

Formal Definition of the Chiron-2 Software Architectural Style

Nenad Medvidovic

UCI-ICS Technical Report 95-24
Department of Information and Computer Science
University of California, Irvine, CA 92717-3425

`neno@ics.uci.edu`

April 19, 1996

Abstract

The Chiron-2 style is a new software architectural style designed to support the particular needs of applications with a graphical user interface aspect. Several experimental systems have been built to demonstrate its intended goals. The conceptual architectures of those systems were depicted by “boxes and arrows.” This notation is too ambiguous and a more formal definition is needed.

This paper presents a formal specification of the C2 style. In particular, C2 components and connectors, their rules of composition, and communication between them are defined. This definition is evaluated with a set of independently devised requirements. Its utility and future uses are discussed.

1 Introduction

Software architectural styles are key design idioms. Unix’s pipe-and-filter style is more than twenty years old; blackboard architectures have long been common in AI applications. User interface software has typically made use of two run-time architectures: the client-server style and the call-back model. Also well known is the model-view-controller (MVC) style [KP88], commonly used in Smalltalk applications. The Arch style is more recent, and has an associated meta-model [Wor92].

Chiron-2, or C2, style [TMA+95] is a new architectural style that drew its key ideas from many sources, including the styles mentioned above, as well as specific experience with the Chiron-1 system [TJ93]. The C2 style is designed to support the particular needs of applications that have a graphical user interface aspect, but it clearly has the potential for supporting other types of applications as well. A key motivating factor behind the development of the C2 style is the emerging need, in the user interface world, for a more component-based development economy. User interface software frequently accounts for a very large fraction of application software, yet reuse in the UI domain is typically limited to toolkit (widget) code. The architectural style presented supports a paradigm in which UI components, such as dialogs, structured graphics models (of

various levels of abstraction), and constraint managers, can more readily be reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active (and described in different formalisms), multiple media types may be involved, and multiple user tasks (“processes”) supported.

We have not yet demonstrated that all these goals are achievable or especially supported by this style. However, we have examined several key properties and built several diverse experimental systems, and believe that our preliminary findings are encouraging and that the style has substantial utility “as is.” At the same time, we recognize that much additional work on the style and supporting tools is needed. One of the facets that must be studied further is formal representation of the style. The notation used to model individual C2 architectures thus far has been graphical (“boxes and arrows”). While its expressive power appears to be sufficient for this purpose, the notation has introduced ambiguities in interpreting architectures. Furthermore, its inherent imprecision makes the kinds of formal analyses of architectures that we would want to perform in the proposed C2 design environment [RWMT95] improbable. Therefore, there clearly exists a need for a formal definition of the C2 style. By producing such a definition, while retaining the existing graphical notation, we intend to preserve the representational simplicity of architectures and the potential for their direct manipulation, but also add the ability to formally reason about them.

The paper is organized as follows: Section 2 discusses a set of requirements that the C2 formalism should satisfy. These requirements were determined independently by the Chiron-2 Formalism Workgroup at University of California, Irvine. Section 3 presents an overview of the C2 style and Section 4 formal definitions of its major features. Section 5 discusses the results of the formalism. The conclusion is given in Section 6, while the entire formal definition of C2 is provided in the Appendix.

2 Chiron-2 Formalism Requirements

The formalism of the C2 architectural style has the following requirements:

1. The formalism must be a standard for all C2 work.
2. The formalism must easily communicate architectures:
 - the formalism should be easy to learn, and
 - once the formalism is learned, architectures expressed in it should be easy to understand.
3. It must be suitable for analysis of architectures, such as automatic critiquing, manual analysis, concurrency analysis, and performance analysis.
4. It must be suitable for testing architectures specified with the formalism, with an emphasis on testing their concurrent properties.
5. It must be capable of being input to a generation process.
6. Both an ASCII linearization and direct graphical manipulation must be possible for architectures specified with the formalism.

7. The formalism must be capable of extension.
8. It must be capable of expressing dynamic nature of architectures (components and connectors may be added and removed at run-time).
9. It must be capable of expressing certain implementation characteristics of components, such as operating system, host, process, task, and address space.

3 Chiron-2 Overview

The C2 style can be informally summarized as a network of concurrent components hooked together by connectors, i.e., message routing devices. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a single connector. Note that when two connectors are attached to each other, it must be from the bottom of one to the top of the other (see Figure 1).

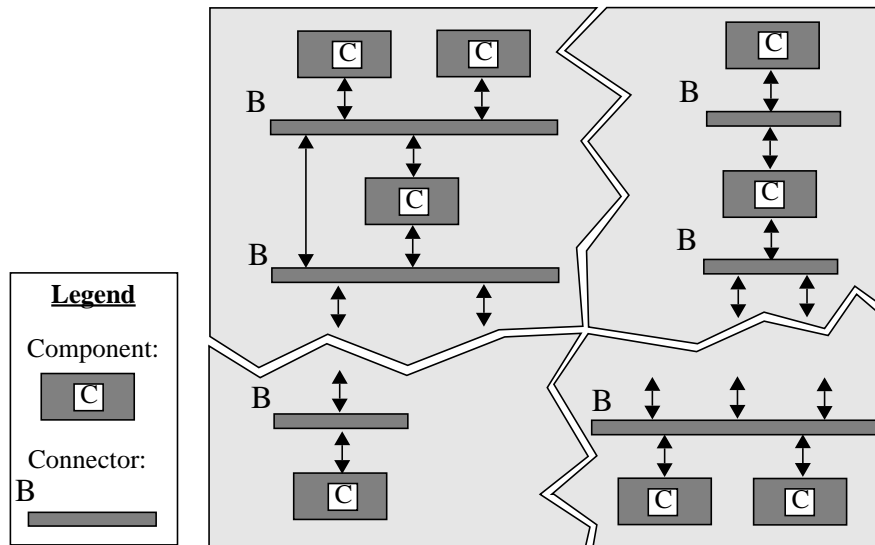


Figure 1: A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

Each component has a top and bottom domain. The top domain specifies the set of notifications to which a component responds, and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds.

All communication between components is solely achieved by exchanging messages. This requirement is suggested by the asynchronous nature of component-based architectures, and, in particular, of applications that have a GUI aspect, where both users and the application perform actions concurrently and at arbitrary times and where various components in the architecture must be notified of those actions. Message-based communication is extensively used in distributed environments for which this architectural style is suited.

Central to the architectural style is a principle of limited visibility or *substrate independence*: a component within the hierarchy can only be aware of components "above" it, i.e., components typically closer to the

”application,” and thus further from, e.g., the windowing system. Components are completely unaware of the components - including toolkits - which reside ”beneath” them.

Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. One issue that must be addressed, however, is the apparent dependence of a given component on its ”superstrate,” i.e., the components above it. If each component is built so that its top domain closely corresponds to the bottom domains of those components with which it is specifically intended to interact in the given architecture, its reusability value is greatly diminished and it can only be substituted by components with similarly constrained top domains. For that reason, the C2 style introduces the notion of event translation. For each component, a mapping from the messages it emits on its top side to those the components above it are capable of receiving on their bottom sides can be produced, as shown Figure 2. The C2 development environment is intended to provide support for accomplishing this task. The internal architecture of a component shown in Figure 2 is targeted to the user interface domain. While issues concerning composition of an architecture are independent of a component’s internal structure, for purposes of exposition below, this internal architecture is assumed.

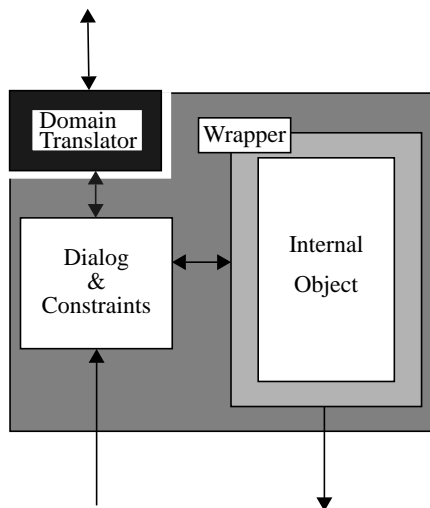


Figure 2: The internal architecture of a C2 component.

Each component may have its own thread(s) of control, a property also suggested by the asynchronous nature of tasks in the GUI domain. It simplifies modeling and programming of multi-component, multi-user, and concurrent applications and enables exploitation of distributed platforms. Note that separating components into different threads of control is not a requirement. Moreover, a proposed conceptual architecture is distinct from an implementation architecture, so that it is indeed possible for components to share threads of control.

Finally, there is no assumption of a shared address space among components. Any premise of a shared address space in an architectural style that allows composition of heterogeneous, highly distributed components, developed in different languages, with their own threads of control, internal structures, and domains of discourse, would be unreasonable.

4 Formal Definition of Several Chiron-2 Features

The formal definition of C2 is given in the Z notation [Spi89]. Z is a specification language based on mathematical constructs and it was chosen because of its promise of automatic translation for quick prototypes and its lack of bias toward any particular implementation language.

A C2 component is defined as follows:

<p><i>C2 Component</i></p> <p><i>name</i> : <i>COMP_NAME</i></p> <p><i>top_port</i>, <i>bot_port</i> : <i>COMM_PORT</i></p> <p><i>top_in</i>, <i>top_out</i>, <i>bot_in</i>, <i>bot_out</i> : $COMM_PORT \sim \text{ff } COMM_MSG$</p> <p><i>top_domain</i>, <i>bot_domain</i> : $\text{ff } COMM_MSG$</p> <p><i>dialog_in</i> : $COMM_MSG \sim OBJ_STATE$</p> <p><i>wrapper</i>, <i>dialog_top_out</i> : $OBJ_STATE \sim COMM_MSG$</p> <p><i>msg_to_handle</i> : <i>NEXT_MSG</i>;</p> <p><i>domain_trans</i> : $COMM_MSG \sim COMM_MSG$</p> <p><i>internal_states</i> : $\text{ff } OBJ_STATE$</p> <p><i>start_state</i> : <i>OBJ_STATE</i></p> <p><i>state_transitions</i> : $(OBJ_STATE \times (COMM_PORT \sim \text{ff } COMM_MSG)) \rightarrow$ $(OBJ_STATE \times \{ (COMM_PORT \sim \text{ff } COMM_MSG),$ $(COMM_PORT \sim \text{ff } COMM_MSG) \})$</p> <hr/> <p><i>top_port</i> \neq <i>bot_port</i></p> <p>$\text{dom } top_in = \{ top_port \}$ $\text{dom } top_out = \{ top_port \}$ $\text{dom } bot_in = \{ bot_port \}$ $\text{dom } bot_out = \{ bot_port \}$</p> <p><i>top_domain</i> = $top_in(top_port) \cup top_out(top_port)$ <i>bot_domain</i> = $bot_in(bot_port) \cup bot_out(bot_port)$</p> <p>$\forall state1, state2 : OBJ_STATE;$ $ps1, ps2, ps3 : COMM_PORT \sim \text{ff } COMM_MSG \bullet$ $((state1, ps1), (state2, \{ ps2, ps3 \})) \in state_transitions \Rightarrow$ $state1 \in internal_states$ $\wedge state2 \in internal_states$ $\wedge (\text{dom } ps1 = \{ top_port \} \vee \text{dom } ps1 = \{ bot_port \})$ $\wedge (\text{dom } ps2 = \{ top_port \})$ $\wedge (\text{dom } ps3 = \{ bot_port \})$ $\wedge (ps1(top_port) \subseteq top_in(top_port) \vee$ $ps1(bot_port) \subseteq bot_in(bot_port))$ $\wedge ps2(top_port) \subseteq top_out(top_port)$ $\wedge ps3(bot_port) \subseteq bot_out(bot_port)$</p>

This definition corresponds to the internal architecture of a component shown in Figure 2. Since a component's dialog can decide when and if to handle a particular message it receives at its top and bottom

ports [TMA+95], the *msg_to_handle* function is defined to select one of the messages at a port. A *transition* in a component is defined as processing a message received at either the top or the bottom port and possibly generating outgoing messages. For the sake of simplicity, this definition assumes that a message generated by a component may be a null message (i.e., it is not a requirement that a component produce two outgoing messages for each incoming message it processes). Note that all definitions involving components assume that they are internal components, i.e., they are neither top- nor bottom-most in an architecture. However, the top- and bottom-most components are easily described as special cases of the given definitions by omitting from schemas references to their sides, top or bottom, that are outermost in an architecture.

The following schema expresses substrate independence. A component must utilize the domain translator for the messages it both receives and sends on its top side. At the same time, it has no knowledge and makes no assumptions about its substrate, so that the wrapper around the internal object emits messages in component's own domain of discourse on its bottom side, unbeknownst to the dialog. See the Appendix for the definition of processing messages a component receives on its bottom side.

<p><i>HandleMessageFromAbove</i></p> <hr/> <p>$\Delta C2ComponentState$</p> <hr/> <p>$comp' = comp$</p> <p>$((current_state, top_in_data),$ $(current_state', \{ top_out_data', bot_out_data' \})) \in$ $comp.state_transitions$</p> <p>$top_out_data'(comp.top_port) =$ $top_out_data(comp.top_port) \cup$ $\{ comp.domain_trans($ $comp.dialog_top_out($ $comp.dialog_in($ $comp.domain_trans($ $comp.msg_to_handle($ $top_in_data(comp.top_port)))))) \}$</p> <p>$bot_out_data'(comp.bot_port) =$ $bot_out_data(comp.bot_port) \cup$ $\{ comp.wrapper($ $comp.dialog_in($ $comp.domain_trans($ $comp.msg_to_handle($ $top_in_data(comp.top_port)))) \}$</p> <p>$top_in_data(comp.top_port) =$ $top_in_data'(comp.top_port) \cup$ $\{ comp.msg_to_handle(top_in_data(comp.top_port)) \}$</p> <p>$bot_in_data'(comp.bot_port) = bot_in_data(comp.bot_port)$</p>
--

A C2 connector has multiple ports on its top and bottom sides, one for each component attached to it. It is defined as follows:

C2Connector

$top_ports, bot_ports : \text{ff } COMM_PORT$
 $top_in, top_out, bot_in, bot_out :$
 $COMM_PORT \sim \text{ff } COMM_MSG$
 $Filter_TB : CONN_FILTER$
 $Filter_BT : CONN_FILTER$

$top_ports \cap bot_ports = \mathbb{E}$

$\text{dom } top_in = top_ports$
 $\text{dom } top_out = top_ports$
 $\text{dom } bot_in = bot_ports$
 $\text{dom } bot_out = bot_ports$

$\bigcup(\text{ran } bot_out) \subseteq \bigcup(\text{ran } top_in)$
 $\bigcup(\text{ran } top_out) \subseteq \bigcup(\text{ran } bot_in)$

A connector may have the ability to filter messages, so that the messages it emits on its bottom side are a subset of those that come in from above and the messages it emits on its top side are a subset of those that come in from below. It is thus possible to define filtering functions *Filter_TB* and *Filter_BT* that determine for each port whether a particular message will be filtered out or propagated. The below schema shows how the *Filter_TB* function is used to decide whether a given message *msg* is filtered out or propagated to a bottom port *port2*. Again, for the sake of simplicity, the two functions are assumed to filter out a message by propagating a null message.

RoutMessageFromAbove

$\Delta C2ConnectorState$

$conn' = conn$

$\forall msg : COMM_MSG; port1 : conn.top_ports \mid msg \in top_in_flow(port1) \bullet$
 $\quad \forall port2 : conn.bot_ports \bullet$
 $\quad \quad top_in_flow(port1) = top_in_flow'(port1) \cup \{ msg \}$
 $\quad \quad \wedge top_out_flow'(port1) = top_out_flow(port1)$
 $\quad \quad \wedge bot_in_flow'(port2) = bot_in_flow(port2)$
 $\quad \quad \wedge bot_out_flow'(port2) =$
 $\quad \quad \quad bot_out_flow(port2) \cup \{ conn.Filter_TB(port2, msg) \}$

The property that a component can only be attached to single connectors on its top and bottom sides is expressed as follows:

ComponentToConnectorLinks

C2Link

components : ff *C2Component*

connectors : ff *C2Connector*

$\forall comp : components \bullet$

$\exists_1 conn1, conn2 : connectors; tport, bport : COMM_PORT \mid$

$tport \in conn2.top_ports \wedge bport \in conn1.bot_ports \wedge conn1 \neq conn2 \bullet$

$(comp.top_port, bport) \in Link \wedge (comp.bot_port, tport) \in Link$

The rules for linking connectors to components and other connectors are defined in a similar manner and are provided in the Appendix.

Finally, in any given architecture, there is no guarantee that all of a component's services will be utilized by components above and below it or that the component will understand all the requests sent to it. This property is a byproduct of component reusability. A component may be used in multiple architectures, and different aspects of it may be needed in each. Since components communicate via connectors, it is possible to specify pairwise relationships between the domains of any connector and each component attached to it, and express the utilization of a component's services in terms of that relationship. For example, partial utilization of a component's services by a connector, where the component will receive some, but not all, of the messages it is capable of handling, is defined as follows:

PartialServiceUtilization

vc : *ValidC2Connections*

components : ff *C2Component*

connectors : ff *C2Connector*

$\forall comp : components; conn : connectors; conn_port : COMM_PORT \bullet$

$(conn_port \in conn.top_ports \wedge (comp.bot_port, conn_port) \in vc.Link \Rightarrow$

$conn.top_out(conn_port) \cap comp.bot_in(comp.bot_port) \neq \emptyset$

$\wedge comp.bot_in(comp.bot_port) \cap conn.top_out(conn_port) \subset$

$comp.bot_in(comp.bot_port))$

$\wedge (conn_port \in conn.bot_ports \wedge (comp.top_port, conn_port) \in vc.Link \Rightarrow$

$conn.bot_out(conn_port) \cap comp.top_in(comp.top_port) \neq \emptyset$

$\wedge comp.top_in(comp.top_port) \cap conn.bot_out(conn_port) \subset$

$comp.top_in(comp.top_port))$

5 Results of the Chiron-2 Formalism

Several of the requirements outlined in Section 2 are clearly satisfied, a few are satisfiable with a simple extension to the formalism, while it is difficult to ascertain at this time whether the formal definition satisfies the rest of the requirements. The degree to which each individual requirement is fulfilled is discussed below:

1. Requirement that the formalism must be a standard for all C2 work cannot be enforced. It will always be possible to develop architectures in the C2 style without utilizing the formalism. However, use of the formalism can be encouraged by the C2 development environment.

2. The formalism appears to be easy to learn, since Z is relatively commonly used. Furthermore, the formal definition is compact enough so that architectures expressed in it are likely to be easy to understand. However, these claims cannot be proven a priori.
3. The formalism appears suitable for analysis of architectures, but its suitability cannot be fully established before attempting to utilize it in the design environment.
4. Testing of architectures will be aided by their formal specification. At the same time, Z does not allow for explicit representation of concurrency. Therefore, it may be useful to augment this formal definition with a method, such as statecharts [Har87], that explicitly represents concurrent activities.
5. Model-based specifications can, in general, be automatically translated [Dan91]. However, it is difficult to ascertain the suitability of the C2 definition for a generation process independently of designing code generation tools themselves.
6. Currently, both a graphical and a Z representation for C2 architectures exist. What is also needed is a 1-to-1 mapping between the two.
7. The formalism is clearly capable of extension. The difficulty of such a task would most likely depend on the nature of the particular extension.
8. In its current form, the formalism does not preclude the possibility of dynamic behavior of components and connectors. On the other hand, it does not explicitly express them.
9. While the formalism currently only defines the properties of conceptual architectures, adding implementation details, such as specifying the components and connectors that are in a given process, would be a simple task.

The benefits of formally defining C2 extend beyond the degree to which the definition satisfies the proposed requirements. Merely going through the process of adding rigor to the definition of a style is a verification process. It crystallizes one's understanding of what that style is and exposes any inconsistencies and inaccuracies that existed in its less formal definition. In particular, formalizing C2 corrected a mistake in the prior definition of service utilization axioms. It, furthermore, affirmed the suspected need for the explicit representation of communication links, or paths, between individual component and connector ports. Finally, it clarified the definition of communication between a component and a connector as being on individual port-to-port basis, as opposed to component-to-entire-connector communication, as previously expressed.

6 Conclusion

The C2 style is being developed to provide a basis for the increasingly distributed, complex, multi-media, heterogeneous, and multi-user systems and particularly their interfaces. The style is based on a small set of simple rules, yet it has a great deal of power and flexibility. In order for this power and flexibility to be fully exploited, it is necessary to provide the means for rigorous specification of architectures in the style. A set of independently devised requirements that such a formal definition needs to satisfy is presented.

This paper introduces an attempt at the formal specification of C2. While it does not fully satisfy all the requirements in its present form, the definition clearly has value and shows a lot of promise. It reinforced our understanding of the style and highlighted and eliminated inconsistencies from C2's less formal

description. Furthermore, the definition can be amended if needed, both with new postulates and with other formal specification methods. Further work on the formal definition should be done in concert with developing experimental C2 systems and the development environment and only then will we be able to fully ascertain its utility.

References

- [AAG93] Gregory Abowd, Robert Allen, and David Garlan. "Using Style to Understand Descriptions of Software Architecture." In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 9-20, Los Angeles, California, December 1993.
- [AG92] Robert Allen and David Garlan. "A Formal Approach to Software Architectures." In *Proceedings of the IFIP'92*, September 1992.
- [BO92] Don Batory and Sean O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components." *ACM Transactions on Software Engineering and Methodology*, pages 355-398, October 1992.
- [Dan91] Bent Dandanell. "Rigorous Development Using RAISE." *Proceedings of the 13th International Conference on Software Engineering*, Austin, Texas, May 1991.
- [Har87] David Harel. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, 1987.
- [KP88] Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model-View-Controller Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming*, pages 26-49, August/September 1988.
- [RWMT95] Jason E. Robbins, E. James Whitehead Jr., Nenad Medvidovic, and Richard N. Taylor. "A Software Architecture Design Environment for Chiron-2 Style Architectures." Tech. Report Arcadia-UCI-95-01, U.C. Irvine, Irvine, CA, January 1995.
- [Spi89] J. M. Spivey. "The Z Notation: A Reference Manual." *Prentice Hall*, 1989.
- [TJ93] Richard N. Taylor and Gregory F. Johnson. "Separations of Concerns in the Chiron-1 User Interface Development and Management System." In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 367-374, Amsterdam, April 1993.
- [TMA+95] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., and Jason E. Robbins. "A Component- and Message-Based Architectural Style for GUI Software." In *Proceedings of the 17th International Conference on Software Engineering*, pages 295-304, Seattle, WA, April 1995.
- [WRMT95] E. James Whitehead Jr., Jason E. Robbins, Nenad Medvidovic, and Richard N. Taylor. "Software Architecture: Foundation of a Software Component Marketplace." In *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 276-282, Seattle, WA, April 1995.
- [Win90] Jeannette M. Wing. "A Specifier's Introduction to Formal Methods." *IEEE Computer*, pages 8-24, September 1990.
- [Wor92] The UIMS Tool Developers Workshop. "A Metamodel for the Runtime Architecture of an Interactive System." *SIGCHI Bulletin*, pages 32-37, January 1992.

[YTT88] Michal Young, Richard N. Taylor, and Dennis B. Troup. “Software Environment Architectures and User Interface Facilities.” *IEEE Transactions on Software Engineering*, pages 697-708, June 1988.

A Z Specification of Chiron-2

A.1 Types

$[COMP_NAME, COMM_PORT, COMM_MSG, OBJ_STATE]$

A.2 Chiron-2 Components

$NEXT_MSG == \text{ff } COMM_MSG \rightarrow COMM_MSG$

C2Component

name : *COMP_NAME*
top_port, *bot_port* : *COMM_PORT*
top_in, *top_out*, *bot_in*, *bot_out* :
 COMM_PORT ~ ff *COMM_MSG*
top_domain, *bot_domain* : ff *COMM_MSG*
dialog_in : ff *COMM_MSG* ~ *OBJ_STATE*
wrapper, *dialog_top_out* : *OBJ_STATE* ~ ff *COMM_MSG*
msg_to_handle : *NEXT_MSG*;
domain_trans : ff *COMM_MSG* ~ ff *COMM_MSG*
internal_states : ff *OBJ_STATE*
start_state : *OBJ_STATE*
state_transitions : (*OBJ_STATE* × (*COMM_PORT* ~ ff *COMM_MSG*)) →
 (*OBJ_STATE* × { (*COMM_PORT* ~ ff *COMM_MSG*),
 (*COMM_PORT* ~ ff *COMM_MSG*) })

top_port ≠ *bot_port*

dom *top_in* = { *top_port* }

dom *top_out* = { *top_port* }

dom *bot_in* = { *bot_port* }

dom *bot_out* = { *bot_port* }

top_domain = *top_in*(*top_port*) ∪ *top_out*(*top_port*)

bot_domain = *bot_in*(*bot_port*) ∪ *bot_out*(*bot_port*)

∀ *state1*, *state2* : *OBJ_STATE*;

ps1, *ps2*, *ps3* : *COMM_PORT* ~ ff *COMM_MSG* •

((*state1*, *ps1*), (*state2*, { *ps2*, *ps3* })) ∈ *state_transitions* ⇒

state1 ∈ *internal_states*

 ∧ *state2* ∈ *internal_states*

 ∧ (dom *ps1* = { *top_port* } ∨ dom *ps1* = { *bot_port* })

 ∧ (dom *ps2* = { *top_port* })

 ∧ (dom *ps3* = { *bot_port* })

 ∧ (*ps1*(*top_port*) ⊆ *top_in*(*top_port*) ∨

ps1(*bot_port*) ⊆ *bot_in*(*bot_port*))

 ∧ *ps2*(*top_port*) ⊆ *top_out*(*top_port*)

 ∧ *ps3*(*bot_port*) ⊆ *bot_out*(*bot_port*)

ComponentNameUniqueness

components : ff *C2Component*

∀ *comp1*, *comp2* : *components* •

comp1.name = *comp2.name* ⇔ *comp1* = *comp2*

A component's state is defined by its current state and the incoming and outgoing data currently at the top and bottom ports.

C2ComponentState

comp : *C2Component*

current_state : *OBJ_STATE*

top_in_data, *top_out_data*, *bot_in_data*, *bot_out_data* :
COMM_PORT ~ ff *COMM_MSG*

current_state ∈ *comp.internal_states*

dom *top_in_data* = { *comp.top_port* }

dom *top_out_data* = { *comp.top_port* }

dom *bot_in_data* = { *comp.bot_port* }

dom *bot_out_data* = { *comp.bot_port* }

top_in_data(*comp.top_port*) ⊆ *comp.top_in*(*comp.top_port*)

top_out_data(*comp.top_port*) ⊆ *comp.top_out*(*comp.top_port*)

bot_in_data(*comp.bot_port*) ⊆ *comp.bot_in*(*comp.bot_port*)

bot_out_data(*comp.bot_port*) ⊆ *comp.bot_out*(*comp.bot_port*)

HandleMessageFromAbove

$\Delta C2$ *ComponentState*

$comp' = comp$

$((current_state, top_in_data),$
 $(current_state', \{ top_out_data', bot_out_data' \})) \in$
 $comp.state_transitions$

$top_out_data'(comp.top_port) =$
 $top_out_data(comp.top_port) \cup$
 $\{ comp.domain_trans($
 $comp.dialog_top_out($
 $comp.dialog_in($
 $comp.domain_trans($
 $comp.msg_to_handle($
 $top_in_data(comp.top_port)))))) \}$

$bot_out_data'(comp.bot_port) =$
 $bot_out_data(comp.bot_port) \cup$
 $\{ comp.wrapper($
 $comp.dialog_in($
 $comp.domain_trans($
 $comp.msg_to_handle($
 $top_in_data(comp.top_port)))) \}$

$top_in_data(comp.top_port) =$
 $top_in_data'(comp.top_port) \cup$
 $\{ comp.msg_to_handle(top_in_data(comp.top_port)) \}$

$bot_in_data'(comp.bot_port) = bot_in_data(comp.bot_port)$

HandleMessageFromBelow

$\Delta C2$ *ComponentState*

$comp' = comp$

$((current_state, bot_in_data),$
 $(current_state', \{ top_out_data', bot_out_data' \})) \in$
 $comp.state_transitions$

$top_out_data'(comp.top_port) =$
 $top_out_data(comp.top_port) \cup$
 $\{ comp.domain_trans($
 $comp.dialog_top_out($
 $comp.dialog_in($
 $comp.msg_to_handle($
 $bot_in_data(comp.bot_port)))) \}$

$bot_out_data'(comp.bot_port) =$
 $bot_out_data(comp.bot_port) \cup$
 $\{ comp.wrapper($
 $comp.dialog_in($
 $comp.msg_to_handle($
 $bot_in_data(comp.bot_port)))) \}$

$top_in_data'(comp.top_port) = top_in_data(comp.top_port)$

$bot_in_data(comp.bot_port) =$
 $bot_in_data'(comp.bot_port) \cup$
 $\{ comp.msg_to_handle(bot_in_data(comp.bot_port)) \}$

ComponentMessageHandling $\hat{=}$

$HandleMessageFromAbove \wedge HandleMessageFromBelow$

A.3 Chiron-2 Connectors

$CONN_FILTER == (COMM_PORT \times COMM_MSG) \rightarrow COMM_MSG$

C2Connector

$top_ports, bot_ports : \text{ff } COMM_PORT$
 $top_in, top_out, bot_in, bot_out :$
 $COMM_PORT \sim \text{ff } COMM_MSG$
 $Filter_TB : CONN_FILTER$
 $Filter_BT : CONN_FILTER$

$top_ports \cap bot_ports = \emptyset$

$\text{dom } top_in = top_ports$
 $\text{dom } top_out = top_ports$
 $\text{dom } bot_in = bot_ports$
 $\text{dom } bot_out = bot_ports$

$\bigcup(\text{ran } bot_out) \subseteq \bigcup(\text{ran } top_in)$
 $\bigcup(\text{ran } top_out) \subseteq \bigcup(\text{ran } bot_in)$

C2ConnectorState

$conn : C2Connector$
 $top_in_flow, top_out_flow, bot_in_flow, bot_out_flow :$
 $COMM_PORT \sim \text{ff } COMM_MSG$

$\text{dom } top_in_flow \cup \text{dom } top_out_flow \subseteq conn.top_ports$
 $\text{dom } bot_in_flow \cup \text{dom } bot_out_flow \subseteq conn.bot_ports$

$\forall port : conn.top_ports \bullet$
 $top_in_flow(port) \subseteq conn.top_in(port) \wedge$
 $top_out_flow(port) \subseteq conn.top_out(port)$
 $\forall port : conn.bot_ports \bullet$
 $bot_in_flow(port) \subseteq conn.bot_in(port) \wedge$
 $bot_out_flow(port) \subseteq conn.bot_out(port)$

RoutMessageFromAbove

$\Delta C2ConnectorState$

$conn' = conn$

$\forall msg : COMM_MSG; port1 : conn.top_ports \mid msg \in top_in_flow(port1) \bullet$
 $\forall port2 : conn.bot_ports \bullet$
 $top_in_flow(port1) = top_in_flow'(port1) \cup \{ msg \}$
 $\wedge top_out_flow'(port1) = top_out_flow(port1)$
 $\wedge bot_in_flow'(port2) = bot_in_flow(port2)$
 $\wedge bot_out_flow'(port2) =$
 $bot_out_flow(port2) \cup \{ conn.Filter_TB(port2, msg) \}$

$\text{RoutMessageFromBelow}$ <hr/> $\Delta C2ConnectorState$ <hr/> $conn' = conn$ $\forall msg : COMM_MSG; port1 : conn.bot_ports \mid msg \in bot_in_flow(port1) \bullet$ $\quad \forall port2 : conn.top_ports \bullet$ $\quad \quad top_in_flow'(port2) = top_in_flow(port2)$ $\quad \wedge top_out_flow'(port2) =$ $\quad \quad top_out_flow(port2) \cup \{ conn.Filter_BT(port2, msg) \}$ $\quad \wedge bot_in_flow(port1) = bot_in_flow'(port1) \cup \{ msg \}$ $\quad \wedge bot_out_flow'(port1) = bot_out_flow(port1)$
--

$ConnectorMessageRouting \hat{=} RoutMessageFromAbove \wedge RoutMessageFromBelow$

A.4 Rules of Composition between Components and Connectors

$LINK == COMM_PORT \leftrightarrow COMM_PORT$

$C2Link$ <hr/> $Link : LINK$ <hr/> $\forall port1, port2 : COMM_PORT \bullet$ $\quad (port1, port2) \in Link \Leftrightarrow (port2, port1) \in Link$
--

$ComponentToConnectorLinks$ <hr/> $C2Link$ $components : ff C2Component$ $connectors : ff C2Connector$ <hr/> $\forall comp : components \bullet$ $\quad \exists_1 conn1, conn2 : connectors; tport, bport : COMM_PORT \mid$ $\quad \quad tport \in conn2.top_ports \wedge bport \in conn1.bot_ports \wedge conn1 \neq conn2 \bullet$ $\quad \quad (comp.top_port, bport) \in Link \wedge (comp.bot_port, tport) \in Link$
--

$ConnectorToComponentLinks$ <hr/> $C2Link$ $components : ff C2Component$ $connectors : ff C2Connector$ <hr/> $\forall conn : connectors; tport, bport : COMM_PORT \mid$ $\quad tport \in conn.top_ports \wedge bport \in conn.bot_ports \bullet$ $\quad \exists_1 comp1, comp2 : components \mid comp1 \neq comp2 \bullet$ $\quad \quad (tport, comp1.bot_port) \in Link \wedge (bport, comp2.top_port) \in Link$
--

ConnectorToConnectorLinks

*C2Link**components* : ff *C2Component**connectors* : ff *C2Connector*

$$\begin{aligned} &\forall \text{conn} : \text{connectors}; \text{tport}, \text{bport} : \text{COMM_PORT} \mid \\ &\quad \text{tport} \in \text{conn.top_ports} \wedge \text{bport} \in \text{conn.bot_ports} \bullet \\ &\quad \exists_1 \text{conn1}, \text{conn2} : \text{connectors}; \text{c2tport}, \text{c1bport} : \text{COMM_PORT} \mid \\ &\quad \quad \text{c2tport} \in \text{conn2.top_ports} \wedge \text{c1bport} \in \text{conn1.bot_ports} \wedge \\ &\quad \quad \text{conn} \neq \text{conn1} \wedge \text{conn} \neq \text{conn2} \wedge \text{conn1} \neq \text{conn2} \bullet \\ &\quad \quad (\text{tport}, \text{c1bport}) \in \text{Link} \wedge (\text{bport}, \text{c2tport}) \in \text{Link} \end{aligned}$$

ValidC2Connections $\hat{=}$ *ComponentToConnectorLinks* \wedge *ConnectorToComponentLinks* \wedge *ConnectorToConnectorLinks*

A.5 Communication among Components and Connectors

A connector's domain is determined dynamically by the domains of the components attached to it at a given time. Hence, a connector will accept every message sent by the components attached to it.

ConnectorDomains

vc : *ValidC2Connections**components* : ff *C2Component**connectors* : ff *C2Connector*

$$\begin{aligned} &\forall \text{conn} : \text{connectors}; \text{comp} : \text{components}; \text{conn_port} : \text{COMM_PORT} \bullet \\ &\quad \text{conn_port} \in \text{conn.bot_ports} \wedge (\text{comp.top_port}, \text{conn_port}) \in \text{vc.Link} \Rightarrow \\ &\quad \quad \text{comp.top_out}(\text{comp.top_port}) \subseteq \text{conn.bot_in}(\text{conn_port}) \\ &\quad \wedge \text{conn_port} \in \text{conn.top_ports} \wedge (\text{comp.bot_port}, \text{conn_port}) \in \text{vc.Link} \Rightarrow \\ &\quad \quad \text{comp.bot_out}(\text{comp.bot_port}) \subseteq \text{conn.top_in}(\text{conn_port}) \end{aligned}$$

TransmitMessageUpFromComponentToConnector

\exists *ValidC2Connections*

Δ *C2ComponentState*

Δ *C2ConnectorState*

$\forall msg : COMM_MSG; bot_conn_port : conn.bot_ports \bullet$
 $(comp.top_port, bot_conn_port) \in Link \wedge$
 $msg \in top_out_data(comp.top_port) \Rightarrow$
 $(top_out_data(comp.top_port) =$
 $top_out_data'(comp.top_port) \cup \{ msg \}$
 $\wedge bot_in_flow'(bot_conn_port) =$
 $bot_in_flow(bot_conn_port) \cup \{ msg \}$
 $\wedge conn' = conn$
 $\wedge top_in_flow' = top_in_flow$
 $\wedge top_out_flow' = top_out_flow$
 $\wedge bot_out_flow' = bot_out_flow$
 $\wedge comp' = comp$
 $\wedge current_state' = current_state$
 $\wedge top_in_data' = top_in_data$
 $\wedge bot_in_data' = bot_in_data$
 $\wedge bot_out_data' = bot_out_data)$

TransmitMessageDownFromComponentToConnector

\exists *ValidC2Connections*

Δ *C2ComponentState*

Δ *C2ConnectorState*

$\forall msg : COMM_MSG; top_conn_port : conn.top_ports \bullet$
 $(comp.bot_port, top_conn_port) \in Link \wedge$
 $msg \in bot_out_data(comp.bot_port) \Rightarrow$
 $(bot_out_data(comp.bot_port) =$
 $bot_out_data'(comp.bot_port) \cup \{ msg \}$
 $\wedge top_in_flow'(top_conn_port) =$
 $top_in_flow(top_conn_port) \cup \{ msg \}$
 $\wedge conn' = conn$
 $\wedge top_out_flow' = top_out_flow$
 $\wedge bot_in_flow' = bot_in_flow$
 $\wedge bot_out_flow' = bot_out_flow$
 $\wedge comp' = comp$
 $\wedge current_state' = current_state$
 $\wedge top_in_data' = top_in_data$
 $\wedge top_out_data' = top_out_data$
 $\wedge bot_in_data' = bot_in_data)$

TransmitMessageUpFromConnectorToConnector

\exists *ValidC2Connections*

from_conn, *to_conn* : Δ *C2ConnectorState*

\forall *msg* : *COMM_MSG*;

from_port : *from_conn.conn.top_ports*; *to_port* : *to_conn.conn.bot_ports* •
(*from_port*, *to_port*) \in *Link* \wedge
msg \in *from_conn.top_out_flow*(*from_port*) \Rightarrow
(*from_conn.top_out_flow*(*from_port*) =
 from_conn.top_out_flow'(*from_port*) \cup { *msg* }
 \wedge *to_conn.bot_in_flow'*(*to_port*) =
 to_conn.bot_in_flow(*to_port*) \cup { *msg* }
 \wedge *from_conn.conn'* = *from_conn.conn*
 \wedge *from_conn.top_in_flow'* = *from_conn.top_in_flow*
 \wedge *from_conn.bot_in_flow'* = *from_conn.bot_in_flow*
 \wedge *from_conn.bot_out_flow'* = *from_conn.bot_out_flow*
 \wedge *to_conn.conn'* = *to_conn.conn*
 \wedge *to_conn.top_in_flow'* = *to_conn.top_in_flow*
 \wedge *to_conn.top_out_flow'* = *to_conn.top_out_flow*
 \wedge *to_conn.bot_out_flow'* = *from_conn.bot_out_flow*)

TransmitMessageDownFromConnectorToConnector

\exists *ValidC2Connections*

from_conn, *to_conn* : Δ *C2ConnectorState*

\forall *msg* : *COMM_MSG*;

from_port : *from_conn.conn.bot_ports*; *to_port* : *to_conn.conn.top_ports* •
(*from_port*, *to_port*) \in *Link* \wedge
msg \in *from_conn.bot_out_flow*(*from_port*) \Rightarrow
(*from_conn.bot_out_flow*(*from_port*) =
 from_conn.bot_out_flow'(*from_port*) \cup { *msg* }
 \wedge *to_conn.top_in_flow'*(*to_port*) =
 to_conn.top_in_flow(*to_port*) \cup { *msg* }
 \wedge *from_conn.conn'* = *from_conn.conn*
 \wedge *from_conn.top_in_flow'* = *from_conn.top_in_flow*
 \wedge *from_conn.top_out_flow'* = *from_conn.top_out_flow*
 \wedge *from_conn.bot_in_flow'* = *from_conn.bot_in_flow*
 \wedge *to_conn.conn'* = *to_conn.conn*
 \wedge *to_conn.top_out_flow'* = *to_conn.top_out_flow*
 \wedge *to_conn.bot_in_flow'* = *to_conn.bot_in_flow*
 \wedge *to_conn.bot_out_flow'* = *from_conn.bot_out_flow*)

A component will accept only those messages sent by a connector that it understands.

TransmitMessageUpFromConnectorToComponent

\exists *ValidC2Connections*

Δ *C2ComponentState*

Δ *C2ConnectorState*

$\forall msg : COMM_MSG; top_conn_port : conn.top_ports \bullet$
 $(comp.bot_port, top_conn_port) \in Link \wedge$
 $msg \in top_out_flow(top_conn_port) \Rightarrow$
 $(top_out_flow(top_conn_port) =$
 $top_out_flow'(top_conn_port) \cup \{ msg \}$
 $\wedge msg \in comp.bot_in(comp.bot_port) \Rightarrow$
 $bot_in_data'(comp.bot_port) =$
 $bot_in_data(comp.bot_port) \cup \{ msg \}$
 $\wedge msg \notin comp.bot_in(comp.bot_port) \Rightarrow$
 $bot_in_data'(comp.bot_port) = bot_in_data(comp.bot_port)$
 $\wedge conn' = conn$
 $\wedge top_in_flow' = top_in_flow$
 $\wedge bot_in_flow' = bot_in_flow$
 $\wedge bot_out_flow' = bot_out_flow$
 $\wedge comp' = comp$
 $\wedge current_state' = current_state$
 $\wedge top_in_data' = top_in_data$
 $\wedge top_out_data' = top_out_data$
 $\wedge bot_out_data' = bot_out_data)$

TransmitMessageDownFromConnectorToComponent

 $\exists \text{ValidC2Connections}$ $\Delta \text{C2ComponentState}$ $\Delta \text{C2ConnectorState}$
$$\begin{aligned} & \forall \text{msg} : \text{COMM_MSG}; \text{bot_conn_port} : \text{conn.bot_ports} \bullet \\ & (\text{comp.top_port}, \text{bot_conn_port}) \in \text{Link} \wedge \\ & \text{msg} \in \text{bot_out_flow}(\text{bot_conn_port}) \Rightarrow \\ & \quad (\text{bot_out_flow}(\text{bot_conn_port}) = \\ & \quad \quad \text{bot_out_flow}'(\text{bot_conn_port}) \cup \{ \text{msg} \} \\ & \wedge \text{msg} \in \text{comp.top_in}(\text{comp.top_port}) \Rightarrow \\ & \quad \text{top_in_data}'(\text{comp.top_port}) = \\ & \quad \quad \text{top_in_data}(\text{comp.top_port}) \cup \{ \text{msg} \} \\ & \wedge \text{msg} \notin \text{comp.top_in}(\text{comp.top_port}) \Rightarrow \\ & \quad \text{top_in_data}'(\text{comp.top_port}) = \text{top_in_data}(\text{comp.top_port}) \\ & \wedge \text{conn}' = \text{conn} \\ & \wedge \text{bot_in_flow}' = \text{bot_in_flow} \\ & \wedge \text{top_in_flow}' = \text{top_in_flow} \\ & \wedge \text{top_out_flow}' = \text{top_out_flow} \\ & \wedge \text{comp}' = \text{comp} \\ & \wedge \text{current_state}' = \text{current_state} \\ & \wedge \text{bot_in_data}' = \text{bot_in_data} \\ & \wedge \text{bot_out_data}' = \text{bot_out_data} \\ & \wedge \text{top_out_data}' = \text{top_out_data} \end{aligned}$$
 $C2\text{MessageTransmission} \hat{=}$
$$\begin{aligned} & \text{TransmitMessageUpFromComponentToConnector} \wedge \\ & \text{TransmitMessageDownFromComponentToConnector} \wedge \\ & \text{TransmitMessageUpFromConnectorToConnector} \wedge \\ & \text{TransmitMessageDownFromConnectorToConnector} \wedge \\ & \text{TransmitMessageUpFromConnectorToComponent} \wedge \\ & \text{TransmitMessageDownFromConnectorToComponent} \end{aligned}$$

The *ServiceUtilization* schemas represent the communication between components and connectors from component's perspective, i.e., usage of the services provided by a component.

FullServiceUtilization

 $vc : \text{ValidC2Connections}$ $\text{components} : \text{ff } C2\text{Component}$ $\text{connectors} : \text{ff } C2\text{Connector}$
$$\begin{aligned} & \forall \text{comp} : \text{components}; \text{conn} : \text{connectors}; \text{conn_port} : \text{COMM_PORT} \bullet \\ & (\text{conn_port} \in \text{conn.top_ports} \wedge (\text{comp.bot_port}, \text{conn_port}) \in \text{vc.Link} \Rightarrow \\ & \quad \text{comp.bot_in}(\text{comp.bot_port}) \subseteq \text{conn.top_out}(\text{conn_port})) \\ & \wedge (\text{conn_port} \in \text{conn.bot_ports} \wedge (\text{comp.top_port}, \text{conn_port}) \in \text{vc.Link} \Rightarrow \\ & \quad \text{comp.top_in}(\text{comp.top_port}) \subseteq \text{conn.bot_out}(\text{conn_port})) \end{aligned}$$

PartialServiceUtilization

$vc : ValidC2Connections$
 $components : ff C2Component$
 $connectors : ff C2Connector$

$\forall comp : components; conn : connectors; conn_port : COMM_PORT \bullet$
 $(conn_port \in conn.top_ports \wedge (comp.bot_port, conn_port) \in vc.Link \Rightarrow$
 $conn.top_out(conn_port) \cap comp.bot_in(comp.bot_port) \neq \mathbb{E}$
 $\wedge comp.bot_in(comp.bot_port) \cap conn.top_out(conn_port) \subset$
 $comp.bot_in(comp.bot_port))$
 $\wedge (conn_port \in conn.bot_ports \wedge (comp.top_port, conn_port) \in vc.Link \Rightarrow$
 $conn.bot_out(conn_port) \cap comp.top_in(comp.top_port) \neq \mathbb{E}$
 $\wedge comp.top_in(comp.top_port) \cap conn.bot_out(conn_port) \subset$
 $comp.top_in(comp.top_port))$

The *Communication* schemas represent the communication from connector's perspective, i.e., fulfillment of the requests submitted by a connector.

FullCommunication

$vc : ValidC2Connections$
 $components : ff C2Component$
 $connectors : ff C2Connector$

$\forall comp : components; conn : connectors; conn_port : COMM_PORT \bullet$
 $(conn_port \in conn.top_ports \wedge (comp.bot_port, conn_port) \in vc.Link \Rightarrow$
 $conn.top_out(conn_port) \subseteq comp.bot_in(comp.bot_port))$
 $\wedge (conn_port \in conn.bot_ports \wedge (comp.top_port, conn_port) \in vc.Link \Rightarrow$
 $conn.bot_out(conn_port) \subseteq comp.top_in(comp.top_port))$

PartialCommunication

$vc : ValidC2Connections$
 $components : ff C2Component$
 $connectors : ff C2Connector$

$\forall comp : components; conn : connectors; conn_port : COMM_PORT \bullet$
 $(conn_port \in conn.top_ports \wedge (comp.bot_port, conn_port) \in vc.Link \Rightarrow$
 $conn.top_out(conn_port) \cap comp.bot_in(comp.bot_port) \neq \mathbb{E}$
 $\wedge comp.bot_in(comp.bot_port) \cap conn.top_out(conn_port) \subset$
 $conn.top_out(conn_port))$
 $\wedge (conn_port \in conn.bot_ports \wedge (comp.top_port, conn_port) \in vc.Link \Rightarrow$
 $conn.bot_out(conn_port) \cap comp.top_in(comp.top_port) \neq \mathbb{E}$
 $\wedge comp.top_in(comp.top_port) \cap conn.bot_out(conn_port) \subset$
 $conn.bot_out(conn_port))$

NoInteraction

vc : *ValidC2Connections*
components : ff *C2Component*
connectors : ff *C2Connector*

$\forall comp : components; conn : connectors; conn_port : COMM_PORT \bullet$
 $(conn_port \in conn.top_ports \wedge (comp.bot_port, conn_port) \in vc.Link \Rightarrow$
 $conn.top_out(conn_port) \cap comp.bot_in(comp.bot_port) = \mathbb{E})$
 $\wedge (conn_port \in conn.bot_ports \wedge (comp.top_port, conn_port) \in vc.Link \Rightarrow$
 $conn.bot_out(conn_port) \cap comp.top_in(comp.top_port) = \mathbb{E})$

ComponentServiceUtilization $\hat{=}$

FullServiceUtilization \vee *PartialServiceUtilization* \vee *NoInteraction*

ConnectorToComponentCommunication $\hat{=}$

FullCommunication \vee *PartialCommunication* \vee *NoInteraction*