Architecture Descriptions of Software Systems: Complex Connectors vs Realisability

Mert Ozkaya
Istanbul Kemerburgaz University, Istanbul, Turkey

Keywords: Architectural Languages, Interaction Protocols, Realisability.

Abstract: With the advent of software architectures, architectural languages have become an active research area for the specification of software architectures in terms of components & connectors and for some extra capabilities such as formal analysis and code generation. In this paper, the existing architectural languages have been analysed for two important language features - i.e., *interaction protocols* and *realisability*. The analysis results show that only a few languages support interaction protocols via their first-class connector elements (also referred to as complex connectors). However, complex connectors of those languages lead to unrealisable specifications due to enabling global constraints which may not be possible for distributed systems. Therefore, practitioners cannot implement the system in the way specified, and any analyses (e.g., performance) made on the unrealisable specifications will all be invalid.

1 INTRODUCTION

Software architecture (Clements et al., 2003; Perry and Wolf, 1992; Garlan and Shaw, 1994) has been proposed in the early nineties as a high-level design method for specifying software systems in terms of components and their relationships (i.e., connectors). Since then, many architectural languages (ALs) have been developed through which software architectures can be specified at varying levels of abstractions and further used for some useful operations such as formal analysis and code generation.

According to the recent analyses (Lago et al., 2015; Malavolta et al., 2012), there are approximately 120 known ALs[1]. These ALs can be either UML-like languages, architecture description languages (ADLs), and formal specification languages. UML-like languages derive from UML (Rumbaugh et al., 2005) , which is an informal modelling language that offers a visual (i.e., diagrammatic) notation set for designing software systems including architectural design. ADLs (Medvidovic and Taylor, 2000) are precise languages that are exclusively used for architectural design. ADLs can be either general-purpose (used for any systems) or domain-specific. General-purpose ADLs are useful for the high-level specifications and their formal analysis to detect high-level design errors such as deadlocking components. Domain-specific languages are better for detailed specifications of systems in particular domains which can then be used for, e.g., implementation generation. Formal specification languages (e.g., ProMeLa (Holzmann, 2004) and FSP (Magee et al., 1997)) are based on formal syntax and semantics (i.e., mostly algebraic), which enable the exhaustive formal analysis of architectural models. Given so many ALs with different scopes and capabilities, one might expect ALs to be so popular among practitioner in industry. However, this has never been the case - ALs remain in the focus of research communities only (Malavolta et al., 2012; Ozkaya, 2016b). As stated in (Malavolta et al., 2012), there are very few ALs, such as UML (Rumbaugh et al., 2005) and AADL (Feiler et al., 2006), which practitioners prefer in their architectural modelling. Most ALs are either *(i)* used by expert practitioners who have deep knowledge and experience in architectural modelling or *(ii)* used for research purposes only.

Lago etal (Lago et al., 2015) proposed a framework for classifying ALs on a set of requirements that are grouped as language definition, language features, and tool support (Lago et al., 2015). Language definition is concerned with the syntax and semantics of languages, e.g., support for first-class components and connectors. Language features are concerned

[1]The list of ALs can be accessible here: https://sites.google.com/site/ozkayamert1/als

Ozkaya M.
Architecture Descriptions of Software Systems: Complex Connectors vs Realisability.
DOI: 10.5220/0006139503820391
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 382-391
ISBN: 978-989-758-210-3

with the capabilities of languages in specifying software architectures, e.g., support for multiple viewpoints and extensibility. Tool support is mainly to do with formal analysis and implementation code generation facilities. Inspired from Lago etal's classification framework, I analysed the existing 120 ALs for a number of requirements (Ozkaya, 2016a). My analysis sheds light into many interesting issues that may have been ignored by the language designers which may however affect the practical use of the languages. In this paper, I focus on two key requirements, i.e., component & connector view and realisability, which I believe are highly important for the practical use of the languages by practitioners. Components & connectors view separates high-level complex interaction mechanisms (i.e., interaction protocols) from components' computations and considers them as connectors. Note that connectors encapsulating interaction protocols are referred to as complex connectors in the rest of the paper. Realisability is, informally, the capability of implementing a software system in exactly the same way as its software architecture specification. That is, the implemented system is expected to have the same components that behave and interact in the same way. The goal of this paper is to analyse the existing languages for these two key requirements and discuss the relationship between complex connectors and realisability.

In the rest of the paper, languages' support for complex connectors (i.e., interaction protocol) is discussed firstly. This is then followed by the discussion of the realisability of the languages. Lastly, the relationship between supporting complex connectors and guaranteeing the realisability of software architectures is discussed.

2 COMPLEX CONNECTOR SUPPORT

To describe *complex connectors*, one should start from one of the best regarded definition of software architecture made by Garlan and Shaw (Garlan and Shaw, 1994): *an architecture of a specific system is treated as a collection of computational components – or simply components – together with a description of the interactions between these components – the connectors.* So, while components are the units of computations, connectors in software architectures represent the units of interactions for the components. A connector can be as simple as communication links through which components exchange synchronous/asynchronous messages among each other. Procedure call and event-broadcasting are some familiar examples of simple connectors, which simply describe the style of communications. A complex connector also deals with the constraints on the way the communicating components interact with each other, which is also known as the protocols of interactions. An interaction protocol (Kloukinas and Ozkaya, 2012) essentially describes for a set of communicating components the order of messages that the components exchange so as to be composed together to a successful system.

Specifying software architectures in terms of components and complex connectors enhances the modularity and thus the practical use of the languages. First of all, architectural designs will be more understandable as one can easily distinguish between the computation (i.e., components) and interaction (i.e., complex connectors). Modularity also aids in the analysis of software architectures. Indeed, detecting any design errors due to wrong (e.g., deadlocking) interaction protocols that hinder the composition of system components or wrong computations of components will be easier since the computations and interaction protocols are cleanly separated. Moreover, finding out the optimum design decisions will be easier too as components can easily be re-used and experimented with different interaction protocols.

In the rest of this section, the languages are evaluated for their complex connector (i.e., interaction protocols) support. Each language is taken into consideration in terms of their support for components and complex connectors. However, due to the space restriction, only the most well-known languages (i.e., most cited by industry and academia) are considered herein that have inspired many of the existing languages today. The full list of ALs can be found in (Ozkaya, 2016a).

Wright. (Allen and Garlan, 1997). A component type in Wright is specified with interfaces and a computation. A component interface can operate as many actions as desired in its environment. A component computation is used (optionally) to specify either (i) a configuration of component and connector instances, or, (ii) a protocol for coordinating the interface behaviours.

Connector Support. In Wright, besides first-class component elements, *connectors* are also first-class elements, thus enabling the explicit specification of interactions among components. Indeed, one can describe with Wright connectors either simple interconnection mechanisms (e.g., procedure call) or complex ones (e.g., complex interaction protocols such as an auction).

Connectors in Wright are instantiated from connector types, which enables reuse of the same interaction pattern on different contexts and also the analysis of connectors in isolation. A connector type is described with *roles* representing the participating components and a *glue* coordinating the behaviour of the roles. Roles and glue are each specified with a protocol representing their behaviours.

C2. (Medvidovic et al., 1996; Taylor et al., 1996). A component type in C2 is specified with an interface and a computation. A component interface specification is two-fold: a *top_domain* and a *bottom_domain*. The *top_domain* represents *requests*, which are emitted by the component, and *notifications*, which it reacts to. The *bottom_domain* represents *requests*, which can be received, and *notifications*, which can be sent. For a component computation, it comprises a set of methods, representing the inner functionality of the component, and a behaviour part, coordinating the calls made to these methods.

Connector Support. C2 does not allow for specifying complex interaction mechanisms. However, it offers a connector element that can either route event messages between components or broadcast messages from a component to multiple components. Connectors allow also the filtering of messages via a set of built-in policies, i.e., *no filtering, notification filtering, prioritised*, and *message sink*.

A connector is specified with roles consisting of top- and bottom-domains, which are used to specify the components it connects together.

Moreover, connectors in C2 are specified as part of the *architecture* element, which is used to specify a configuration of components and connectors for a system. So unlike Wright connectors, C2 connectors cannot be specified as abstractions and re-used in different configurations. Within the body of *architecture*, the style of the connector is specified that describes its policies for message filtering. Then, in its *architecture_topology*, the bottom and top domains of the connector are associated with components.

Darwin. (Magee and Kramer, 1996). Darwin is one of the first architecture description languages, intended as a general-purpose language for specifying distributed systems as configurations of components.

In Darwin, software architectures are specified in terms of hierarchical components. Component types in Darwin are specified with interfaces they provide to their environment and require from them too. Each interface of a component is responsible for the communication of a single message.

Connector Support. Darwin does not support the specification of connectors in architectural designs. Components interact with each other through *bindings* specified in composite component types. However, bindings cannot describe the way interaction occurs between components, thus resulting in the protocols of interactions being hard-wired inside components. This not only overcomplicates component specifications but also reduces their re-usability and hampers the architectural evaluation of different candidate interaction protocols.

Rapide. (Luckham, 1996). A component type in Rapide is specified with interfaces, which serve for either asynchronous (observing and generating events) or synchronous communication (providing and requiring functions). Each interface consists of actions that represent asynchronous events or synchronous functions. An interface also include *behaviour* specifications, representing the external behaviours of the components.

Connector Support. Like Darwin, Rapide adopts an approach that considers system architectures as collections of components which are wired together via mere connections. So, unlike Wright, there is no first-class connector element offered, leading complex interaction patterns to be implicitly specified in component specifications.

On the other hand, Rapide introduces *architectural constraints*, through which global interaction protocols for the interacting components can be specified. But, unlike Wright, where connectors are independent elements, Rapide constraints are embedded within an *architecture* specification, and thus cannot be re-used in different architecture specifications.

MetaH. (Binns et al., 1996). Unlike other ADLs introduced so far, MetaH does not allow designers to specify their own component types. Instead, a set of low-level pre-defined types are offered. Software architectures are specified with *subprogram* and *packages* component types, while hardware architectures with *monitor, memory, process, channel*, and *device* types. Designers can use the pre-defined component types and include interfaces inside their component type specifications. Interfaces herein are first-class elements in MetaH that are specified externally and then used in component specifications. Finally, the computations of components are specified as a collection of attributes, which are used to describe the non-functional requirements, e.g., schedulability and reliability details of components.

MetaH offers additional component types (i.e., *modes* and *macros*) for specifying the configuration of components.

Connector Support. MetaH does not offer first-class connectors. Connectors are viewed as connection links, which are used in configuration components to connect the interfaces of their sub-components.

UniCon. (Shaw et al., 1995). Component types in UniCon can be either primitive or composite. Every component is specified with a *type*, which is chosen among the pre-defined types offered by UniCon. (e.g., *sharedData*, *process*, and *filter*). These pre-defined types determine the interface of the components, i.e., its actions (named as *players* in UniCon). Note that UniCon also has a *general* component type, allowing designers to specify generic types without any restriction on interfaces. A primitive component type can also include implementation details (e.g., location of source code).
Connector Support. Connector types in UniCon are introduced as first-class elements. A connector type is specified with an interaction protocol, acting as a mediator of interaction among components. Protocols herein, just like Wright connectors, consist essentially of roles. However, unlike Wright, UniCon restricts protocols to be of certain types, e.g., *Pipe*, *DataAccess*, and *ProcedureCall*, thus preventing designers from freely specifying their own (complex) types.

Koala. (van Ommering et al., 2000). As in Darwin, system architectures in Koala are specified in terms of components. However, unlike Darwin, Koala offers first-class *interface* elements, encapsulating methods. So, interfaces are then employed within components, which either require or provide their methods.

Component types in Koala can also be composite by including a computation (i.e., a configuration of components). Koala introduces *contains* and *connects* constructs (corresponding to *inst* and *bindings* in Darwin respectively) for specifying the configuration of components in composite types.
Connector Support. Like Darwin, Koala does not support connectors in system architectures either. Interactions between components are merely specified by *connects* in composite component types. Being simple links, these cannot be used to specify complex interaction protocols independently of components. Nevertheless, as aforementioned, Koala offers *module*, which can be employed in a composite component and connected with the interfaces of the interacting sub components to order their the method-calls. So, modules in Koala can be used to specify interaction protocols for components.

COSA. (Oussalah et al., 2004). Component types in COSA are specified with interfaces and a computation. Furthermore, a component type can include *properties* for specifying non-functional properties and a *constraint*, which is again a *property* that is used to specify certain policies to be met by the components.

While components can be composite too, consisting of component and connector instances, COSA also offers first-class *instance* elements for specifying configurations of components and connectors.

Connector Support. COSA offers first-class connector types, which are specified with a collection of *roles* for participating component interfaces and a *glue* for representing a global interaction protocol. Designers can also specify some other interaction details as part of connectors, such as the type of connections or the mode of connections. While the connection types can be either communication, conversion, coordination, or facilitation, the connection-mode can be synchronous or asynchronous connections.

COSA also introduces *composite* connector types. A composite connector is specified via a *glue* element, with which designers can specify a configuration of component and connector instances. So with COSA, it is possible to specify complex connectors modularly, by re-using the existing component and connector type instances.

XADL. (Dashofy et al., 2002). XADL provides the very basic elements for an architecture description in terms of XML schemas. These basic schemas can be extended to new schemas by adding/removing new features, which enables the creation of specific constructs fitting better the designers own needs. XADL offers a *design-time* schema, which can be used by designers to specify their software architectures. The *design-time* schema includes the commonly used architectural constructs, e.g., component, connector, interface, and link, and allows designers to specify basic information about them, e.g., their id, description, and type. So, designers can use the *design-time* schema to specify their component types with their type, description, and interface(s). However, if the existing features of the construct are not enough, designers can extend the *design-time* schemas and add features meeting their particular needs. Indeed, designers can add features for specifying behaviours in some formalisms, e.g., Wright's interface protocols.
Connector Support. Just like component types, connector types are also supported by the *design-time* schema, specified with type, description, and interface(s). So, designers can either use its connector construct as it is or extend it to add extra features, e.g., Wright's complex connector.

CONNECT. (Issarny et al., 2011). Components are simply specified with interfaces representing the behaviours of the components in their interaction with their environment.

Connector Support. Just like Wright connectors, connectors in CONNECT are specified with *roles* and a *glue*, where the roles represent the participating components (namely their interfaces) and the glue represents their coordination.

LEDA. (Canal et al., 1999). Component types in LEDA are specified either as primitive or composite. Regardless of being primitive or composite, a component type is specified with interfaces and computation (i.e., optional). An interface is a first-class element, which, once specified, can then be used externally in component specifications to describe their interaction points. Computation of components is specified as a protocol, which coordinates the interface behaviours. Moreover, composite component types further include *composition* and *attachments*, representing its computation as a configuration of components. *Connector Support.* Like Darwin, LEDA does not support connectors either. Indeed, its only interaction mechanism is the simple *attachments*, specified within composite component types for linking the component interfaces. Complex interaction mechanisms (i.e., interaction protocols) can only be specified as part of components, which makes components less re-usable and protocol dependent.

SOFA. (Plasil and Visnovsky, 2002; Bures et al., 2006). A component type in SOFA is specified with component *frame* and a computation. A component frame represents the external view of a component type that consists of required (*requires*) or provided (*provides*) interfaces. It should be noted that these interfaces are essentially the instances of interface abstractions that are specified externally as first-class elements. For component computations, they are specified for composite components and consist of component instances and connection links between required and provided interfaces of these component instances. *Connector Support.* SOFA offers pre-defined basic interaction mechanisms, i.e., *procedure call, messaging, streaming*, and *blackboard* (Bures and Plasil, 2004). So, designers can specify their component interactions using these basic interactions. Moreover, SOFA allows designers to specify their own connectors too. It provides connector generation tools, through which designers can choose any of the pre-defined interaction mechanisms and specify some non-functional properties for these mechanisms (Galik and Bures, 2005; Bures, 2005). Nevertheless, it is not possible to specify complex interaction mechanisms (i.e., interaction protocols) for the interacting components via the connectors (and the tools) in SOFA.

RADL. (Reussner et al., 2003). Component types can be either basic or composite. Basic types are specified with interfaces. An interface can be either *provided*, offering methods to their environment, or *required*, making method-calls. Note however that unlike other ADLs, RADL constrains each basic type to have at least one required and one provided interfaces.

Composite component types are specified with interfaces and also a computation. A computation herein describes a configuration of sub component instances whose interfaces are connected to each other via *bindings*. Besides bindings, RADL offers *mappings* too, which allows an interface of a sub component to be connected with an interface of the composite component (if both interfaces have the same type). *Connector Support.* Like Darwin, RADL does not offer first-class connector elements. One can only specify simple communication links via *bindings* to connect sub components of composite component types. However, complex interaction protocols for their sub components cannot be specified explicitly.

AADL. (Feiler et al., 2006). Just like MetaH, AADL does not provide a generic type for specifying component abstractions. Instead, component types are categorised into three groups, each consisting of a collection of component types which can be instantiated by designers to specify their system architectures. For specifying a software architecture, component types can be either *(i)* thread, *(ii)* thread group, *(iii)* process, *(iv)* data, or *(v)* subprogram. For specifying a hardware architecture, component types can be *(i)* processor, *(ii)* memory, *(iii)* device, or *(iv)* bus. Lastly, for specifying composite units of the above-mentioned components, component type can then be a system type only.

Component types under these categories are essentially specified with interfaces. A component interface has either *ports* or *subprogram calls*, where ports serve for asynchronous events and data communications, and, subprogram calls for two-way synchronous method communications. A component type in AADL can also include the *(i) extends* functionality for inheriting from other types and *(ii) properties* functionality for specifying non-functional requirements. Furthermore, component types can have a computation that is specified with *(i) subcomponents*, *(ii) calls* and *connections* to specify interactions between subcomponents, *(iii) extends* to inherit

from another implementation, and *(iv) properties* to specify non-functional requirements for the subcomponents.
Connector Support. AADL does not offer first-class connector elements. However, it provides a pre-defined collection of interaction mechanisms: *(i)* port connections, *(ii)* component access connections, *(iii)* subprogram calls, and, *(iv)* parameter connections.

Port connections are concerned with the interactions through component ports by sending or receiving data/events asynchronously. Component access connections are employed when a shared data is to be accessed by components. As to subprogram calls and parameter connections, they relate to synchronous interaction between components through subprogram calls.

Archface. (Ubayashi et al., 2010). Archface is inspired from Aspect Oriented Programming (Kiczales and Hilsdale, 2001), offering notations for e.g., pointcut and advice. A component is specified with interfaces, each representing a single method of communication that is either required from the component environment or provided to its environment. Furthermore, component specifications can include *pointcut* declarations for the interface methods, such as "call (method call), execution (method execution), and cflow (control flow)".

Composite components are not supported by Archface. Configurations of components are specified via the *architecture* construct.

Connector support Connectors in Archface are first-class elements, which are specified as a collection of connections between the required and provided interfaces of some interacting components. Each connection has an *advice* that is specified for the provided method of the connection. The advice for the provided method describes that whenever the pointcut of the provided method is satisfied, this is followed by the inter-connected required method whose pointcut is expected to be satisfied subsequently. Note however that complex interaction protocols for components (i.e., the order of method-calls or method executions) cannot be specified using Archface connectors.

3 REALISABILITY

The issue of realisability essentially originates from the work of Allen and Garlan on formalising connectors (Allen and Garlan, 1997). They define the architectural connectors in terms of *(i) roles* and *(ii) a glue*. A role is defined for each component that

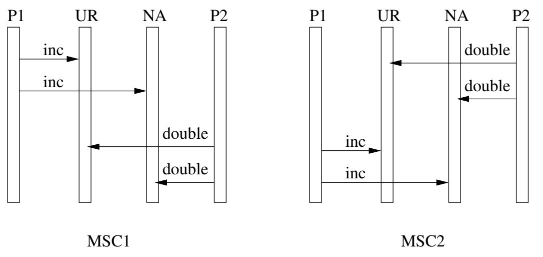

Figure 1: A nuclear power plant's (unrealisable) MSCs (Alur et al., 2003).

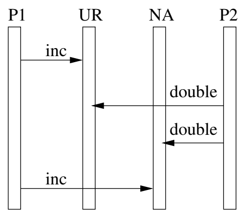

Figure 2: An unavoidable bad behaviour in the nuclear plant (Alur et al., 2003).

the connector coordinates; and it describes the interaction behaviour of the component in terms of a sequence of messages sent/received. The glue of a connector choreographs the component role behaviours by describing a global sequence(s) of messages sent/received by the components. However, software architectures with global constraint specifications (i.e., the glue) may not always be realised in the same way it is specified (i.e., with the same components, their behaviours, and their configuration). This is because global constraints cannot be imposed on distributed system components in reality as the distributed components are autonomous entities and can only observe their own local states - the global system state is unknown to them. To realise the global constraint specifications during implementation, one will possibly need to use a controller that controls the distributed component behaviours to ensure the global constraint. This will however turn the distributed system into a centralised one and violate the design decisions specified for the distributed nature of the system and invalidate any analyses made over the design decisions.

To illustrate realisability, consider the Nuclear Power Plant system (Alur et al., 2003), which consists of two clients (P1 and P2) whose goal is to access and modify the Nitric Acid (NA) and Uranium (UR) data controlled by the nuclear power plant. In the plant, the quantities of UR and NA need to be the same at all times. The clients P1 and P2 respectively increase and double these quantities and to ensure the plant's safety they need to strictly follow the protocol described by the message sequence charts of Figure 1. However the protocol in Figure 1 was proved to be

Table 1: The ALs that ignore complex connectors (Note: * represents the languages that use some other notations for specifying interaction protocols).

Architectural Languages
C2
Darwin
Rapide*
MetaH
UniCon
Koala*
LEDA
SOFA*
RADL
AADL
Archface

Table 2: The ALs that support complex connectors.

Architectural languages
Wright
COSA
XADL
CONNECT

unrealisable. It cannot be realised in a decentralised manner so that bad behaviours like the one in Figure 2 are avoided (Alur et al., 2003).

Nevertheless, Allen and Garlan's approach for architectural connectors defined in the nineties may have inspired many other languages in their definition of connectors, or more generally component interactions. Table 1 and Table 2 show respectively the analysed languages in Section 2 that have been find out to ignore complex connectors and support complex connectors. Due to the space restriction again, I focus on these ALs for realisability, especially those that support the specification of interaction protocol constraints using complex connectors or another notation.

Wright. Allen and Garlan has used their connector definition in their Wright architecture description language (Allen and Garlan, 1997). So, software architecture specifications in Wright may not be realisable in distributed cases due to the allowance for the global constraints via the connectors.

XADL. Realisability may be a concern for XADL (Dashofy et al., 2002) when an extended schema adopts the features of connector-centric ADLs such as Wright. If an extended form of connector types allows for a glue construct to coordinate the behaviours of the components, this would naturally lead to potential unrealisability.

COSA. Like Wright, COSA (Oussalah et al., 2004) enforces a glue in connector specifications, through which protocols of interaction among components are specified. Thus, COSA too allows potentially unrealisable specifications.

CONNECT. Just like Wright specifications, CONNECT (Issarny et al., 2011) specifications are potentially unrealisable due to the requirement for a *glue* in connector specifications.

Rapide. Rapide (Luckham, 1996) offers *architectural constraints*, which are essentially global constraints imposed on the interaction of the components within *architecture* specifications. These constraints are intended to coordinate the actions taken by the components, ensuring their compliance to particular global ordering of actions. Therefore, Rapide architectural constraints serve just as Wright *glues* and allow potentially unrealisable specifications.

Koala. In Koala (van Ommering et al., 2000), the modules specified within composite components act as glues, coordinating the sub components of the composite components. So, this can cause unrealisable specifications if the sub components are distributed.

SOFA. In SOFA (Plasil and Visnovsky, 2002; Bures et al., 2006), the protocol behaviour of a component computation can essentially impose a global constraint on the configuration of components. As aforementioned, global constraints lead to system specifications that cannot always be realised in a distributed manner.

4 DISCUSSIONS & CONCLUSION

Software architectures have been so popular since the nineties as a high-level design method for specifying software systems in terms of components and their relationships (i.e., connectors). Since then, there has been an ever-increasing attempt towards developing architectural languages. Today, there are more than 120 known architectural languages (ALs) for specifying software architectures at varying levels of details and then performing many useful operations such as formal analysis and implementation code generation (Ozkaya, 2016a). However, ALs did not gain the expected momentum unfortunately, and most of them remain unused by the practitioners in industry (Malavolta et al., 2012; Ozkaya, 2016b).

To better understand the ALs and compare them with each other, I analysed the existing 120 languages for a set of language requirements determined by Lago etal (Lago et al., 2015), which are grouped as

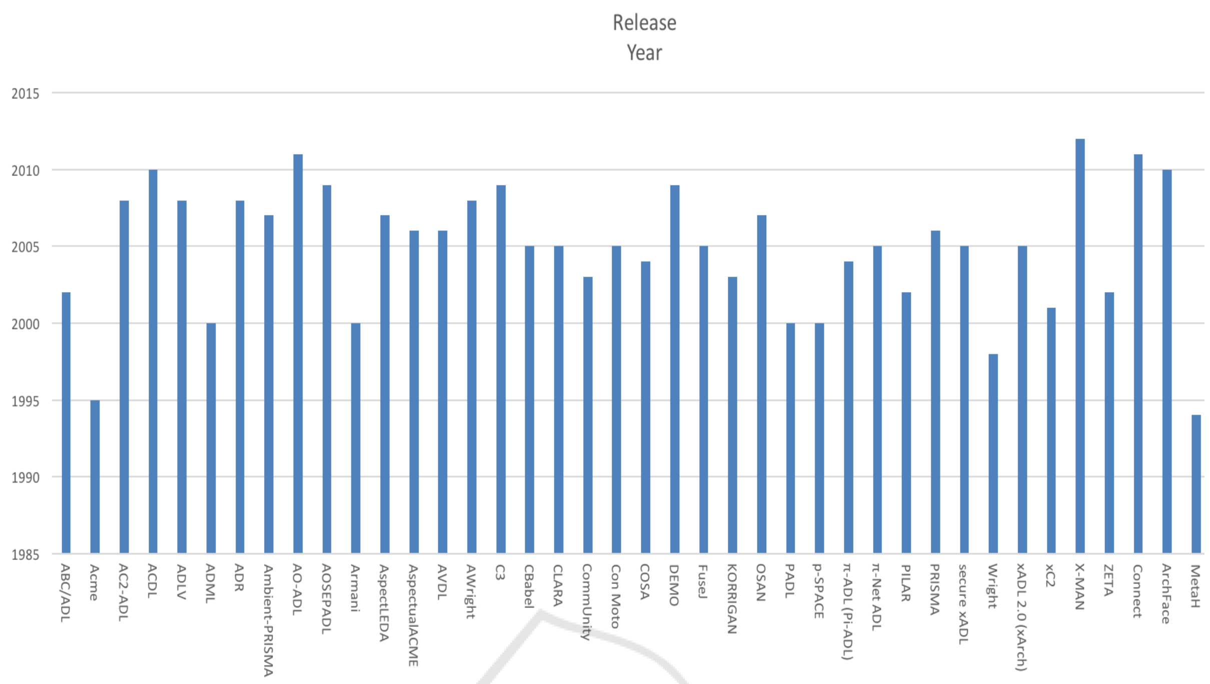

Figure 3: The ALs that support complex connectors.

language definition, language features, and tool support. The analysis results (Ozkaya, 2016a) showed for each AL the level of support provided for the language requirements in question and the relationships between the requirements. In this paper, I focus on two of the requirements analysed, which are *(i)* complex connector support (i.e., interaction protocols) and *(ii)* the realisability of software architectures. By doing so, the goal is to determine whether the existing ALs allow for the realisable specification of software architectures in terms of first-class components and complex connectors (i.e., interaction protocols).

Among the 120 ALs I analysed, only 37 of them depicted in Figure 3 support complex connectors through which interaction protocols can be separated from components. The rest of the languages in (Ozkaya, 2016a) either *(i)* ignore interaction protocols or *(ii)* allow designers to inject interaction protocols within component specifications. However, ignoring interaction protocols can lead to architectural mismatch (Garlan et al., 1995), i.e., the inability to compose seemingly compatible components due to wrong assumptions these make about their interaction. If the interaction protocols can be injected inside components, components may not then be easily re-used in different contexts - they become protocol dependent. This will also hinder the understandability of the system computations independently from the system interactions. The analysis of system components in isolation will be hindered too, and one may not easily understand during the analysis whether a design error is due to a wrong computation or wrong interaction protocol.

While complex connectors aid in the modular separation of interaction protocols from components in software architectures, the interaction protocol specifications can sometimes be dangerous for practitioners. Languages with complex connector support allow practitioners to specify interaction protocols that can globally constrain the interacting components (via the glue element of connectors) [2]. However, such global constraints cannot be guaranteed for distributed components, which are autonomous entities and can only know their own local state - global state is not visible to them. Therefore, software architecture specifications with complex connectors cannot be realised for distributed systems. To realise the global constraints, practitioners will need to add a new controller component which will make their system centralised however. In such a case, any analyses (e.g., performance) that have been made on the unrealisable specification will all be invalid as the analysed software architecture specification can no longer be used as it is for the implementation.

[2]Note that a few languages (e.g., Rapide (Luckham, 1996), Koala (van Ommering et al., 2000), and SOFA (Plasil and Visnovsky, 2002)) allow the specification of global constraints in different ways (using different notations).

None of the analysed languages addresses the unrealisability of the connector-centric architectural languages - except the XCD language (Ozkaya and Kloukinas, 2014) that has been introduced very recently. In XCD, while connectors are used for specifying interaction protocols, all protocol constraints are locally imposed on the components - global constraints are not possible. An XCD connector consists structurally of roles, where each role describe the local interaction protocol for a participating component. If a global constraint is required, this can only be specified via a controller component that sits among other components and coordinates their interaction according to a global constraint.

ACKNOWLEDGEMENTS

This work has been supported by the project of the Scientific and Technological Research Council of Turkey (TUBITAK), Grant No: 215E159.

REFERENCES

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249.

Alur, R., Etessami, K., and Yannakakis, M. (2003). Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633.

Binns, P., Englehart, M., Jackson, M., and Vestal, S. (1996). Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227.

Bures, T. (2005). Automated synthesis of connectors for heterogeneous deployment. Tech. report no. 2005/4, Dep. of SW Engineering, Charles University, Prague.

Bures, T., Hnetynka, P., and Plasil, F. (2006). Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA*, pages 40–48. IEEE Computer Society.

Bures, T. and Plasil, F. (2004). Communication style driven connector configurations. In Ramamoorthy, C., Lee, R., and Lee, K., editors, *Software Engineering Research and Applications*, volume 3026 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin Heidelberg.

Canal, C., Pimentel, E., and Troya, J. M. (1999). Specification and refinement of dynamic software architectures. In Donohoe, P., editor, *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer.

Clements, P. C., Garlan, D., Little, R., Nord, R. L., and Stafford, J. A. (2003). Documenting software architectures: Views and beyond. In Clarke, L. A., Dillon, L., and Tichy, W. F., editors, *ICSE*, pages 740–741. IEEE Computer Society.

Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). An infrastructure for the rapid development of xml-based architecture description languages. In Tracz, W., Young, M., and Magee, J., editors, *ICSE*, pages 266–276. ACM.

Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute.

Galik, O. and Bures, T. (2005). Generating connectors for heterogeneous deployment. In Nitto, E. D. and Murphy, A. L., editors, *SEM*, pages 54–61. ACM.

Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185.

Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical report, Pittsburgh, PA, USA.

Holzmann, G. J. (2004). *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

Issarny, V., Bennaceur, A., and Bromberg, Y.-D. (2011). Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In Bernardo, M. and Issarny, V., editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 217–255. Springer.

Kiczales, G. and Hilsdale, E. (2001). Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26(5):313–.

Kloukinas, C. and Ozkaya, M. (2012). Xcd - modular, realizable software architectures. In Pasareanu, C. S. and Salaün, G., editors, *Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers*, volume 7684 of *Lecture Notes in Computer Science*, pages 152–169. Springer.

Lago, P., Malavolta, I., Muccini, H., Pelliccione, P., and Tang, A. (2015). The road ahead for architectural languages. *IEEE Software*, 32(1):98–105.

Luckham, D. C. (1996). Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford University, Stanford, CA, USA.

Magee, J. and Kramer, J. (1996). Dynamic structure in software architectures. In *SIGSOFT FSE*, pages 3–14.

Magee, J., Kramer, J., and Giannakopoulou, D. (1997). Analysing the behaviour of distributed software architectures: a case study. In *FTDCS*, pages 240–247. IEEE Computer Society.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2012). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 99.

Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. (1996). Using object-oriented typing to support architectural design in the c2 style. In *SIGSOFT FSE*, pages 24–32.

Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93.

Oussalah, M., Smeda, A., and Khammaci, T. (2004). An explicit definition of connectors for component-based software architecture. In *ECBS*, pages 44–51. IEEE Computer Society.

Ozkaya, M. (2016a). Analysis of architectural languages. https://sites.google.com/site/ozkayamert1/als. Online; accessed: 2016-12-17.

Ozkaya, M. (2016b). What is software architecture to practitioners: A survey. In Hammoudi, S., Pires, L. F., Selic, B., and Desfray, P., editors, *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016.*, pages 677–686. SciTePress.

Ozkaya, M. and Kloukinas, C. (2014). Design-by-contract for reusable components and realizable architectures. In Seinturier, L., de Almeida, E. S., and Carlson, J., editors, *CBSE'14, Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (part of CompArch 2014), Marcq-en-Baroeul, Lille, France, June 30 - July 4, 2014*, pages 129–138. ACM.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52.

Plasil, F. and Visnovsky, S. (2002). Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076.

Reussner, R., Poernomo, I., and Schmidt, H. (2003). Reasoning about Software Architectures with Contractually Specified Components. In Cechich, A., Piattini, M., and Vallecillo, A., editors, *Component-Based Software Quality*, volume 2693 of *Lecture Notes in Computer Science*, page 287–325. Springer Berlin Heidelberg.

Rumbaugh, J. E., Jacobson, I., and Booch, G. (2005). *The unified modeling language reference manuel - covers UML 2.0, Second Edition*. Addison Wesley object technology series. Addison-Wesley.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. *IEEE Trans. Software Eng.*, 21(4):314–335.

Taylor, R. N., Medvidovic, N., Anderson, K. M., Jr., E. J. W., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. (1996). A component- and message-based architectural style for gui software. *IEEE Trans. Software Eng.*, 22(6):390–406.

Ubayashi, N., Nomura, J., and Tamai, T. (2010). Archface: A contract place where architectural design and code meet together. In Kramer, J., Bishop, J., Devanbu, P. T., and Uchitel, S., editors, *ICSE*, pages 75–84. ACM.

van Ommering, R. C., van der Linden, F., Kramer, J., and Magee, J. (2000). The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85.