural style. The proposed approach was followed for an independent analysis of the GMSEC implementation. We discovered a) a middleware abstraction layer that offers vendors' independent abstract interfaces to interact with the communication bus of different vendors, b) C-C views, c) sequence diagrams, and d) some behavioral violations resulting in high-priority bugs due to inconsistencies between the implementations of the same interface for different vendors.

Contributions of the paper: We have found little discussion on the topic of analyzing structure and behavioral constraints of architectural styles in the reverse engineering community. To this end, we hope the paper contributes the following:

1. A practical approach for analyzing the publisher-subscriber style using a combination of static and dynamic analysis. The technical set-up and analysis questions are also reusable on other systems based on the same style.
2. The reverse engineered GMSEC architecture is also reusable, offering novel insights on how to design and implement middleware abstraction layer, which frees organizations from being vendor-locked.

2. Approach

In this section, we will introduce the "high-level" concepts of the publisher-subscriber style, which will be used to derive a set of questions for the architecture analysis. In order to answer the questions, we will present the approach for discovering a set of views, using both the static and dynamic analysis.

2.1. Concepts of the Publisher-Subscriber Style

In the Publisher-Subscriber style, each participant can play the role of a publisher of message, a subscriber of messages, or both. Messages are key entities in this style. Typically, each message has a subject (a.k.a. topic) as well as a structure containing a number of fields and values that carry the data to be sent [3]. The central artifact is the broker or the bus (a.k.a. software bus). A typical bus has elements for connection management, subscription management, message buffer and routing management, as well as interfaces for publishing and subscribing, as shown in Figure 1. The connection management is used by the publishers/subscribers for connecting to and disconnecting from the bus. Subscribers use the interfaces of the bus to subscribe to messages of interest. Similarly, publishers use the interfaces of the bus to publish messages, which are stored in an internal buffer, often created and managed by the software bus [3]. The subscription management is used by the bus to manage the list of subscribers. One important job of the bus is to route the messages to appropriate subscribers, thus it needs to have message routing management concepts. Note that the bus can also play a role of publisher and/or subscribers by using its own interfaces.

There are several implementations of software buses on the market, typically based on middleware. Systems that make use of a middleware may want to hide/wrap and even generalize the interfaces of the middleware in order to avoid being dependent on a particular middleware from a particular vendor. To provide true flexibility, the programming language used to implement the bus should not dictate the programming language to be used by the subscribers and publishers.

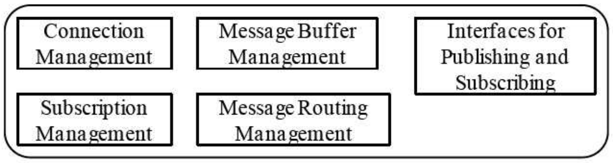
Connection Management
Message Buffer Management
Interfaces for Publishing and Subscribing
Subscription Management
Message Routing Management

Figure 1 Typical elements of a software bus

2.2. Deriving Analysis Questions from the Style

In order to understand and assess the implementation of systems based on the publisher-subscriber style, we now derive a set of typical questions based on the high-level concepts of this style, introduced above.

1. Are there any middleware used for implementing a software bus?
2. If middleware is used, is there any middleware abstraction layer that hides the knowledge of a particular middleware API?
3. Can publishers and subscribers be implemented in different programming languages? If yes, how are the variants in the languages managed?
4. Can publishers and subscribers come-and-go dynamically at run-time?
5. Can publishers, subscribers, and the software bus run on different machines?
6. Which subscribers receive messages from which publishers and in what order?
7. Do subscribers receive messages that are not subscribed by them?
8. Can subscribers subscribe to the same message more than once without an intermediate unsubscribe?
9. Can subscribers unsubscribe to messages that are not subscribed by them?
10. Are there timing delays in delivering messages to subscribers?

The above set of questions is concerned with both the structural and the behavioral aspects of implementations. For instance, questions 1-5 deal with the module-structure of the system, whereas questions 6-10 deal with behaviors. Therefore, our approach has a static and a dynamic analysis phase. In the static analysis phase, we discover key header files, classes, interfaces, and routines related to the software bus concepts. In addition, we create "box-and-lines" views of modules related to the software bus concepts that we have discovered. In the dynamic analysis phase, we use the results from the static analysis in order to a) create probes that monitor the running system, and b) for pattern recognition of run-time events. The data from the probes is used to discover components and connectors and are documented as component-connector views and sequence diagrams. The discovered relationships depicted in component-connector views offer insights related to the run-time structure of the software, which are difficult to obtain using static analysis techniques. The views are used to understand the implemented architecture and to draw conclusions on the systems' implementation quality. Now, we explain how we perform static and dynamic analysis to answer these questions.

2.3. Static Analysis Strategies

Our goal of the static analysis step is to locate "high-level" concepts of the publisher-subscriber style in the source code. We automatically extract dependency models (e.g., include relations, call relations) from the source code and analyze them semi-automatically. Our static analysis strategy is based on observations that systems typically do not implement the publisher-subscriber style from scratch; instead they build on top of external entities such as middleware frameworks offered by commercial or open source vendors.

For the analysis, we use a set of tools and other resources that have been developed during the past 10 years at Fraunhofer. For example, we stored the names of header files, classes, and methods/functions of frameworks and programming language libraries that are architecturally-significant for each architectural style. For example, Table 1 shows a small snippet of the stored data for two middleware vendors, namely the Tibco and the Apache Active MQ. We store this data in a relation database model, and use it in conjunction with dependency models in order to locate the files that are involved in the implementation of the software bus concept. For querying the dependency model, we use the relation partition algebra (RPA) [5]. The search result is used to a) discover the presence of any abstraction layers or wrappers to such vendor libraries, and b) detect potential architecture issues. For example, if the system under study uses vendor libraries directly without any intermediate wrappers then it indicates a potential design issue, because the system is now vendor-locked, impeding switching to another vendor's solution.

Table 1 an excerpt of middleware vendors' APIs.

Vendor	Method Name	Purpose
Tibco Smart Sockets (SS)	SSConnection::SSConnection	Initializes connection to the middleware
Tibco SS	SmartSockets::TipcSrv::send	Publishes the given message
Tibco SS	SmartSockets::TipcSrv::setSubscribe	Subscribes to the given message
Apache Active MQ	CMSConnection::CMSConnection	Initialize connection to the middleware
Apache Active MQ	cms::Session::createProducer	Publishes the given message
Apache Active MQ	cms::MessageConsumer::setMessageListener	Subscribes to the given message

In order to reason about how the system handles programming language variants, we locate dependencies to common header files and trace backwards. For example, JNI [15] and XS [14] are respectively used for communication between Java to C++, and Perl to C++. Using this knowledge, we can locate files that are dealing with more than one programming language. We attempt to understand how the language-to-language translation is separated from other concerns.

We also keep track of typical variable and routine names developers use for implementing different architectural styles of systems. For example, in a previous case study, programmers used "publish", "subscribe", "sendMsg", "rcvMsg", and "unsubscribe" to implement the publisher subscriber style in the C language [6]. We use these sets of keywords to search the code base and/or the extracted dependency models to recognize the presence of potential architectural styles in the source code. To facilitate searching, we developed a robust implementation of the vector space model, based on [18], which allows us to search the source code base and outputs a ranked list of files similar to a given list of such keywords. This search technology also facilitates the static analysis of systems that support different programming languages for implementing the publisher and subscriber style. For example, we can select the Java implementation of the software bus and request for files "similar" to the given set of files, which might implement the software bus for other programming languages.

We stop the static analysis after getting answers to questions 1-5. Since discovery is an iterative process, we often conduct more static analysis as dynamic analysis raises new questions. The output of the static analysis is the discovery of the presence (or absence) of significant components as well as key header filers and/or interfaces corresponding to the "high-level" concepts shown in Figure 1. This output feeds into the dynamic analysis activities.

2.4. Dynamic Analysis Strategies

Our goal of the dynamic analysis is twofold. First, to discover components and connectors in the implementation, and create corresponding component-connector views of the publisher-subscriber style. Second, to create sequence diagrams, showing messages exchanged among the publishers and subscribers, including the support for hiding the intermediate software bus. We need dynamic analysis because a) there is an inherent dynamism in the publisher-subscriber style, meaning that publishers/subscribers can connect and disconnect as each individual component feels necessary. Thus, the actual architectural configuration of the system is often only known at run-time, and b) questions related to ordering of messages and timing are not easy to answer statically, if not impossible. The correctness of systems based on the publisher-subscriber style depends on the correct functioning of the software bus, and also depends on the publishers and subscribers using the software bus in the right way. A misbehaving component could easily prevent it from functioning properly.

In several cases, we observed that the software bus and the publisher/subscribers are developed by different teams, who may even belong to different organizations. Thus the developers never meet in person and build components solely based on interface control documents and similar information. Therefore, it is often not possible to statically check the correctness of the behavior of the resulting system, and verify that the publishers/subscribers and software bus are all "safe". Thus, we need to monitor the running system and be able to check the correctness at run-time so that constructive actions can be taken. For example, notifying the misbehaviors for further investigations by system administrators, triggering requests for removing misbehaving parties, etc.

Now, we will enumerate a few key challenges related to such analysis and how we address them in dynamic analysis of the publisher-subscriber style.

First, in order to minimize both the overhead of injected monitoring code and the amount of run-time data, we need to locate the right spots where the sought for data can be collected. We address this challenge by inserting probes that monitor the usages of the interfaces of the software bus that were discovered using static analysis. Depending on the context factors (e.g., software architecture, programming languages, and organizational boundaries), we choose the most appropriate instrumentation strategy for defining and inserting probes. In [7], we used run-time weaving using Aspect-J for Ricoh's photocopy machine. In the context of the publisher-subscriber style, we can design probes to emit run-time events, as messages with the special subject "trace", into the software bus itself.

Second, we need to collect the emitted run-time events and forward them to analyzers. In our approach, we introduce and integrate a special component called the Run-time Event Collector (RECO) into the publisher-subscriber architecture, which subscribes to all messages with "trace" as the subject.

Third, we need to systematically handle interleaved run-time events of the publishers and subscribers for discovering component-connector views and sequence diagrams. For instance, when one publisher is in the process of creating a connection to the software bus, another subscriber might already have subscribed to another or the same message, and at the same moment, another subscriber might be waiting for other set of messages to come in. Note that publishers and subscribers could run on different machines, processes and threads. As a consequence, the emitted run-time events are highly interleaved and difficult to analyze.

We tackle the challenge of analyzing interleaved events by using Colored Petri Nets (CP-nets), which are shown to be a useful executable formal language for handling asynchronous systems behavior [4]. CP-nets are well-elaborated in [7]. Basically, a CP-net is a graph with two types of nodes called *places* and *transitions*. *Tokens* are entities with data attributes/fields, associated with places. Each *transition* can have one or more input places and one or more output places. Every transition has a precondition which is a Boolean expression associated with input places. A transition fires if there is at least one token in each of the input places that satisfy the precondition. If a transition fires, it removes one input token from each of the input places. Every transition has an action, which is a sequence of assignment statements that assign tokens to output places when transition fires. We developed a prototype implementation of CP-nets that is described in [12].

Based on the publisher-subscriber style and its implementation concepts, which was discovered during static analysis, we constructed CP-nets that process the incoming run-time events and recognize pre-planned patterns, and produce the data necessary for constructing C-C views and sequence diagrams. By pre-planned patterns, we mean the implementation constructs corresponding to architectural constructs. For example, calls to the "publish" method of the Connection class correspond to the abstract publishing concept in the style. The output of CP-nets is used to visualize both C-C views and sequence diagrams that hide the software bus. If we do not hide the software bus, all communication will be between a component and the software bus. We developed Dyn-SAVE to automatically create and visualize sequence diagrams based on output from such CP-nets [13].

It is worth noting that CP-nets run in parallel to the system under study and analyses the run-time events at run-time. Thus, it is a novel formalism for building architectural monitoring and compliance checking at run-time for dynamically reconfigurable systems. In our approach, we also use CP-nets for checking behavioral constraints at run-time. For example, we can check whether a subscriber receives any message other than what was subscribed as follows: One transition of the CP-net can wait for subscribed events and output, to one of its output places, the list of subscribed messages by subscribers. The second transition of the CP-net can wait for events that read messages from the software bus to occur, and output, to one of its output places, the list of messages read by each subscriber. The third transition of the CP-net, designed to consume the output of the above two transitions, can output to its one of the output places, the list of unwanted messages wrongly routed by the software bus. Figure 2 summarizes the conceptual elements of our dynamic analysis environment for analyzing systems based on the publisher-subscriber style. It is worth noting that the RECO component is in fact plugged into the running system – just like other components – probes can also be injected into the RECO component in order to make sure this trace collection component uses the software bus in the right way.

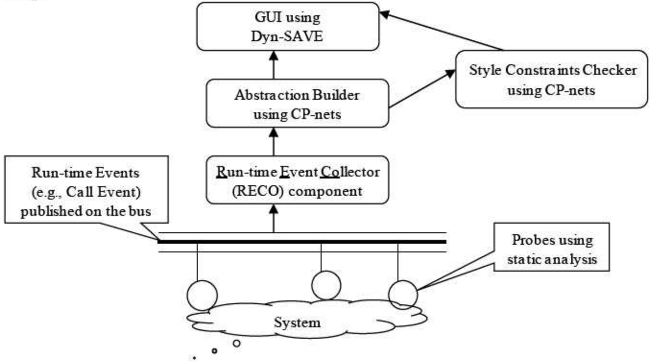

Figure 2 Conceptual Elements of Dynamic Analysis. Arrows denote the direction of data flow.

3. Analysis of the NASA's GMSEC

3.1. Objectives of the Case Study

The NASA's GMSEC branch has developed the GMSEC software architectures as a reusable framework for missions inside and outside the NASA. In addition to their rigorous reviews and testing, they also prefer an independent organization to review the implementation quality, and report to them architecture/design issues as well as behavioral problems that could lead to failures.

3.2. General Process for the Analysis

In the first part of the analysis, the NASA's GMSEC team provided the GMSEC framework 2.6 as well as some example applications that illustrate the publisher-subscriber architectural style. The Fraunhofer team then analyzed this version statically from the point of view of product lines and software architectures. This analysis led us to understand the implemented software architecture of the GMSEC framework. After the analysis, the Fraunhofer team presented the discovered architectural issues to the GMSEC team, which addressed some of the high-priority issues in versions 3.0 and 3.1. The GMSEC team then provided the 3.1 version and a set of real applications for analysis. The Fraunhofer team set up a test-bed for running and performing dynamic analysis of the GMSEC, which added to the results in this paper. This fruitful process, which is supported by NASA IV&V's Software Assurance Research Program (SARP), has been going-on for a year.

3.3. Static Analysis of the GMSEC

The GMSEC source code contains several programming languages (C, C++, Java, and some Perl). We extracted code-level dependency models (e.g., include, import, call relations) and stored them as binary relations for querying using the RPA. We briefly explain how the dependency models were used to discover the software bus and the middleware abstraction layer in GMSEC.

Our approach was a combination of bottom-up and top-down strategies. Recall that we stored a list of middleware frameworks and the names of header files, classes, function/methods that deal with concept such as connecting to the middleware, sending, and receiving messages, etc., (see Table 1). In the bottom-up strategy, we queried the dependency models for all usages of commercial middleware frameworks and sockets. This led us to locate the directories and files of the GMSEC framework that implement the concepts of the publisher-subscriber style. We lifted the file-level dependencies to directory-level dependencies using the RPA's lift operator. Our conclusion is that the GMSEC framework has a clean implementation that separates the concerns of using and supporting several commercial middleware from providing software bus services to subscribers and publishers. The separation of concerns is implemented using a set of wrappers, one for each external middleware framework, as well as for the standard socket library. Each wrapper provides the same set of services to publishers and subscribers, thus hiding the differences between different middlewares. Thus, usages of external vendors' libraries are only allowed through a wrapper. For example, the ICE (Internet Communication Engine) is a commercial middleware that offers APIs for developing systems based on the publisher-subscriber style. The GMSEC framework offers a wrapper called *ice* that accesses the ICE APIs. All vendor libraries supported by the GMSEC framework are wrapped in the same way, see Figure 3.

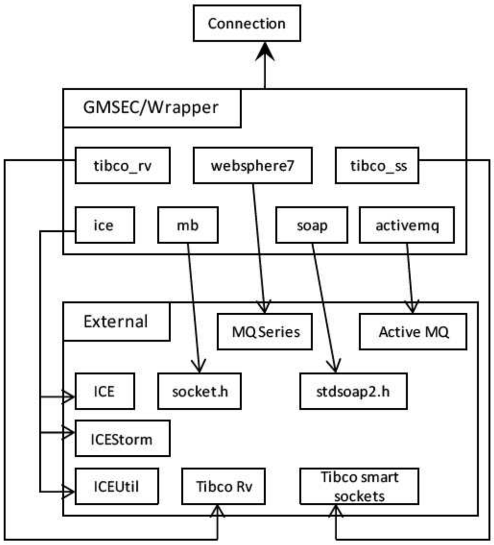

Figure 3 A slice of the GMSEC from the viewpoint of dependencies to external middleware and socket libraries. Boxes are directories (except socket.h and stdsoap2.h and Connection). Arrows denote the direction of module dependencies. The filled arrow denotes that each module within the wrapper folder inherits from the Connection class, which offers interfaces for publishing, subscribing, etc.

All wrappers are fully implemented in C++. Each wrapper inherits from the abstract base class called Connection which contains interfaces for connecting to the middleware, publishing, subscribing, etc. In addition to commercial middleware, the GMSEC team also implemented their own middleware, which is called the *software message bus* (mb), based on standard sockets. We queried the dependency models to understand who uses each wrapper of the middleware vendors, which showed that there are no static dependencies to the wrapper folder. In order to understand this design, we also used a top-down strategy using the simple example applications offered as part of the GMSEC distribution. The examples clarified that there is a class called *ConnectionFactory*, which is responsible for dynamically loading the wrapper of a vendor at run-time based on configuration settings. The build process of the wrapper showed that there is a dynamically loaded library (dll) for each vendor. For example, the wrapper implementation of the ActiveMq middleware is compiled into *gmsec_activemq.dll* for Windows, and *gmsec_activemq.so* for Linux. The source code of the ConnectionFactory class revealed that each wrapper implements standardized interfaces (e.g., CreateConnection), which are called by the ConnectionFactory at run-time to initialize the wrapper by loading the corresponding dll. The CreateConnection method of the loaded dll creates an instance of the corresponding middleware's connection class.

As discussed above, the core of the GMSEC is implemented in C++. However, the GMSEC also supports other publishers and subscribers being implemented in languages such as C, Java, and Perl. We analyzed the implementation of the core in order to understand how it handles variability due to programming languages. We used our text-based similarity tool for this purpose, and it revealed that there is an equivalent of Connection and ConnectionFactory class for each programming language. This was possible to discover automatically because the GMSEC team used the same method, variable names, and signatures in all programming languages. Since the GMSEC uses the "jni.h" file, which supports communication between Java and C++, it became clear that all calls to the Java implementation of the Connection, ConnectionFactory are redirected to respective C++ method calls using JNI [15]. Similarly, all calls to the Perl version of the Connection, ConnectionFactory are redirected to C++ method calls using the XS Perl to C++ interface [14].

To sum up the results from the static analysis, an attractive aspect of the GMSEC framework is that flexibility is built into the architecture, meaning that a) missions can easily add new middleware vendor of their interest by inheriting and implementing the abstract base class (Connection), b) missions can switch between different middleware using configuration settings and without changing the source code; the ConnectionFactory class will take care of loading and binding to the selected middleware wrapper, c) applications (i.e., publishers/subscribers) are agnostic to middleware vendor's API because they program to the vendor independent abstract base class, d) applications can be programmed in different languages, and e) applications can freely enter at run-time by connecting to the running software bus. From the static analysis, we understood the structure of the GMSEC, key interfaces, and classes involved in the publisher-subscriber style. We will now use this knowledge to analyze behavioral aspects of the style using dynamic analysis.

3.4. Dynamic Analysis of the GMSEC

During the static analysis, we observed that dependencies among applications of the GMSEC are impossible to extract statically because all communication is indirect using the intermediate software bus using middleware. Thus, it is difficult to determine exactly which application sends and receives messages. Therefore, we also conducted dynamic analysis, and customize Figure 2 to the GMSEC.

3.4.1. Defining Probes of the GMSEC Application

All "real" applications given to us are implemented in Java. Therefore, we have chosen AspectJ as the language for inserting probes [11]. Because the static analysis showed us that the Connection class is the core class for connecting to the middleware, publishing, subscribing messages, etc., we injected probes before and after the invocation of the methods of the Connection class in each application. We inserted probes "before" and "after" so that a) we could calculate the execution time of each method, and b) we could capture parameter values, which might be updated due to call-by-reference. We weaved our probes into the compiled binary class files of the GMSEC applications, which use the framework. Our probes emit run-time events as messages (with subjects "trace.before" or "trace.after") using the APIs of the Connection class, resulting in the publication of run-time events using the software bus itself. It is worth noting that we can use any middleware, for example a middleware, different from the one used for publishing/subscribing "real" messages, to send out trace messages. We use different middleware for "trace" and "real" messages in order to avoid any communication conflicts, see Figure 4.

3.4.2. Developing the RECO component

We developed the RECO component using the GMSEC APIs, and thus it can be plugged into the GMSEC run-time environment like any other GMSEC compliant application. The RECO plays the role of a subscriber by subscribing to all messages with the subject "trace.before" or "trace.after". One precondition for the RECO component is that it should use the same middleware that was used by the probes; otherwise, the trace messages will not be delivered to it by the GMSEC software bus (see Figure 4). We run the RECO component in monitoring mode in order to verify that it follows the behavioral constraints of the publisher-subscriber style and that it works well with all configurations of the middleware type. This is why there is a bidirectional arrow between the RECO component and the software bus for trace message. The RECO component reads traces of other applications and publishes its own traces (see Figure 4). Note also that the RECO component can be configured and deployed to run on a different machine, similar to other GMSEC applications.

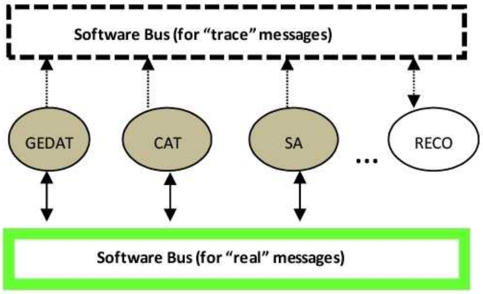
Software Bus (for "trace" messages)
GEDAT
CAT
SA
...
RECO
Software Bus (for "real" messages)

Figure 4 an overview of the environment for the dynamic analysis of the GMSEC. Each filled bubble is a GMSEC application. Two software buses are respectively used for publishing "trace" and "real" messages. RECO is the component for collecting traces emitted by other applications, including its own traces. Arrows denote data-flow.

3.4.3. Discovering C-C views and Sequence Diagrams

Readers who are unfamiliar with the concepts of CP-nets are requested to [7], which offers an in-depth discussion on how CP-nets were used for analyzing the Pipe-and-Filter architectural style of Ricoh's photocopy machine software. We used the same concept to perform analyses of the Publisher-Subscriber architectural style below. Here, we informally explain the design of CP-nets for discovering the C-C view of the publisher-subscriber style. We designed our CP-nets in a modular fashion, meaning that it was a composition of several CP-nets such as a) One CP-net for recognizing the creation of a connection to the software bus by monitoring call events to the 'Create' method of the GMSEC API (i.e., to the ConnectionFactory class explained in static analysis), b) One CP-net for creating the connection port used for publishing messages on the software bus. The CP-net implements this capability by monitoring call events to the 'publish' method of the Connection class, c) One CP-net for creating the connection port used for subscribing messages on the software bus. The CP-net implements this capability by monitoring call events to the 'subscribe' method of the Connection class, and d) One CP-net for attaching the ports of publishers with subscribers. Two ports are attached if one party consumes the messages published by the other party. Matching the subject of the published message with the subscribed message is the key activity of this CP-net.

In order to discover the C-C view, we ran the constructed CP-nets on events emitted by the running system. We used the RECO component, which places each run-time event into the different places that are responsible for holding call-events. Different parts of the CP-net processed the call-events, as explained above, and produced the C-C view as a collection of tokens. Here, we have manually drawn the C-C view by using the discovered high level events; see Figure 5 for an example. In addition, we can see all connections to the software bus that are created by each application and can determine how many they are, for example.

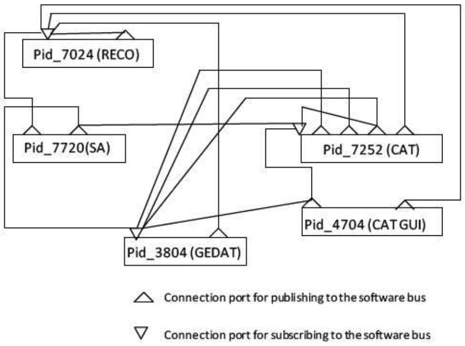
Pid_7024 (RECO)
Pid_7720(SA)
Pid_7252 (CAT)
Pid_4704 (CAT GUI)
Pid_3804 (GEDAT)
△ Connection port for publishing to the software bus
▽ Connection port for subscribing to the software bus

Figure 5 An example C-C view discovered using run-time events. Each box is a run-time process and the names of applications are placed in brackets.

All applications communicate with the RECO component because it consumes the run-time events that are published by the other applications. When we showed Figure 5 to the GMSEC team, they mentioned that this view is very useful as it nicely captures inter-communication among different applications at a high level of abstraction, a view that is normally difficult to create. The good news is that there are no surprising dependencies between the applications. However, one of the developers mentioned that he did not know the fact that the CAT has 3 connections to the GMSEC bus for publishing messages to other GMSEC applications. Thus, this view can be also used by developers to understand exactly the dynamic architecture of a complex system. In order to understand how messages are exchanged among different parties of the publisher-subscriber style, our CP-nets used the subjects of the messages that were subscribed by the subscribers and the subjects of the messages published by the publishers. If the subjects of the messages match, then our CP-nets store the connection between publishers and subscribers, the messages, and sending and receiving time of the messages. We visualized that output using the Dyn-SAVE tool, as shown Figure 6 for example. We can visualize messages and their data fields, too.

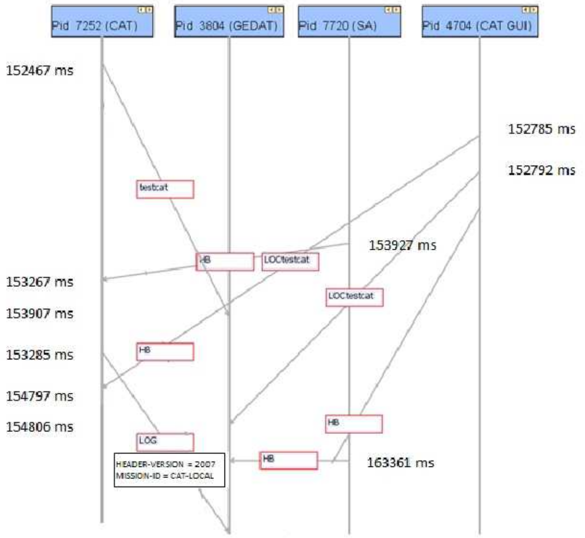

Figure 6 A snippet of the sequence diagram showing messages exchanged among publishers and subscribers. It also shows the time at which messages are sent / received. Contents of messages can also be visualized.

3.4.4. Detection of a High-Priority Bug

Here we briefly explain one of the bugs in the GMSEC framework, which is due to inconsistencies in the implementation of the same abstract interface (Connection) by different wrappers of middleware vendors, see Figure 3. We developed three CP-nets that check constraints of the publisher-subscriber style. One CP-net keeps track of all calls to the "subscribe" method of the Connection class, another CP-net keeps track of all calls to the "unsubscribe" method of the Connection class. A third CP-net detects multiple calls to the subscribe event, without an intermediate unsubscribe.

The CP-nets reported that the RECO component subscribed to the same message more than once. The GMSEC API has a feature that allows applications to have a call-back capability when a message arrives from other applications. We used that feature and subscribed to the same message three times, because we wanted to print "trace" messages in three different formats using three different call-backs. When we used the GMSEC's software bus (i.e., “mb” in Figure 3) to send and receive “trace” messages, the return code of all three calls to the “subscribe” method was NO_ERROR. We tested how activemq behave for the scenario and switched from the mb to the activemq middleware wrapper. Its “subscribe” method returned MIDDLEWARE_ERROR reporting that the RECO component makes multiple subscriptions to the same message.

We showed the RECO implementation to the GMSEC team, and discussed the behavioral inconsistency between the two middleware wrapper implementations of the same abstract interface (i.e., the Connection class, Figure 3), which caused the RECO to fail when we switched from one middleware wrapper to another wrapper. They agreed that this is an important bug and will fix so that all middleware wrappers will behave in an equivalent way with respect to their return code. Otherwise, applications cannot reliably choose and/or switch between different middleware wrappers. We would not have detected this important bug unless we modeled and verified the run-time behavioral properties of the publisher-subscriber style.

3.5. Answering the Questions

Question	Answers/Comments
1. Are there any middleware used for implementing a software bus?	Yes. The GMSEC software bus is implemented using middleware technology.
2. If middleware is used, is there any middleware abstraction layer that hides the knowledge of a particular middleware API?	Yes. There is an abstraction layer that “hides” vendor-specific APIs.
3. Can publishers and subscribers be implemented in different programming languages? If yes, how are the variants in the languages managed?	Yes. The core of the GMSEC is implemented in C++. However, there are interfaces for C, Java, and Perl. Language variants are managed using JNI [15] and XS [14].
4. Can publishers and subscribers come-and-go dynamically at run-time?	Yes. Publishers and/or subscribers can freely enter/leave the running system.
5. Can publishers, subscribers, and the software bus run on different machines?	Yes. Publishers and subscribers just need the IP address and the port number of the software bus.
6. Which subscribers receive messages from which publishers and in what order?	The discovered sequence diagrams answers this question for the scenarios we have executed.
7. Do subscribers receive messages from connections that are not subscribed by them?	No. However, we cannot extrapolate because dynamic analysis results are not generalizable.
8. Can subscribers subscribe to the same message more than once without an intermediate unsubscribe?	Yes and No. This is a bug the GMSEC team is currently fixing it.
9. Can subscribers unsubscribe to messages that are not subscribed by them?	No, not for the execution traces we analyzed.
10. Are there timing delays in delivering messages to subscribers?	Currently, we are analyzing timing aspect for different middleware configurations using the discovered sequence diagrams.

3.6. Connecting the GMSEC’s Business Goals with Implemented Architectural Decisions

In order to understand the relationship between the GMSEC’s business goals and the implemented high-level architectural decisions, we discussed with the GMSEC project manager, the product leader, and senior engineers. Based on this discussion, we were able to explicitly link the business goals and software architectural decisions (see Figure 7).

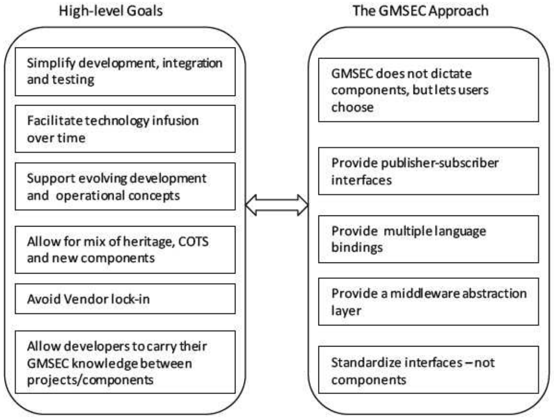
High-level Goals
Simplify development, integration and testing
Facilitate technology infusion over time
Support evolving development and operational concepts
Allow for mix of heritage, COTS and new components
Avoid Vendor lock-in
Allow developers to carry their GMSEC knowledge between projects/components
The GMSEC Approach
GMSEC does not dictate components, but lets users choose
Provide publisher-subscriber interfaces
Provide multiple language bindings
Provide a middleware abstraction layer
Standardize interfaces – not components

Figure 7 Relationships between business goals and supporting architectural decisions.

It is interesting to note from this study that important architectural decisions were primarily influenced by business goals. Therefore, it is advisable that the implemented architecture needs to consistent with the specified architecture, as shown to be true for the GMSEC with some exceptions mentioned above.

4. Brief Comparison to Existing Work

We will first list a few related articles and then highlight common differences to our work. Riva et al. extract both the module-structure and sequence diagrams by respectively using static and dynamic analysis [17]. Wendehals and Orso extract automata using execution traces [22]. Giannakopoulou et al. use LTL to verify behavioral properties of execution traces [10]. Schmerl et al. use the pair of architecture and implementation styles to discover C-C views of a running system [19]. Stroulia and Systa [21], and Cornelissen et al. provide an in-depth survey on other dynamic analysis techniques [1]. Dong et al. review methods and research tools for recognition of design patterns from the source code [2].

Key differences between our work and the existing work are: We extracted component-connector views, which showed the run-time structure of the software.

We used CP-nets to systematically tackle the interleaving of runtime events with respect to the software architecture. We discussed several adaptations of the DiscoTect method in [7]. Our sequence diagrams can a) hide the software bus for analyzing the publisher-subscriber style, and b) show attributes/parameters of messages exchanged between parties. If we exclude parameters, like in many existing work, we cannot distinguish calls to the subscribe method on two different message subjects, for example. Cornelissen et al. mentioned that there is a short-coming of research, in reverse engineering, on systems that evolve at run-time like the GMSEC. Regarding design pattern discovery, our focus was on bridging the abstraction gap between run-time events and the publisher-subscriber style.

5. Conclusion and Future Work

We presented a practical approach for analyzing systems based on the publisher-subscriber architectural style. We derived a set of analysis questions, which focused on both the structural and behavioral constraints of the style. First, we performed the static analysis to answer the questions related to the structural constraints. Second, we used the results of the static analysis to organize the dynamic analysis for answering behavioral constraints. We discovered component-connector views and sequence diagrams using execution traces, which were fed into our Colored Petri Nets, for tackling the challenge of interleaving of run-time events. Using this approach on the NASA's GMSEC, we discovered that the GMSEC has a) a good middleware abstraction layer, which helps in avoiding vendor lock-in, and b) has some high-priority bugs due to behavioral discrepancies among different middleware wrapper implementation. Our future work will focus on analyzing timing aspects of different middleware wrappers.

References

1. B. Cornelissen, A. Zaidman, A. Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. In IEEE TSE, 35(5), 2009.
2. J. Dong, Y. Zhao, and T. Peng. Architecture and Design Pattern Discovery Techniques – A Review. International Conference on Software Engineering Research and Practice, 2007.
3. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. ACM Computing Surveys, 35(2), 2003.
4. K. Jensen, Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Springer Verlag, 1993.
5. L. Feijs, R. Krikhaar, and R. Van Ommering. A Relational Approach to Support Software Architecture Analysis. Software Practice and Experience, 28(4), 1998.
6. D. Ganesan, M. Lindvall, D. McComas, and M. Bartholomew. Verifying Architectural Design Rules of the Flight Software Product Line. In Software Product Line Conference (SPLC), 2009.
7. D. Ganesan, T. Kueler, and Y. Nishimura. Architecture compliance checking at run-time. Journal of Information and Software Technology (IST). 51, (2009).
8. D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. International conference on Model checking software, 2003.
9. B. L. Ghezzi and C. L. Mottola. On Accurate Automatic Verification of Publish-Subscribe Architectures. In ICSE, 2007.
10. D. Giannakopoulou and K. Havelund. Runtime Analysis of Linear Temporal Logic Specifications. In ASE, 2001.
11. G. Kiczales, J. Lamping, A. Mendhekar, C. Maedar, C. Lopes, J.-F. Loingtier, J. Irwin, Aspect-Oriented Programming, ECOOP, 1997.
12. O. Lehr. A Framework for the Detection and Analysis of Software Connectors. University of Mannheim, Master Thesis, 2010.
13. M. Lindvall. Using sequence diagrams to detect communication problems between systems. In IEEE Aerospace Conference, 2008.
14. XS Module. http://en.wikipedia.org/wiki/XS_(Perl).
15. JNI. http://java.sun.com/j2se/1.4.2/docs/guide/jni.
16. A. Postma. A method for module verification and its application on a large component-based system. In IST 45, 2003.
17. C. Riva, J. Rodriguez. Combining static and dynamic views for architecture reconstruction. In CSMR, 2002.
18. G. Salton, A. Wong, and C. S. Yang. A Vector Space Model for Automatic Indexing. Communications of the ACM. 18, 613-620, 1975.
19. B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and Y. Hong, Discovering architectures from running systems, IEEE Transactions on Software Engineering 32 (7), (2006).
20. M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In COMPSAC, 1997.
21. E. Stroulia and T. Systä. Dynamic Analysis For Reverse Engineering and Program Understanding, Applied Computing Review, ACM, 10(1), 2002.
22. L. Wendehals and A. Orso. Recognizing Behavioral Patterns at Runtime using Finite Automata. In WODA, 2006.

Acknowledgements. Lisa Montgomery and her NASA IV&V team and Sally Godfrey NASA GSFC for supporting this work.