Architectural Modeling of Ozone Widget Framework End-User Compositions

Ivan Ruchkin **Vishal Dwivedi** **David Garlan**
Bradley Schmerl

June 2014
CMU-ISR-14-108

Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Ozone Widget Framework (OWF) is an event-based web platform for lightweight integration of widget applications. This technical report presents a formal model of OWF's widget composition mechanism. First, we present a detailed description of Ozone's end user composition mechanism. Then, we describe our architectural modeling approach and its value for analysis of OWF widget compositions. We go through the process of creating an architectural style to represent assemblies of Ozone widgets, reviewing modeling decision points and style alternatives.

This work was supported in part by the Office of Naval Research (ONR-N000140811223). Additional support was provided by the Center for Computational Analysis of Social and Organizational Systems (CASOS). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research, or the U.S. government.

Keywords: Ozone Widget Framework, end-user architecting, analysis, architectural model, Acme, Z.

Contents

1 Introduction

Event-driven programming has been a time-proven paradigm to create programs where the computation is determined by the flow of distributed events, thus enabling loosely coupled integration. While event-driven programming paradigm itself has shifted from being used in scientific and business programming languages like PL/1 and locally operating systems in 1960s to those designed for large Internet-scale integration, such as widget frameworks [12], its traditional problems have still not been solved. A standard problem scenario is that while event-based decomposition promotes modularity and simplicity by using communication over an event-based infrastructure, it also makes it possible to create convoluted programs that cause ripple affects [6]. Another issue with event-based frameworks is the difficulty of formal verification of their behavior [5]. These issues place a greater value on the principled design of event-based systems that promotes required qualities and inhibits the undesirable ones. And, as the systems are growing to the Internet scale, designing them in a controllable manner becomes even more important because their use affects numerous organizations and social groups.

Another trend in software engineering is the shift of software development by end users who are not merely users, but developers in their own regards [2]. More and more often, users are provided not with a rigid interface with few points of customization, but with a powerful and expressive means of creating a customized software system. End-user architecting [7], a sub-discipline of end-user software engineering, deals with end-user environments that allow people to do work similar to that of software architects: compose computations from coarse-grained functional blocks to create a system with given functionality and qualities. One of the essential goals of end-user architecting is supporting end-user architects in their activities through model-based tools and techniques that have already gained respect in the software architecture community.

Frameworks in these end-user programming domains present interesting research challenges and therefore there has been a significant interest towards such research. Such systems provide end users with a visual way to combine components and wire them through an event-oriented environment. For such systems, formal representation and analysis can be beneficial, as it can help users deal with a hard-to-manage event environment. Also, design decisions for such frameworks are of interest because they promote or inhibit specific qualities during their usage. Furthermore, not all design decisions that go along designing these event frameworks are explicitly laid out, and these implicit decisions may have profound influence on the quality attributes of the system that uses the framework. It is worth making such hidden decisions explicit in order to clearly articulate their results. In this report, we discuss one such framework named Ozone Widget Framework (OWF) [10], or just Ozone, which is an open-source event-based integration framework, popular in intelligence and data analysis communities. OWF enables rapid assembly and configuration of rich, lightweight web applications (called widgets) produced by different third party contractors to provide various reports on data. OWF supports integration of such web-applications using a common publish-subscribe event model, where individual widgets can be hosted on different servers, but are integrated together on a centralized portal to provide summary views of dynamic information content.

OWF was designed to be a simple and robust end-user framework. Unlike most widget frameworks today that require writing extensive code, Ozone abstracts out unnecessary technical details from the end users and provides a simple web front-end where end users can compose widgets and restrict communication amongst widgets. Additionally, Ozone provides APIs based on JavaScript to standardize the development, and ease the integration effort. However, this abstraction comes at a cost: it makes the underlying Ozone computation model more complicated that needs better understanding. Architecturally speaking, a major deviation from publish-subscribe style of computation comes from additional restrictions over event communications, and other abstractions that hide communication details. This aroused our interest to explore OWF's computation model further, and we formalized the principles of composing OWF widgets to understand and document it better. In doing so, we stumbled across various design decisions of OWF – both explicit and implicit – that affect various qualities during its usage, and analyzed those decisions in their effect on qualities of OWF.

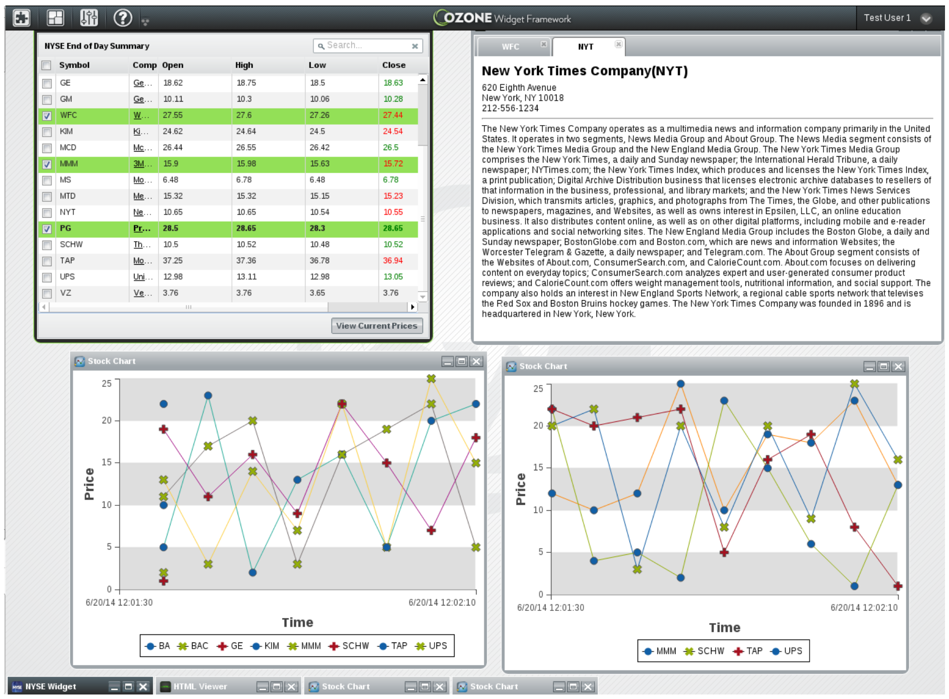
However, this modeling of OWF was not straightforward. And therefore, one of the goals of this report is to document the tradeoffs associated with formal modeling and analysis. This report describes our steps to formally model any such system, interpreting the vocabulary of the design constructs, and the design decisions that lead to particular quality attributes. Apart from providing a better understanding of the system, this exercise could be beneficial to end users in many ways: first, this could help them understand the details of low-level interaction using appropriate abstractions when required (which are at times opaque in many frameworks like OWF); and second, it could help them answer the various what-if questions for which currently they have to dig into code or just guess.

We used an architectural style written in Acme architecture description language [8] to capture the model of Ozone widget compositions; it can be used to define various domain-specific analyses. End users can use such analyses to examine the feasibility of a particular compositions, and examine them for problems at a level of abstraction that can be directly understood without detailed knowledge of the underlying technology. Also, the modeling process has provided the authors with enough insight to discuss Ozone's design decisions and their impact of quality of this framework.

The rest of the report is organized as follows. The next section describes Ozone's computational model from an end user's point of view. Section 3 presents our architectural approach to modeling: why we chose it and how we tailor it to this framework. Section 4 gives our detailed steps towards creating an architectural style for OWF compositions and alternatives of the style; also, this section describes the style and an example of formal analysis based on the style. And finally, we discuss future research directions in Section 5.

2 Ozone Widget Framework

As we mentioned above, Ozone Widget Framework[1] allows its users to place widgets on a web dashboard. Users of Ozone accesses this dashboard through a web browser. Widgets, technically being HTML iframes, are opened on this dashboard. Users selects a widget type to deploy (for example, a graph plotting widget), and after that a corresponding widget instance is created on the dashboard. Thus, one can create many instances of the same widget and work with them separately. In this report, the word *'widget'* is invariably used as an instance of a widget. You can see a dashboard with four widgets in Figure 1.



	Symbol	Comp	Open	High	Low	Close
☐	GE	Ge…	18.62	18.75	18.5	18.63
☐	GM	Ge…	10.11	10.3	10.06	10.28
✓	WFC	W…	27.55	27.6	27.26	27.44
☐	KIM	Ki…	24.62	24.64	24.5	24.54
☐	MCD	Mc…	26.44	26.55	26.42	26.5
✓	MMM	3M…	15.9	15.98	15.63	15.72
☐	MS	Mo…	6.48	6.78	6.48	6.78
☐	MTD	Me…	15.32	15.32	15.15	15.23
☐	NYT	Ne…	10.65	10.65	10.54	10.55
✓	**PG**	**Pr…**	**28.5**	**28.65**	**28.3**	**28.65**
☐	SCHW	Th…	10.5	10.52	10.48	10.52
☐	TAP	Mo…	37.25	37.36	36.78	36.94
☐	UPS	Uni…	12.98	13.11	12.98	13.05
☐	VZ	Ve…	3.76	3.76	3.65	3.76

New York Times Company(NYT)

620 Eighth Avenue
New York, NY 10018
212-556-1234

The New York Times Company operates as a multimedia news and information company primarily in the United States. It operates in two segments, News Media Group and About Group. The News Media segment consists of the New York Times Media Group and the New England Media Group. The New York Times Media Group comprises the New York Times, a daily and Sunday newspaper; the International Herald Tribune, a daily newspaper; NYTimes.com; the New York Times Index, which produces and licenses the New York Times Index, a print publication; Digital Archive Distribution business that licenses electronic archive databases to resellers of that information in the business, professional, and library markets; and the New York Times News Services Division, which transmits articles, graphics, and photographs from The Times, the Globe, and other publications to newspapers, magazines, and Websites, as well as owns interest in Epsilen, LLC, an online education business. It also distributes content online, as well as on other digital platforms, including mobile and e-reader applications and social networking sites. The New England Media Group includes the Boston Globe, a daily and Sunday newspaper; BostonGlobe.com and Boston.com, which are news and information Websites; the Worcester Telegram & Gazette, a daily newspaper; and Telegram.com. The About Group segment consists of the Websites of About.com, ConsumerSearch.com, and CalorieCount.com. About.com focuses on delivering content on everyday topics; ConsumerSearch.com analyzes expert and user-generated consumer product reviews; and CalorieCount.com offers weight management tools, nutritional information, and social support. The company also holds an interest in New England Sports Network, a regional cable sports network that televises the Red Sox and Boston Bruins hockey games. The New York Times Company was founded in 1896 and is headquartered in New York, New York.

Figure 1: Ozone dashboard with four widgets.

The core service OWF provides to widgets is message exchange. It allows them to send and receive information from each other when they are open in the form of *messages* (or *events*). Such communication happens through *channels*. A channel is a runtime entity to which widgets can publish and subscribe. The semantics of publishing and subscribing in Ozone is similar to the general publish-subscribe architectural style [5]: if a widget publishes to a channel, all subscribers receive the message. There is no caching or logging of published events, so there is no way to read the "history" of messages sent to any channel. Each channel has a unique name that is used by widgets when they publish and subscribe to it. Channels tunnel only plain-text data.

From Ozone's point of view, a widget's subscription to a channel is a persistent aspect of the system's state: it can be canceled only by explicitly unsubscribing (done by the widget's code)

[1]This report targets OWF version 6.0.2, released publicly at [10].

or removing the widget (done by the user). At the same time, publishing is a one-time event: a widget executes an appropriate command, and a message is delivered. There is no preparation required for a widget before it engages in publishing as well as no obligation after it publishes. This distinction between subscribing and publishing is important for further modeling of Ozone widget compositions.

Widgets are produced and maintained by third parties. It is up to a user which widgets to deploy of his dashboard. Once a widget is downloaded by the user, its front end runs inside the user's browser, whereas back end (if any) may be hosted on some other server. Event messaging also happens inside the user's web browser. This approach makes it easier to assemble widgets into systems that are capable of performing complex computations.

Publishing and subscribing is a powerful mechanism of interaction, yet it is a complex one: a person who designs event-based interaction has to provide for appropriate reaction for relevant events for every widget as well as for naming of channels and protocols of event exchange. Perhaps having these considerations in mind, Ozone's developers did not permit end users to explicitly alter what and where is published by any widget. It is up to widgets' programmers to determine names of channels widgets subscribe to and event processing and publishing. This information is encoded in widgets' sources and is out of users' control unless a widget's developers design a user interface for users to work with the mentioned details of event exchange. Moreover, event communications paths are completely opaque for users since channels have no visualization in Ozone dashboard.

2.1 Eventing Restrictions in Ozone

To compensate for the users' inability to control communications, Ozone gives them a tool to limit event exchange — *a restriction line*[2]. Initially, any widget can subscribe to any channel and, consequently, receive messages from all other widgets that publish to the channel. Similarly, any widget can publish messages to any channel; the messages will be delivered to all the subscribers of this channel. Restriction lines control which widgets can talk to which. Once a user draws a restriction line between a pair of widgets, he blocks all communication between each of widgets in the pair and all the other widgets, but preserves the communication inside this pair. To permit other directions of message exchange for any of widgets in this pair, the user would have to draw another restriction line or to delete the first one.

More precisely, two distinct widgets in Ozone composition are permitted to communicate (via any channels — existing or created in the future) according to eventing lines if and only if one of two conditions holds:

- None of these two widgets is connected by the restriction line to any other widget.
- These two widgets are directly connected by the restriction line.

Note: a widget will always get its own messages from channels it both publishes and subscribes to no matter how restriction lines are placed.

Restriction lines are drawn by end users in a separate view. You can see a screenshot of such a view in Figure 2.

[2]Starting OWF version 6, restriction lines are disabled in a default installation.

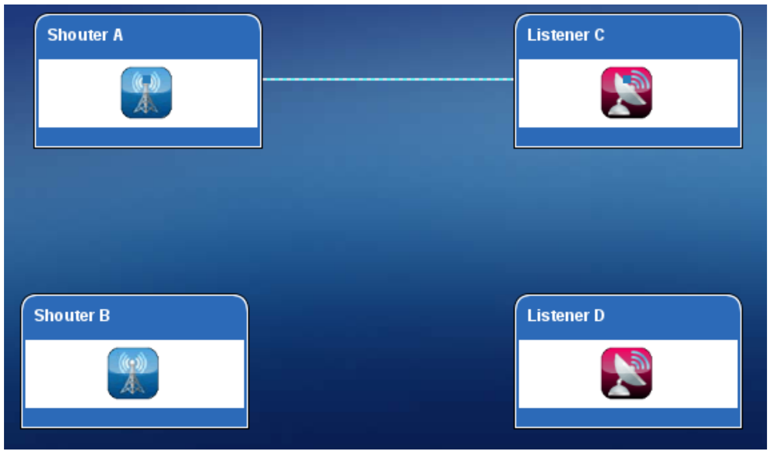


Figure 2: Ozone view for restriction lines. A line connects Shouter A and Listener C.

You can find the unambiguous documentation of Ozone messaging behavior and restriction lines carried out with a Z specification language model [11] in Appendix A. This Z model, while being mathematically precise and correct, does not allow for as much end-user support as an architectural model of widget compositions that is created in Sections 3 and 4. This motivates our deeper exploration of the computational model behind Ozone event system.

By deploying widgets that exchange messages and by manipulating restrictions on this messaging, users can create arbitrary dynamic *compositions* of widgets (also called *configurations* or *orchestrations*) that can serve a variety of computational purposes. The next section looks closer what information is present in an Ozone composition. It is helpful to be able to reference different parts of it when discussing modeling and analyses.

2.2 Information in Ozone Composition

Each OWF composition is a superposition of following *units of information*:

1. The set of widgets deployed on the dashboard. This is a basic unit of information that is required so that subsequent ones are meaningful: it is impossible to describe what widgets publish to channels if the set of widgets is not defined.

2. The set of existing channels and publishing and subscribing relations on widgets and channels. This unit is based on how widgets are implemented because publishing and subscribing to channels is generally specified in source code.

3. *Restrictions*. The set of all restriction lines is the information in this unit. This information depends of how end users set up restriction lines. There is an important sub-unit of information here that actually determines which communications work and which do not:

   a. The relation on widgets, “who can talk to whom”. It describes which widget is allowed to talk to which one according to restriction lines. This unit of data can be derived from the set of all restriction lines, but not the other way around.

4. Actual communications [3]in the system. This unit of information describes pairs of widgets that actually exchange events. This information can be inferred from the units 1–3. The space of all ongoing communications is split by restriction lines into two sets: public ones and private ones.

As we will see later, an ability of an OWF composition model to contain these units of information determines analyses that the model can provide.

2.3 Motivation for Formal Modeling

Formal modeling of OWF widget compositions brings the following benefits:

- Ozone is one of few representatives of event-based systems for end-user architecting. Precisely describing Ozone's compositions is valuable to study this class of systems.
- Clarifying decisions that Ozone's creators made and their impact on qualities of Ozone may provide insight into what are tradeoffs when designing an end-user composition environment, at least in the domain of event-based environments.
- A formal model documents principles upon which widgets are combined and facilitates the unambiguous understanding of these principles. For example, the exact rules of restriction lines' behavior are not precisely documented and were discovered empirically by the authors.
- If a visual formal model of compositions is implemented as a view in Ozone, it can help users by explicitly showing channels and publish-subscribe relations between them and widgets.
- A formal model permits formal analysis that can help end users. For example, because of the complex logic behind restriction lines, it is presumably difficult for users to evaluate changes introduced by adding or removing a restriction. Automatic analysis can show the effects of such an action to users.

In the next section, we present our approach to modeling Ozone widget compositions.

[3]A communication is an ongoing message exchange between two widgets through a channel.

3 Architectural Approach to Formal Modeling

This section describes our approach to modeling OWF compositions through architectural representation.

3.1 Architectural Style

As described in the introduction, end-user analysts' activity while using composition tools (and Ozone in particular) is similar to that of software architects. This fact brings us to modeling Ozone compositions with an architectural style [1] in an architecture description language (ADL) [4]. There are several specific reasons to this decision. First, Ozone compositions form a runtime architecture of an ad hoc computational system, and the component and connector model is an accepted way to represent runtime architectures [9]. Second, a style describes a vocabulary of a system's elements and rules of combining them and unambiguously explains what a system is and what it is not. Last but not least, having a style would permit us to perform analysis on compositions to help end users in their tasks. We use Acme ADL to create an architectural style for OWF because Acme has strong support for customized analysis of architectural models and style extension.

An Acme architectural style (called *family* in Acme language) in the is a collection of types for modeling runtime elements of software systems. These types describe the following elements:

- Components — principal computational elements of a system. Components can contain ports.
- Ports — components' interfaces for interaction. Ports can only be attached to roles.
- Connectors — elements that encapsulate interaction between components. Connectors can contain roles.
- Roles — responsibilities for interaction in connectors. Roles can only be attached to ports.
- Systems — configurations of components and connectors attached connected to each other through ports and roles. In this report we use the term *compositions* meaning actual assemblies of widgets or ADL systems that model them.
- Rules — first order logic predicates that specify what systems are considered valid. Rules can be either invariants (cannot be violated ever) or heuristics (can be violated selectively).

See Figure 3 for illustration of the mentioned ADL concepts.

Apart from providing a clear vocabulary of OWF composition elements and giving an opportunity for analysis, having an architectural style has additional benefits:

- Potential reuse of analyses in other end-user composition systems that would have the same or similar architectural style.

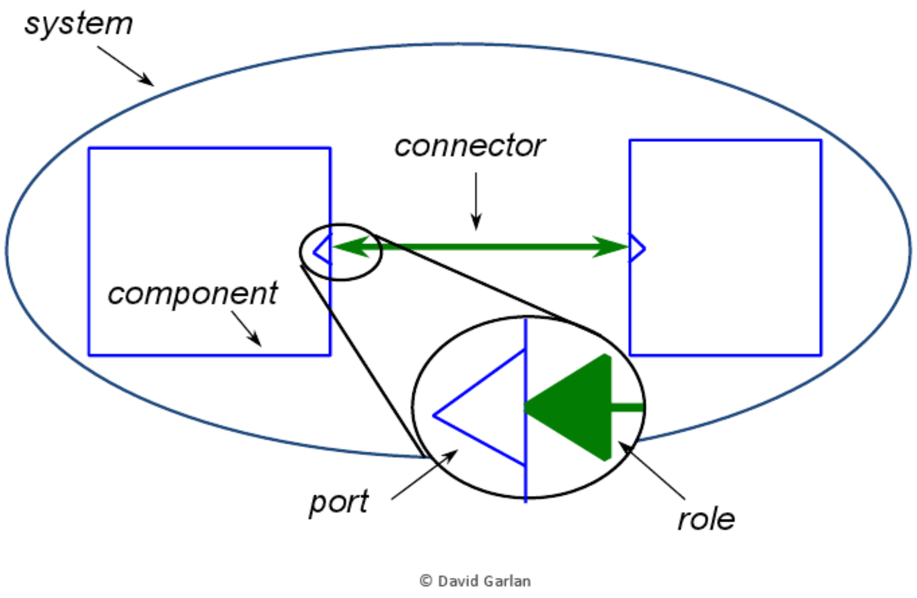


Figure 3: Structural Primitives of Component and Connector Model

- A style can be a basis for a more detailed model, should the need arise. For instance, a specialization of OWF style could permit analysis for data mismatch detection and repair. Another example of specialization is adding more detailed components that describe widget of a particular domain, e.g. geospatial intelligence analysis.
- An architectural representation makes communication pathways explicit to users unlike the current Ozone visualization does. If users could see what channels are used, it would arguably make it easier for them to operate restrictions. Hence, a visual implementation of an architectural model in Ozone supports the activity of end-user analysts.

3.2 Requirements for Ozone Style

Using an architectural style, we can fulfil the motivation for formal modeling described in Section 2.3: a style provides a way of modeling a system, and modeling leads to clear documentation of underlying principles of Ozone and to analysis of widget compositions. However, not every style is equally good at describing composition laws and at providing analysis. So, we come the following style-specific requirements:

- Simplicity and understandability of systems produced by the style vocabulary. It means that models of OWF compositions should add as less complexity as possible to the inherent complexity of each particular composition. One important aspect is "scalability" of a style—measure of how much the number of the model's elements grows in response to an increase in the number of elements in actual composition. For example, if adding one channel in OWF world would make us create 3 connectors in the model, this way of modeling is less scalable than the one that would react with 2 connectors. Number and complexity of properties (of components, connectors, ...) also affects the simplicity.

- Providing enough information for analyses. For example, including information about where restriction lines go allows tools to predict how drawing a new line would affect message exchange, whereas not including such information makes this analysis impossible. [4]

Before we start designing an architectural style for OWF compositions, it is worth outlining the potential analyses that the style might support. Style design alternatives will be evaluated based on what analyses are enabled and disabled by them.

3.3 Architectural Analyses

A formal architectural model of OWF user compositions can theoretically support analyses that are described below. The analyses are supposed to help users overcome speculative [5] shortcomings of OWF or extend their possibilities in assembling widget-based systems. To classify analyses, they are grouped by ideas and discriminated in terms of their inputs. It is convenient to use units of information defined in Section 2.2 as inputs [6].

- Message reachability analysis. This analysis may come in different versions depending on what information about a composition is available. This analysis helps users overcome OWF's inability to inform about what events are delivered to which widget. See Table 1 for variants of this analysis. Most of variants in this table answer questions: "who can talk/is talking to whom?" or "how to make these widgets talk?".

- Detecting data loss channels. A *data loss channel* is a channel to which at least one widget publishes, but no widget is subscribed. An analysis can warn end users of such situations. See Table 2 for variants.

- Privacy analysis. Widgets are divided into two classes: widgets from trusted parties and untrusted parties. The goal is to prevent leaking of private data from trusted widgets to untrusted ones. This analysis needs additional specification of trusted and untrusted widgets. See Table 3 for variants.

- Type mismatch analysis. It might turn out that some widgets are publishing to channels they are not supposed to (e.g. users get control over assigning channels or there is a channel naming conflict in widgets coming from different parties). This situation can result in misbehavior of widgets because they might not be programmed to have wrong structure

[4]Choosing between the two requirements is a tradeoff: a style may produce quite simple systems with transparent structure and ways of interaction, but such a simple model would probably not contain enough information for most of analyses mentioned in the previous section.

[5]To find out actual usability problems in Ozone, a user study is needed.

[6]We assume that Unit 0—the set of deployed widgets—is always provided by a style.

Input Unit	Additional Input	Description of analysis	Output
1	Two sets of pairs or widgets	How to set up restriction lines to make these pairs of widgets communicate and those pairs not communicate?	Set of restriction lines
3a	-	Can restriction lines be recreated from the given restrictions? If yes, what are they? An answer provides Unit 2 and enables corresponding analysis.	Potentially, set of restriction lines
2	An existing restriction line	What communications will be enabled and disabled by creating or removing the given restriction line?	Two sets of communications

Table 1: Variants of message reachability analysis.

Input Unit	Additional Input	Description of analysis	Output
1	-	What are data loss channels? In this case, a non-empty answer indicates that a user might have forgotten to deploy some widgets.	Set of channels or connectors.
3	-	What are data loss channels? In this case, a non-empty answer indicates that a user might have set up too strong restrictions on communication.	Set of channels or connectors.

Table 2: Variants of discovering data loss channels.

of messages in channels they are subscribed to. This analysis helps detect such situations. It needs additional specification of data type for each publishing widget. See Table 4 for variants.

- Mining common widget configurations. This analysis examines users' behavior and extends users' possibilities by recognizing common patterns in compositions, helping to establish them quickly, and showing differences between current compositions and patterns. See Table 5 for variants.

3.4 Style Design Heuristics

In the context of this report, choosing a style means committing to a particular way of modeling Ozone compositions. As it turned out, it is difficult to create a style that would evidently defeat all other options and, therefore, be the best solution for this modeling task. The design space of styles for OWF is non-trivial and vast. This is the reason why the next section is devoted to a detailed discussion of decision points and style alternatives.

Input Unit	Additional Input	Description of analysis	Output
1	-	What are potential privacy violations?	Set of channels or connectors.
3	-	What are actual privacy violations?	Set of channels or connectors.

Table 3: Variants of privacy analysis.

Input Unit	Additional Input	Description of analysis	Output
1	Data type for each port.	What is potential data type mismatch?	Sets of widgets that have data type conflict.
3	Data type for each port.	What is actual data type mismatch?	Sets of widgets that have data conflict.

Table 4: Variants of type mismatch analysis.

The style alternatives will be informally evaluated based on the two style requirements. However, there is a number of loose style design heuristics[7] that influence reasoning that is conducted below. Here is a list of the most relevant heuristics:

- Using properties of an architectural element (component, ...) to reference other architectural elements is not considered a good practice. There are two reasons for that: (i) it makes architectures less obvious because many information is hidden in properties, and (ii) rules that constrain values of such properties tend to be difficult to write, read, and evaluate.

[7]Style design heuristics (rules of thumb in style design) and Acme language heuristics (first order logic predicates that are not strictly obligatory) are not to be confused.

Input Unit	Additional Input	Description of analysis	Output
2	Statistics of usage	What are patterns of restriction lines?	Sets of restriction lines
3a	Statistics of usage	What are patterns of restrictions?	Sets of restrictions
3	Statistics of usage	What are patterns of communications?	Compositions of widgets and channels
3	Statistics of usage	What is the difference between a pattern and a current composition?	A set of widgets and a set channels

Table 5: Variants of pattern detection analysis

- Components with their ports should be *interaction-independent*. It means that a component or a port should not have properties that are related to the state of communication with other components—such information should be stored in connectors and roles.
- Connectors represent interaction between components, not just any relation between components. For instance, connectors are not intended to directly model restriction lines.
- Complex connectors should be avoided. The complexity of a connector is determined by how complicated the connector's protocol of communication is. For example, if there is one event bus that serves pairs of components selectively based on several properties of roles (e.g. if a certain numeric property of two roles is equal), it is probably worth modeling it in a more detailed way.

Now, we are ready to go through the steps of devising an architectural style for OWF compositions.

4 Formal Modeling of Ozone Compositions

This section is devoted to detailed modeling Ozone widget compositions: it goes through decision points in the style design, evaluation of style alternatives, and the description of the finally selected style.

4.1 Basis of Modeling

The task is to come up with an architectural style for Ozone widget compositions. This section describes basic decisions. However somewhat apparent, they have to be mentioned to set up a foundation for less trivial aspects of creating a style.

First of all, widgets are to be represented with components through a strict one-to-one mapping. This decision rests upon that domain-specific computation is performed solely by widgets. We do not add any types of widgets or specific properties for them in this style because we are mostly interested in widgets' interactions (modeled by ports), not their internal attributes. Nevertheless, specific properties of widgets can be added in specializations of the designed style.

As ports express the intent of inter-component communication, there is little flexibility of how they are used. Two types of widgets' actions—publishing and subscribing—directly map into two corresponding types of ports.

If we speak in terms of how OWF technically works[8], publishing happens instantly when the corresponding code is executed; that is, publishing is more an action than a process or a state. However, Acme language implies a more structure-oriented picture of runtime and compels us to view publishing as a recurring interaction (relation) and to model it with a port-role attachment.

The rest of modeling decisions are less straightforward and are covered in the next section.

4.2 Decision Points for Style

While designing a style, we came to a number of decision points that were not an easy task to go through. They are intertwined with each other, and each option presents tradeoffs that affect the resulting style. This section lists main decision points, alternatives, and their impact. The subsections follow the same pattern: a description, a list of options, a list of tradeoffs. Neither the list of decision points, nor lists of options or tradeoffs are comprehensive: the decisions, when combined, present even more detailed options that are not described.

4.2.1 Representing Communication

Description: Message exchange between widgets is carried out through channels. Representing channels and communication through them is a major decision that affects scalability of the style and understandability of produced compositions.
Alternatives:

[8]See Appendix A for details in Z model.

RC1 Channels are one-to-one with connectors: each channel is modeled by one and only one connector.

RC2 Connector for each pair of components interacting: if there is communication between two components going, a connector for this pair is added to the model. All connectors have one publisher role and one subscriber role.

RC3 1-2 event buses are used to describe all channels. One bus is used in case if interactions are not divided to private and public, and two buses are used otherwise.

RC4 Connectors as "subscribe-sets"[9] of a publishing component: components that publish information are complemented with a connector to which all subscribers of this component are connected.

RC5 Connectors as subchannels[10]. Every channel may have a public communication and a number of private communications (those enforced by restriction lines). Every such communication (subchannel) is represented with a connector in the model.

Tradeoffs:

1. Having connectors that represent several channels at a time makes diagrams cleaner (fewer connectors), but increases the complexity of connectors.

2. If connectors do not contain enough information about which channels are present, it hinders analysis capabilities.

3. If a connector strictly represents one channel, it results in ambiguous communication pathways (which widgets talks to which through this channel?) or creating complex role types (that store mappings to each other in order to preserve paths of interaction).

4.2.2 Representing Communication Types

Description: As was explained in Section 2.2, it is an important aspect of Ozone framework that every message is dispatched either through public or private communication domains. Representing this information in a style helps its understandability and in many cases can be a sufficient replacement for representing restrictions (as we see later).
Alternatives:

RCT1 Public/private roles. Type of role models type of communication (public or private) that is done through it.

RCT2 Public/private connectors. Type of connector models type of communication (public or private) that is done through it.

[9]A *subscribe-set* of a widget is a set of widgets that are subscribed to that widget through all channels it publishes to.

[10]See Section 4.4 for a more formal definition of the term *subchannel*

RCT3 No representation. Interactions are not divided into public and private.

Tradeoffs:

- These options depend on what kind of connector serves widget interaction, and not all combinations are possible.
- Not representing communication types makes the architectural model of compositions not information-rich enough for most analyses.

4.2.3 Representing Restrictions

Description: Restriction lines set up the relation of permitted communication[11]. In other words, after a user draws a set of restriction lines, Ozone is able to answer the question, "who is permitted to talk to whom". It is a relation in the space of components. Acme language lacks first-class entities for representing custom relations on components. This fact brings about a question if we need to model restriction lines as they are (to be able to understand where they go given an Acme model of a composition), to model only restrictions (because they actually define what communication are permitted), or not to model at all.
Alternatives:

RR1 Restriction lines represented with properties: every component gets a property that lists other components to which it is connected.

RR2 Restrictions represented with properties: every component gets a property that lists other components to which it is allowed to communicate to.

RR3 External restrictions: restrictions are stored in a separate file and are used as an input for analysis.

RR4 No representation: restrictions are not represented explicitly in our architectural model.

Tradeoffs:

- Modeling a restriction line leads to a complicated style with many constraints that are difficult to write and evaluate. Also, it lowers the style's understandability.
- Externalizing restrictions can be a delayed decision: it is possible to add the external restrictions up to any style that does not represent them.

[11]This relation specifies what widgets are permitted to deliver messages to which.

4.2.4 Modeling Incomplete Communications

Description: This decision point depends on how we choose to represent communication; the least generic question to be answered is, "does architectural model of OWF compositions display inactive[12] connectors?" For example, if a widget is subscribed to a channel that no other widget is subscribed to, should this channel be modeled or omitted from the model?
Alternatives:

MIC1 Include incomplete communications into the model. Depending on choices in other decision points, it might be a connector, a role, or a port that are not contributing to the picture of inter-widget communications (for instance, because of a restriction line).

MIC2 Include only elements (components, connectors) that form successful ongoing communications.

Tradeoffs:

- Omitting not active communications makes diagrams clearer and more understandable, but inhibits opportunities for "what-if" analysis.
- In some cases, one cannot include all incomplete communications (this opportunity may be disabled by other choices), and we would have to limit ourselves to partial choices, which increases the complexity of the style.

4.2.5 Default Ports and Roles

Description: In Acme, default ports for a component type can be specified, as well as roles for a connector type. If done so, every component (connector) of this type is created with a port (role) that cannot be deleted. That is, the decision is how to allocate default roles and ports to connectors' and components' types respectively.
Alternatives:

DPR1 Components [connectors] have a default publishing port [role] and/or a default subscribing port [role].

DPR2 Components [connectors] do not have default ports [roles].

Tradeoffs:

- Making a decision for this point might make a style more usable: if a connector cannot exist without some role, then this role should be a default one for the connector.
- This decision is based on semantics that is associated with port [role]: is it, for instance, "can publish" or "publishes"?

[12]A connector/channel is *inactive* if no events flow through it. Otherwise, it is called *active*

4.3 Style Alternatives

Having the decision points above, it is impractical to decide on each one separately because combinations of various options cause unpredictable benefits, downsides, and further branching of decisions. Thus, styles need to be evaluated independently. To provide illustration for each style, we model a sample system (Figure 4): four widgets communicate through channels; widget A is linked by restriction lines to widgets C and D.

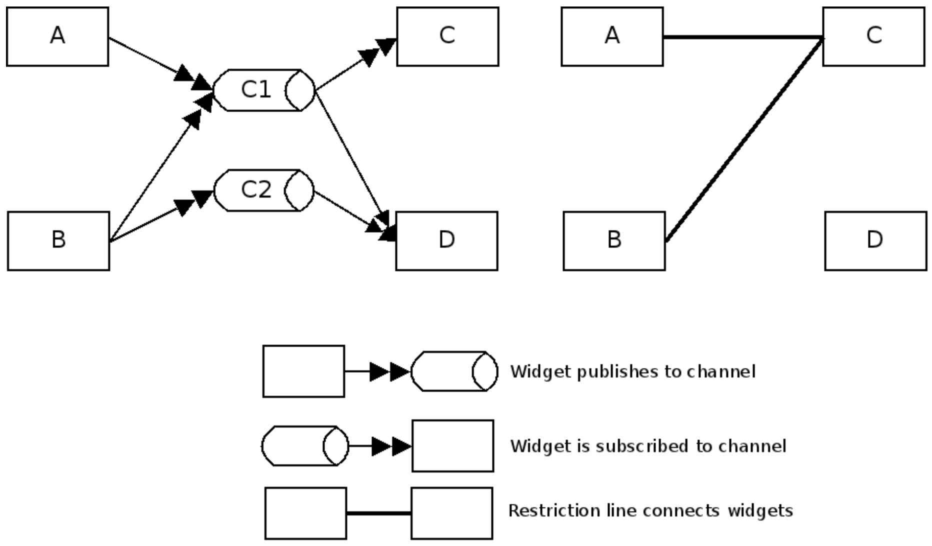

A
C
C1
C2
B
D
A
C
B
D
Widget publishes to channel
Widget is subscribed to channel
Restriction line connects widgets

Figure 4: Sample system for modeling

Note: The option of representing restrictions is independent from other style design choices, and can be decided separately. For the sake of shorter review of alternatives, we do not consider any options other than RR4. Note that any OWF style can be augmented with one of the options above, but question is not considered in this report.

Below, several style alternatives are evaluated.

4.3.1 Style Alternative 1: Public and Private Roles

Decisions: RC1, RCT1, RR4, MIC1, DPR2.

This style alternative is built on choosing the first option in Representing Communication: each Ozone channel is matched with one and only one connector in the model. Let us also choose public/private roles to describe different types of communication. Unfortunately, modeling communication pathways unambiguously with this style is impossible unless we attribute roles with additional information about connections. Consider an example: two pairs of widgets privately communicate through one channel. Architectural model in this case would have one connector with 4 roles of two different types (publisher and subscriber). Without adding properties for roles, the actual pathways of events are not clear because they can go through any pair of differently typed roles. Hence, at least private subscribe roles have to have an attribute set that contains names of components that that component is listening to [13].

You can see a model of the sample system in Figure 5. Notes show the attributes of subscribe roles—sets of components that roles receive information from.

Units of information: 1, 2, 4.

Advantages: Simple correspondence between OWF channels and connectors.

Disadvantages: Complicated logic of connectors; complicated roles.

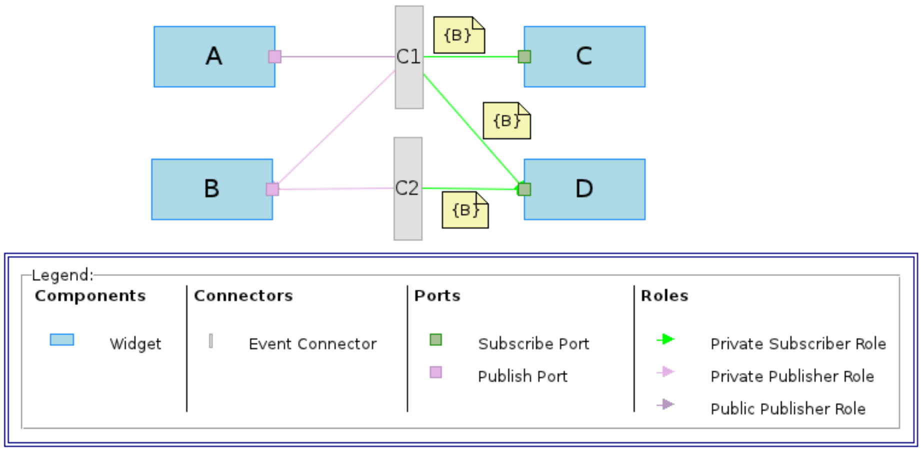


Figure 5: Model of sample system in style alternative 1

4.3.2 Style alternative 2: Connector per Interaction

Decisions: RC2, RCT2, RR4, MIC2, DPR1.

For this style, we choose to have a connector for every pair of interactions. In this case, we can assign typing to connectors since each of them represents an atomic interaction, which is always either public or private. It also makes sense to have a default publisher and a subscriber roles for each connector since it will always connect a pair of interacting components. To preserve information about mapping from channels to connectors, each connector would have a string attribute with a channel name. A model of the sample is shown in Figure 6.

Units of information: 1, 2, 4.

Advantages: Public and private interactions are visually easier to tell apart.

Disadvantages: Many connectors on diagrams; not scalable for many channels (number of connectors grows quickly); difficult to visually identify channels in model.

[13] The attribute to identify which roles interact with which is simplified to keep the description of alternatives shorter. Ideally, roles would have identifiers, and subscribers would point to publishers' identifiers. In any case, this is considered a poor style design decision.

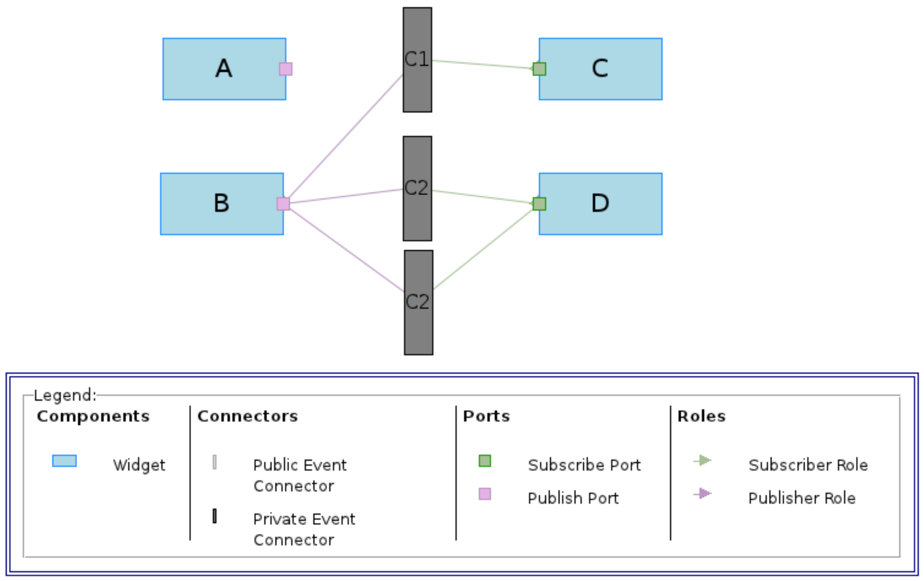


Figure 6: Model of sample system in style alternative 2

4.3.3 Style Alternative 3: Event Buses

Decisions: RC3, RCT1, RR4, MIC1, DPR2.

This style variant attempts to fight the proliferation of connectors as number of channels grows. It is stated that architectural model always has two connectors: one for public domain of communications and the other one for private domain. To add at least information unit 3, we have to add an attribute to link subscriber roles to publisher roles; otherwise, it is impossible to find communication pathways in a diagram. To incorporate level 1, that attribute can be replaced by channel name for every role type. However, it make connectors significantly more complex.

A model of the sample system produced with this alternative is in Figure 7. Notes depict the said attribute of roles, analogous to the one in style 1.

Units of information: 1, 2, 4.

Advantages: Clean diagram due to fewer connectors.

Disadvantages: Overly complicated internal logic of connectors; difficult or impossible to see communication pathways.

4.3.4 Style Alternative 4: Subscribe Sets

Decisions: RC4, RCT3, RR4, MIC1, DPR2.

If we commit to RC4, architectural models of compositions have much less connectors than in case of RC1 or RC2 and do not need to have complicated roles or connectors. However, this style option has a significant drawback: it is impossible to devise what channels are used from such a model. For example, if a widget is publishing to two channels, both of them will be represented by one connector in the model. It means a style does not include information of unit 1 and disables a

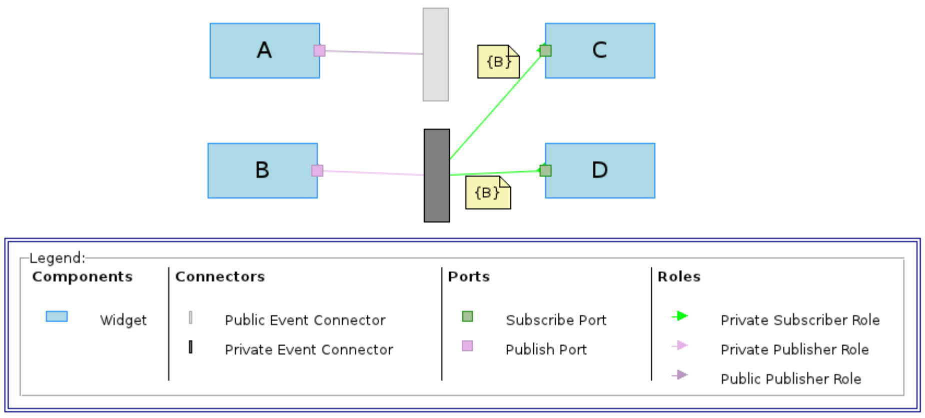


Figure 7: Model of sample system in style alternative 3

number of potential analyses. The result of modeling the sample system with this style is depicted in Figure 8.

Units of information: 1, 4.

Advantages: Relatively fewer connectors; diagrams are clearer.

Disadvantages: Does not include information unit 1; no public/private division is possible.

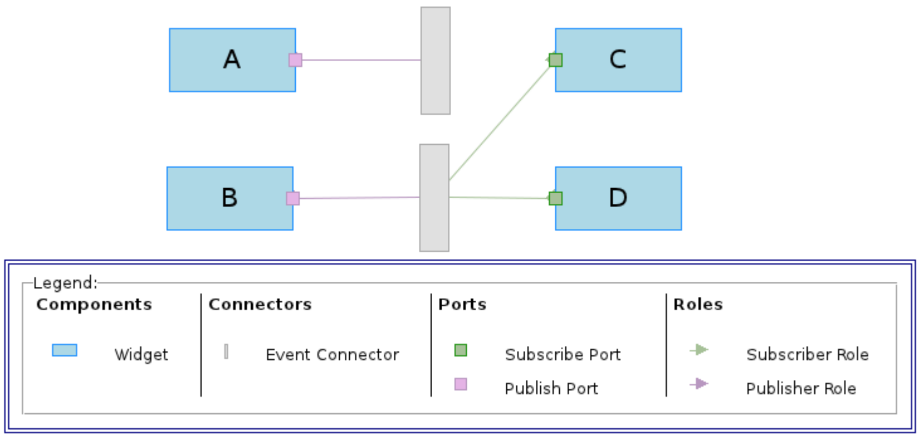


Figure 8: Model of sample system in style alternative 4

4.3.5 Style Alternative 5: Subchannels

Decisions: RC5, RCT2, RR4, MIC1, DPR1

As we can see from previous alternatives, there needs to be a compromise between complexity of connectors' logic, number of connectors on diagrams, and use of attributes. This style uses the following technique: every channel is viewed as one that has a public subchannel and potentially a number of private subchannels. The presence of private subchannels is determined by following criterion: if there is interaction happening between widgets with restriction lines, there is a private connector involved.

This style keeps the number of connectors low as the number of channels grows, and at the same time provide public/private differentiation. A model of the sample system is shown in Figure 9.

Units of information: 1, 2, 4.

Advantages: Relatively fewer connectors; diagrams are clearer.

Disadvantages: More connectors than in alternatives 3 and 4.

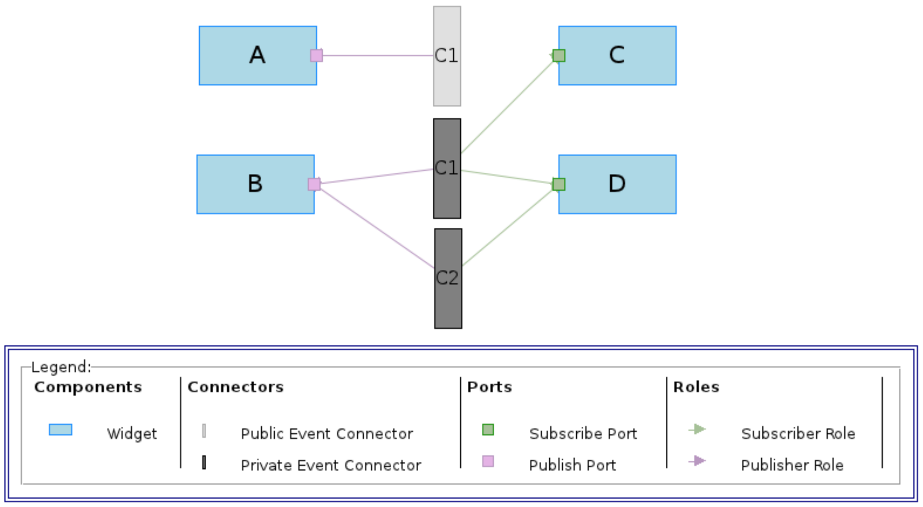


Figure 9: Model of sample system in style alternative 5

4.4 Final OWF Style

From the style alternatives [14], we picked the Style Alternative 5: Subchannels. The main reason for it is that it gives a good balance between style requirements (Section 3.2) as well as style design heuristics (Section 3.4). Below, we refer to the chosen alternative as the OWF style and describe it in details.

The OWF style is a specialization of a generic publish-subscribe style [3]. The latter contains two component types (publisher and subscriber), an event bus connector type, and port and role types for publishing and subscribing.

```
// This Acme family describes the style of Ozone Widget Framework
    compositions
Family OzoneFam extends PubSubFam with {
// ...
}
```

[14]We considered many more than five alternatives, and presented only the most suitable of them.

In Ozone user composition style, widgets are represented with $WidgetT$ - a component type that inherits both publishing and subscribing types.

```
    // A component type that represents widgets placed on the dashboard
    Component Type WidgetT extends PublisherCompT, SubscriberCompT with {

    // ...
    }
```

To elaborate more on how channels are mapped to connectors, we need to formally define what a *subchannel* is. Consider a general situation: a number of widgets are subscribed and publish to a channel; some of widgets are connected with restriction lines. We want to model it with the minimal number of connectors possible, assuming that the logic of connectors stays simple: every widget subscribed to it gets all events from each publisher. Here we come to several definitions:

Definition 1. A subchannel of an Ozone channel X is a set S of widgets satisfying three statements:

- Every widget in S is publishing or subscribing to channel X.
- For every subscribing widget A in S and for every publishing widget B in S, it is true that A receives events from B.
- Set S is maximal: no widget can be added to it without violating at least one of two previous statements.

Definition 2. A subchannel is called $active$ if it has at least one publisher and subscriber.

Definition 3. A subchannel is called $private$ if at least one of its widgets is attached to a restriction line. Otherwise, a subchannel is called $public$[15].

There can be several subchannels for one channel. For example, if two widgets A and B publish to a channel, widget C subscribes to the channel, and A is connected by a restriction line to C, this channel has two subchannels: $\{A, C\}$ (active, private) and $\{B\}$ (inactive, public).

Each connectors in this style represents a subchannel. The connector type $PublicChannelT$ represents public subchannels, and $PrivateChannelT$ represents private subchannels. Both of these connector types have an attribute that stores a corresponding channel's name, which is always known and should be assigned to this attribute. It is a modeling rule that every subchannel of an OWF composition is represented in a model with an appropriate connector.

```
Connector Type PublicChannelT extends EventBusConnT with {
    // Name of a channel that corresponds to the connector
    // This property should be unchangeable
  Property channelName : string;
}

// A private channel connector represents widgets' communication
// that happens through a restriction line ("private").
```

15From the definition of a subchannel, it follows that every widget of a private subchannel is attached to a restriction line, whereas every widget of a public subchannel is not attached to a restriction line.

```
Connector Type PrivateChannelT extends EventBusConnT with {

    // Name of a channel that corresponds to the connector
    // This property should be unchangeable
    Property channelName : string;
}
```

Ports types are inherited from the publish-subscribe style, and signify an intent of publishing or receiving messages. Roles of publisher and subscriber represent corresponding interaction responsibilities and can only be attached to respective ports.

The full code of OWF user composition Acme style is listed in Appendix B.

4.4.1 Data Loss Analysis

To exemplify how the style can be used for analysis, we implement a simple analysis carried out by Acme typechecker. This analysis searches all connectors in which data is lost, i.e. there are components publishing to them, but there are no components subscribed to them. According to how OWF works—no caching or logging messages is done—the information is lost in this situation, which might be of interest to user.

The implementation of data loss analysis consists of a function *dataLossEventBuses* that returns a set of connectors where data is lost and a rule demanding that this set be empty. If an Acme Studio user has a composition with such a connector, he will be notified of this rule's violation. You can see an application of data loss analysis on Figure 10: a connector is marked with an exclamation sign because it violates the rule *noDataLoss*.

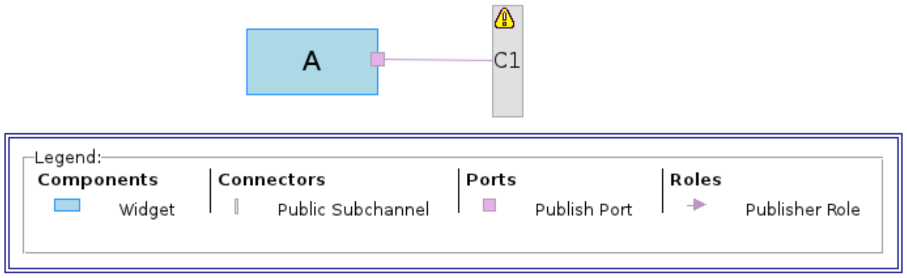


Figure 10: Analysis finds a data loss connector

```
    // Analysis: returns the connectors that are losing data
    // (i.e. some components publish to them, but none subscribe to)
    // in a given system.

    analysis dataLossEventBuses(s: System):set{EventBusConnT} =
    {select eb: EventBusConnT in s.CONNECTORS |
    (exists pub:PublisherRoleT in eb.ROLES | true) and
    (!exists sub:SubscriberRoleT in eb.ROLES | true)};

    // Rule: there should be no data loss in any system.
    rule noDataLoss = heuristic size(dataLossEventBuses(self)) == 0;
```

5 Future Research Directions

The style presented in this report is a basic architectural formalization that enables structural analysis of widget compositions. There is opportunity for creating more specific models of OWF and respective analyses. These specific models would be represented by architectural styles that inherit the OWF style and extend it by adding new properties, or even component or connector types. This additional information would be used by new analyses that are made possible by it. It is reasonable for the extension of this style to be driven by user studies or user reports on interaction problems. The authors did not attempt such studies and had to speculate on what end-user problems might occur. A list of possible analyses has already been given in Section 4, summarizing extensions that might be helpful for OWF users.

Delivering analysis results to end users in a convenient way is a practical task yet to be solved. Building a runtime model of a widget dashboard requires obtaining up-to-date information about present widgets and their publishing and subscribing relations. A major implementation issue is that it is generally unknown to Ozone framework which channels a widget *will* publish to. This fact limits predictive and preventive analysis of widget composition. One possible solution is up-front static analysis of widget source code, but it lacks generality.

OWF and many other event-based frameworks for end-user widget composition remain unexplored from the formal architectural perspective. Devising common architectural styles would help generalize design decisions to the whole area of such frameworks. Also, if several frameworks share an architectural style, formal analyses of one of them can be applied to another. Another possible research direction is the principled design of such frameworks. Building a taxonomy of major design decisions, alternative options, and their impacts for event-based visual integration environments would facilitate the creation of such environments with respect to required quality attributes. There is a big body of potential work in this area: there is no agreement on the set of quality requirements relevant to such systems and on their priorities [13]. And, of course, the decisions themselves are yet to be gathered and categorized.

5.1 Conclusion

In this report, end-user compositions of Ozone Widget Framework were modeled with Acme architecture description language. We described the creation of style in detail: first, requirements for the style were formulated; then style design decision points and associated tradeoffs were examined; finally, several style alternatives and their positive and negative sides were reviewed, and the final style was picked. We have demonstrated that the style can serve as a basis for analysis by creating an analysis that finds channels through which data is lost.

References

[1] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4:319–364, October 1995.

[2] Margaret Burnett, Curtis Cook, and Gregg Rothermel. End-user software engineering. *Commun. ACM*, 47:53–58, September 2004.

[3] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2 edition, October 2010.

[4] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–26, Washington, DC, USA, 1996. IEEE Computer Society.

[5] Jurgen Dingel, David Garlan, Somesh Jha, and David Notkin. Reasoning about implicit invocation. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998.

[6] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[7] David Garlan, Vishal Dwivedi, Ivan Ruchkin, and Bradley Schmerl. Foundations and tools for end-user architecting. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, pages 157–182, Oxford, UK, 2012. Springer.

[8] David Garlan, Robert T. Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON*, pages 7–22, 1997.

[9] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *Proc. of the 6th European Software Engineering Conference*, ESEC, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[10] Next Century Corporation. Ozone widget framework. http://owfgoss.org, 2014.

[11] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2 sub edition, June 1992.

[12] Xiaotao Wu and V. Krishnaswamy. Widgetizing communication services. In *2010 IEEE International Conference on Communications (ICC)*, pages 1–5. IEEE, May 2010.

[13] Zhenzhen Zhao, Sirsha Bhattarai, Ji Liu, and Noel Crespi. Mashup services to daily activities: end-user perspective in designing a consumer mashups. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, pages 222–229, New York, NY, USA, 2011. ACM.

A Z Model of Ozone Compositions

This appendix provides a model of OWF compositions in Z specification languages for clarification purposes. This model supplements the description of Ozone in Section 2.

A.1 Context of Z Modeling

The main purpose of the Z model [11] of Ozone compositions is to unambiguously convey how they work; our approach is to stay as close to the actual OWF messaging as possible (as long as it is simple). Unfortunately, it would hinder the generality of this model, but also it would provide deeper insights into how exactly Ozone widgets communicate.

The following main decisions for this Z model have been made:

- Only the details that are relevant to messaging and restrictions is included into the model.
- Data about current widgets, channels, and subscriptions is stored on the top level of the configuration. This decision is made to simplify constraints and pre-post conditions.
- Channels are created on-the-fly when they are accessed. After that, they stay in the model infinitely.
- Closed world assumption: all widgets that are currently in the system are known. Adding new widgets to a system is done through an operation.

A.2 Event Exchange Model

First, let us specify event dispatch of OWF in Z. By doing so, we will create a platform that can be used to model restriction lines and analyses. This decision will let us keep schemas simpler and compose them later as opposed to throwing all details in at once.

Widgets and channels are represented as basic types $Widget$ and $Channel$ (not as schemas) to make the specification more compact: putting data about present widgets, channels, and publish-subscribe relations between them into a single schema makes it possible to describe effects of publishing and subscribing with one schema, without using promotion. In the real world, of course, widgets have a rich set of attributes. These attributes might include name, URL, implementation technology, and unique ID, but none of them is relevant to event exchange.

$$[Widget, Channel]$$

The $Configuration$ schema represents an instance of a dashboard with particular widgets and channels on it. The $subscription$ partial function from widgets to sets of channels is included in this schema to keep track of which widgets are subscribed to which channels. However, publishing to a channel is not modeled as a function since it is an instant action in OWF and does not change the state of the system, except for maybe creating a channel. Thus, since channels are tracked separately in $Configuration$, there is no need for publishing as a relation in this schema.

The state invariant for the $\mathit{Configuration}$ schema ensures that only widgets and channels registered in a configuration may participate in the $\mathit{subscription}$ relation. Widgets that have not subscribed to any channel are mapped to an empty set.

$$\begin{array}{|l}
\textit{Configuration} \\
\hline
\mathit{widgets} : \mathbb{P}\,\mathit{Widget} \\
\mathit{channels} : \mathbb{P}\,\mathit{Channel} \\
\mathit{subscription} : \mathit{Widget} \nrightarrow \mathbb{P}\,\mathit{Channel} \\
\hline
\operatorname{dom} \mathit{subscription} = \mathit{widgets} \\
\bigcup(\operatorname{ran} \mathit{subscription}) \subseteq \mathit{channels} \\
\hline
\end{array}$$

In the initial state of the $\mathit{configuration}$ schema, there are no widgets or channels.

$$\begin{array}{|l}
\textit{ConfigurationInit} \\
\hline
\mathit{Configuration} \\
\hline
\mathit{widgets} = \emptyset \\
\mathit{channels} = \emptyset \\
\mathit{subscription} = \emptyset \\
\hline
\end{array}$$

Let's start describing operations on configurations with adding and removing widgets. These operations do not change the set of channels.

The $\mathit{AddWidget}$ operation takes a widget that is not yet present the configuration and adds it. Since it is required that the domain of $\mathit{subscription}$ contains every element of $\mathit{widgets}$, we have to explicitly declare that the new widget is mapped to an empty set.

$$\begin{array}{|l}
\textit{AddWidget} \\
\hline
\Delta \mathit{Configuration} \\
w? : \mathit{Widget} \\
\hline
w? \notin \mathit{widgets} \\
\mathit{widgets}' = \mathit{widgets} \cup \{w?\} \\
\mathit{channels}' = \mathit{channels} \\
\mathit{subscription}' = \mathit{subscription} \cup \{w? \mapsto \emptyset\} \\
\hline
\end{array}$$

The operation of removing a widget is analogous to the one of adding a widget: it removes a given widget from the set $\mathit{widgets}$. Additionally, the $\mathit{subsription}$ function removes the deleted widget from its domain to satisfy the state invariant.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemoveWidget}} \\
\Delta \mathit{Configuration} \\
w? : \mathit{Widget} \\
\hline
w? \in \mathit{widgets} \\
\mathit{widgets}' = \mathit{widgets} \setminus \{w?\} \\
\mathit{channels}' = \mathit{channels} \\
\mathit{subscription}' = \{w?\} \lhd \mathit{subscription} \\
\hline
\end{array}
$$

Now, we can look at operations that change the *subscription* function: subscribe and unsubscribe. As noted before, channels are created on-the-fly, whenever they are published and subscribed to. In terms of our model, this fact means that there are no operations that directly add or remove a channel from *channels*. Addition of channels is indirectly managed by *Subscribe* and *Unsubscribe*: a referenced channel is set up if it doesn't exist. Nevertheless, the main purpose of *Subscribe* is to record the fact that a certain widget now wants to receive messages from a certain channel. It is required that this widget isn't subscribed to this channel at the moment when this operation is executed. After it is executed, the set of subscriptions for the widget is increased by the channel.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Subscribe}} \\
\Delta \mathit{Configuration} \\
w? : \mathit{Widget} \\
ch? : \mathit{Channel} \\
\hline
w? \in \mathit{widgets} \\
ch? \notin \mathit{subscription}(w?) \\
\mathit{widgets}' = \mathit{widgets} \\
\mathit{channels}' = \mathit{channels} \cup \{ch?\} \\
\mathit{subscription}' = \mathit{subscription} \oplus \{w? \mapsto (\mathit{subscription}(w?) \cup \{ch?\})\} \\
\hline
\end{array}
$$

The *Unsubscribe* function removes a channel from a widget's subscription set. Note that it does not delete a channel. That is, after having been added, a channel stays in the configuration forever. This modeling path has been chosen according to the assumption about channels' existence that was stated in Section 1. Another way to decide on the presence of channels could be to delete them once they are not listened to, but this approach just complicates postconditions with no benefits.

$$
\begin{array}{l}
\textit{Unsubscribe} \\
\hline
\Delta \textit{Configuration} \\
w? : \textit{Widget} \\
ch? : \textit{Channel} \\
\hline
w? \in \textit{widgets} \\
ch? \in \textit{subscription}(w?) \\
\textit{widgets}' = \textit{widgets} \\
\textit{channels}' = \textit{channels} \cup \{ch?\} \\
\textit{subscription}' = \textit{subscription} \oplus \{w? \mapsto (\textit{subscription}(w?) \setminus \{ch?\})\} \\
\hline
\end{array}
$$

The operation of publishing represents sending a message to a channel by some widget. Publishing doesn't affect the state of the user's orchestration, except for adding a channel if it hasn't been used before. This operation, however, has an important aspect to model — the set of widgets that receive the sent message. This set is represented by the $\textit{receivers}!$ variable. Until we consider the concept of restriction line, $\textit{receivers}!$ is simply the set of widgets that have subscribed to the channel to which a message is published.

$$
\begin{array}{l}
\textit{Publish} \\
\hline
\Delta \textit{Configuration} \\
w? : \textit{Widget} \\
ch? : \textit{Channel} \\
\textit{receivers}! : \mathbb{P}\ \textit{Widget} \\
\hline
w? \in \textit{widgets} \\
\textit{widgets}' = \textit{widgets} \\
\textit{channels}' = \textit{channels} \cup \{ch?\} \\
\textit{subscription}' = \textit{subscription} \\
\textit{receivers}! = \{w : \textit{widgets} \mid \{ch?\} \subseteq \textit{subscription}(w)\} \\
\hline
\end{array}
$$

This concludes the formalization of basic messaging in the Ozone Widget Framework.

A.3 Event Restriction Model

The messaging model is now ready to be enhanced with the interesting way of controlling communication - the restriction line. We model it as a binary symmetric relation. This decision is made to reflect directly what's going on in the real system.

An abbreviation for relations over widgets is introduced for brevity.

$$\textit{WidgetRel} == \textit{Widget} \leftrightarrow \textit{Widget}$$

As described in Section 1, drawing restriction lines between widgets defines a relation over them. This relation has two properties that follow from how OWF is implemented:

1. A restriction line cannot connect a widget to itself. In terms of relations, there is no pair where the first element is equal to the second element.
2. A restriction line is symmetric (has no direction, or is bidirectional): if it connects widget A to widget B, it also connects widget B to widget A. Thus, the relation should be symmetric (equal to its inverse).

Note: a restriction line is not transitive. That is, if widget A is connected to widget B, which is connected to widget C, widgets A and C are not allowed to communicate unless there is a direct restriction line between A and C.

We use a helper schema to check whether a given relation over widgets can be defined by some combination of restriction lines. The schema variable $allRestrictionLines$ represents an infinite set of widget relations that conform to a definition of restriction line. A relation can be described by a layout of restriction lines if and only if the set $allRestrictionLines$ contains it.

$$\begin{array}{|l}
\textit{RestrictionLineDefinition} \\
\hline
allRestrictionLines : \mathbb{P}\ WidgetRel \\
\hline
allRestrictionLines = \{r : WidgetRel \mid \\
\quad (\forall\, w1, w2 : Widget \bullet (w1, w2) \in r \Rightarrow w1 \neq w2) \land \\
\quad (r = r^{\sim})\} \\
\hline
\end{array}$$

Using the $RestrictionLineDefinition$ schema, it is possible to define the OWF configuration that keeps track not only of widgets, channels, and subscriptions, but also of restriction lines. The relation $restriction$ will store the current setup of restriction lines in the composition. This relation is required to conform to the definition of restriction lines that has been presented above. Moreover, it is necessary that only the widgets that are present in the configuration constitute this relation, so we require the domain of $restriction$ be a subset of $widgets$. From this fact and from the symmetry of $restriction$ it follows that the range of $restriction$ is also a subset of $widgets$.

$$\begin{array}{|l}
\textit{ConfigurationWithRestriction} \\
\hline
Configuration \\
RestrictionLineDefinition \\
restriction : WidgetRel \\
\hline
\operatorname{dom} restriction \subseteq widgets \\
restriction \in allRestrictionLines \\
\hline
\end{array}$$

The only plausible value for the $restriction$ relation in the initial configuration is an empty set because there are no widgets present.

$$\begin{array}{|l}
\textit{ConfigurationWithRestrictionInit} \\
\hline
ConfigurationWithRestriction \\
ConfigurationInit \\
\hline
restriction = \emptyset \\
\hline
\end{array}$$

Two framing schemas are defined below: $\mathit{RetainMessagingFrame}$ and $\mathit{RetainRestrictionsFrame}$. The former declares that the basic messaging part (widgets, channels, subscriptions) is not changed, while the latter ensures that eventing restrictions don't change.

$$
\begin{array}{|l}
\hline
\mathit{RetainMessagingFrame} \\
\Delta \mathit{ConfigurationWithRestriction} \\
\hline
\mathit{widgets}' = \mathit{widgets} \\
\mathit{channels}' = \mathit{channels} \\
\mathit{subscription}' = \mathit{subscription} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{RetainRestrictionsFrame} \\
\Delta \mathit{ConfigurationWithRestriction} \\
\hline
\mathit{restriction}' = \mathit{restriction} \\
\hline
\end{array}
$$

$\mathit{AddRestriction}$ models drawing a restriction line between two different widgets $w1?$ and $w2?$ provided it hasn't been there. In order to modify the $\mathit{restriction}$ relation corretly, both pairs $(w1?, w2?)$ and $(w2?, w1?)$ are added. Otherwise, the symmetry requirement is violated.

$$
\begin{array}{|l}
\hline
\mathit{AddRestriction} \\
\mathit{RetainMessagingFrame} \\
w1?, w2? : \mathit{Widget} \\
\hline
w1? \neq w2? \\
(w1?, w2?) \notin \mathit{restriction} \\
\mathit{restriction}' = \mathit{restriction} \cup \{(w1?, w2?), (w2?, w1?)\} \\
\hline
\end{array}
$$

$\mathit{RemoveRestriction}$ is the opposite of $\mathit{AddRestriction}$: a line between two widgets is deleted, which results into removing two pairs from the relation $\mathit{restriction}$.

$$
\begin{array}{|l}
\hline
\mathit{RemoveRestriction} \\
\mathit{RetainMessagingFrame} \\
w1?, w2? : \mathit{Widget} \\
\hline
(w1?, w2?) \in \mathit{restriction} \\
\mathit{restriction}' = \mathit{restriction} \setminus \{(w1?, w2?), (w2?, w1?)\} \\
\hline
\end{array}
$$

We need to adapt the operations over $\mathit{Configuration}$ to $\mathit{ConfigurationWithRestriction}$. The operations of adding a widget, subscribing, and unsubscribing can be promoted to the restriction level by being conjoined with $\mathit{RetainRestrictionsFrame}$ because they don't modify the layout of eventing lines. The operations promoted to the level with restrictions get a -Rest suffix.

$AddWidgetRest \mathrel{\widehat=} AddWidget \land RetainRestrictionsFrame$
$SubscribeRest \mathrel{\widehat=} Subscribe \land RetainRestrictionsFrame$
$UnsubscribeRest \mathrel{\widehat=} Unsubscribe \land RetainRestrictionsFrame$

Promoting $RemoveWidget$ needs more attention because it changes the messaging restrictions: if the deleted widget had restriction connections, these connections should be destroyed. After translating into the language of the $restriction$ relation, this destruction is equivalent to removing the widget from both the domain and the range of this relation.

```
RemoveWidgetRest
  RemoveWidget
  $\Delta ConfigurationWithRestriction$
  ----
  $restriction' = \{w?\} \lhd (restriction \rhd \{w?\})$
```

Adapting the $Publish$ operation is even more complicated because the output parameter is modified in a non-trivial way by emergent restrictions. The approach is as follows: we take the original operation, rename the output parameter $receivers!$ to $receivers?$, combine it with $RetainRestrictionsFrame$ (because restrictions don't change), and declare a new output parameter $receivers!$. All constraints on $widgets$, $channels$, and $subscription$ are carried over from $Publish$.

How can the set of receivers be updated according to the restrictions? Certain widgets that are prohibited from communicating with $w?$ should be dropped from $receivers?$. This fact turns us to the definition of the restriction line. That is, two distinct widgets are allowed to communicate if and only if they are directly connected with a line or when none of them is connected to any other widget. So in case of one widget $w?$, we start by checking whether it's connected to any other widget $w1$. If yes, then we drop all widgets $w2$ that don't have a direct restriction connection to $w?$ from $receivers?$. If no, we drop all widgets $w3$ that are connected to some other widget $w4$. In the latter case, it is guaranteed that $w4$ is equal neither to $w?$ (because of false statement under if) nor to $w3$ (because of the property of $restriction$).

We shouldn't miss out on one small technical detail: whatever layout of restriction lines there is, a widget will always receive its own messages. However, the if-else expression can remove $w?$ out of $receivers?$. Hence, we explicitly add it to the resulting set of a message receivers.

```
PublishRest
  $Publish[receivers?/receivers!]$
  $RetainRestrictionsFrame$
  $receivers! : \mathbb{P}\ Widget$
  ----
  $receivers! = receivers? \setminus$
      $(\textbf{if } \exists w1 : widgets \bullet (w?, w1) \in restriction$
          $\textbf{then } \{w2 : receivers? \mid (w?, w2) \notin restriction\}$
          $\textbf{else } \{w3 : receivers? \mid \exists w4 : widgets \bullet (w3, w4) \in restriction\})$
      $\cup \{w?\}$
```

This turned out to be a complex and hard to understand invariant. This incomprehensibility is not caused by the modeling language or approach itself — it is brought about by the real OWF implementation. It is intrinsic for the logic behind this restriction line in Ozone. It is difficult to reason about how the operation $\mathit{PublishRest}$ works, and it is even more difficult to reason about the real user orchestration without having a formula to work with.

A.4 Simplifying Restriction

In the remainder of 4 a straightforward simplification of the restriction line is presented. It aims at modeling the eventing restriction through a simple way of creating a “whitelist” for the exchange of events. This approach wraps the complicated concept of restriction line with a simpler concept of permitted communications.

So the function introduced below transforms a restriction relation into a permission relation that expresses the same set of constraints on messaging. The definition of a permission relation is that it contains a pair of widgets if and only if these two widgets are permitted to communicate (through any channel). When can this happen? According to the definition of how restriction lines work, it happens only in three cases:

- Two widgets are form a pair in $\mathit{restriction}$
- Two widgets are not in any pair in $\mathit{restriction}$
- Two widgets are the same widget

$$\begin{array}{|l}
\mathit{restrictionToPermission} : \mathit{WidgetRel} \pfun \mathit{WidgetRel} \\
\mathit{RestrictionLineDefinition} \\
\hline
\operatorname{dom} \mathit{restrictionToPermission} = \{ r : \mathit{WidgetRel} \mid r \in \mathit{allRestrictionLines} \} \\
\forall r : \mathit{WidgetRel} \bullet \mathit{restrictionToPermission}(r) = \\
\quad \{ w1, w2 : \operatorname{dom} r \mid (w1, w2) \in r \lor \\
\quad\quad (\forall w3 : \mathit{Widget} \bullet (w1, w3) \notin r \land (w2, w3) \notin r) \lor \\
\quad\quad (w1 = w2) \}
\end{array}$$

Having defined this $\mathit{restrictionToPermission}$ function, we can now rewrite the publishing operation in a more readable way. We simply demand that all widgets w prohibited from talking to $w?$ (according to the permission relation) be removed from the set of receivers.

$$\begin{array}{|l}
\hline
\mathit{PublishRestSimple} \\
\mathit{Publish}[\mathit{receivers}?/\mathit{receivers}!] \\
\mathit{RetainRestrictionsFrame} \\
\mathit{receivers}! : \mathbb{P}\ \mathit{Widget} \\
\hline
\mathit{receivers}! = \mathit{receivers}? \setminus \\
\quad \{ w : \mathit{Widget} \mid (w?, w) \notin \mathit{restrictionToPermission}(\mathit{restriction}) \} \\
\hline
\end{array}$$

This concludes the description of OWF in Z specification language.

B Ozone Acme Style Source

Below you find the source code of OWF compositions architectural style in the Acme language. It inherits the publish-subscribe style and specifies rules for correct Ozone compositions.

```
import $AS_GLOBAL_PATH/families/PubSubFam.acme;

// This Acme family describes the style of Ozone Widget Framework compositions
Family OzoneFam extends PubSubFam with {

    // A component type that represents widgets placed on the dashboard
    Component Type WidgetT extends PublisherCompT, SubscriberCompT with {
        // Rule: a component that does not communicate might be alarming
        rule OrphanedComponent = heuristic size(self.PORTS) > 0;
    }

    // A public channel connector represents widgets' communication in ``
       public domain'', i.e. not constrained by restriction lines.
    Connector Type PublicSubchannelT extends EventBusConnT with {
        // Name of the corresponding OWF channel
        // This property should be unchangeable
        Property channelName : string;

        // Rule: all roles are of types declared in PubSub
        rule OnlyPubSubRoles = invariant forall r in self.ROLES | exists t in
            {PublisherRoleT, SubscriberRoleT} |
             declaresType(r, t);
    }

    // A private channel connector represents widgets' communication that
       happens through a restriction line (``private'').
    Connector Type PrivateSubchannelT extends EventBusConnT with {

        // Name of a channel that corresponds to the connector (not one-to-one
            )
        // This property should be unchangeable
        Property channelName : string;

        // Rule: all roles are of types declared in PubSub
        rule OnlyPubSubRoles = invariant forall r in self.ROLES | exists t in
            {PublisherRoleT, SubscriberRoleT} |
             declaresType(r, t);

        // Rule: a private subchannel shall have at least one role
        rule AtLeastOneRole = invariant size(self.ROLES) > 0;

    }

    // Rule: a channel cannot have more than two public connectors
       representing it.
    rule UniquePublicChannelInv = invariant forall c1,c2 : PublicSubchannelT
       in self.CONNECTORS |
        declaresType(c1, PublicSubchannelT) and declaresType(c1,
            PublicSubchannelT) and (c1 != c2)
       -> (c1.channelName != c2.channelName);

    // Rule: (property of restriction lines) if a channel is talking on a
       private channel, it cannot communicate over a public one.
    // The fact ``talking on a public -> cannot talk on private'' can be
       inferred from this rule.
    rule ConnectOnlyToPublicOrPrivateDomainInv = invariant forall w : WidgetT
       in self.COMPONENTS, c_pub : PublicSubchannelT in self.CONNECTORS,
       c_priv : PrivateSubchannelT in self.CONNECTORS |
        !attached(c_priv, w) or !attached(c_pub, w);

    // Analysis: returns the connectors that are losing data (i.e. some
       components publish to them, but none subscribe to) in a given system.
    analysis dataLossEventBuses(s: System):set{EventBusConnT} = {select eb:
       EventBusConnT in s.CONNECTORS | (exists pub:PublisherRoleT in eb.ROLES
        | true) and (!exists sub:SubscriberRoleT in eb.ROLES | true)};
    // Rule: there should be no data loss in any system.
    rule noDataLoss = heuristic size(dataLossEventBuses(self)) == 0;

    // Rule: usually, users don't want to have an empty public connector
       representing a channel if we have a private one
    // Because the information of this channel's existence is already captured
        by the private connector
    rule NoEmptyPublicIfHavePrivate = heuristic forall c_pub :
       PublicSubchannelT in self.CONNECTORS, c_priv : PrivateSubchannelT in
       self.CONNECTORS |
        c_pub.channelName == c_priv.channelName -> size(c_pub.ROLES) >= 1;
}
```