Automating the Building of Software Component Architectures

Nicolas Desnos, Christelle Urtado, Sylvain Vauttier, Marianne Huchard

Automating the Building of Software Component Architectures

Nicolas Desnos[1], Sylvain Vauttier[1], Christelle Urtado[1], and Marianne Huchard[2]

[1] LGI2P / Ecole des Mines d'Alès - Parc scientifique G. Besse - 30 035 Nîmes, France
{Nicolas.Desnos, Sylvain.Vauttier, Christelle.Urtado}@site-eerie.ema.fr

[2] LIRMM - UMR 5506 - CNRS and Univ. Montpellier 2
34 392 Montpellier cedex 05, France - huchard@lirmm.fr

Abstract. Assembling software components into an architecture is a difficult task because of its combinatorial complexity. There is thus a need for automating this building process, either to assist architects at design time or to manage the self-assembly of components at runtime. This paper proposes an automatic architecture building process that uses ports, and more precisely composite ports, to manage the connection of components. Our solution extends the Fractal component model. It has been implemented and experiments have been run to verify its good time performance, thanks to several optimization heuristics and strategies.

1 Introduction and Motivation

Software engineering aims at optimizing the cost of design and maintenance while preserving both the quality and reliability of the produced software. Component-based development techniques try to enhance reuse [1–3]. The design process of an application is led by an architect and decomposes into three steps: he selects components, defines an architecture by assembling them[3] and then uses a tool to control the consistency of the assembly to determine if the assembled components are compatible. Components are generally described as a set of interfaces that define what a component can provide and must require. Component assemblies are then built by connecting component interfaces together [4–8].

Most existing works do not provide architects with any guidance during the selection and assembly steps. They rather focus on checking the validity of a previously built architecture [6, 9–12]. The consistency check techniques cannot be used in an iterative building process because of the combinatorial complexity [13]. To guide the architect, we propose an efficient approach to automatically build potentially valid architectures. It produces a reduced set of preselected component assemblies on which it is relevant to perform checks to find valid architectures. It relies on the use of ports, and more precisely of composite ports, to describe known usages of components. A construction algorithm has been successfully implemented and experimented in the Fractal component model [7].

[3] In these works, we will consider that the selected components need no adaptation (or might have already been adapted).

The remainder of this paper is organized as follows. Section 2 discusses the issues raised by the building of valid architectures and introduces a component model which features primitive and composite ports. Section 3 describes a basic algorithm to automatically build architectures along with its optimizations. Section 4 concludes and draws perspectives.

2 Building Valid Architectures

2.1 An Augmented Component Model to Ease Construction

Not to start from scratch, we choose to extend an existing component model named Fractal [7][4]. Classically, a Fractal component is described as a black box that defines the services the component provides and requires through server and client **interfaces** and a content (called the **architecture**) that allows a component to be recursively described. Fractal components are assembled into architectures by connecting client interfaces to server interfaces. This allows components to collaborate by exchanging messages along these connections.

The Fractal model is first extended with ports. As in UML2 [4], ports are used to group together the client and server interfaces that are used by a component in a given collaboration. Ports are thus used to specify various usage contexts for components. We define two kinds of ports. **Primitive ports** are composed of interfaces, as in many other component models [4, 6, 10, 12, 14]. **Composite ports** are composed of other ports. Composite ports are introduced to structurally represent complex collaborations. Figure 1 shows an architecture where *ATM* is an example of component, *Question* one of its provided interfaces, *Transaction* one of its required interfaces and *Money_Withdraw* its composite port which is composed of the two *Money_Dialogue* and *Money_Transaction* primitive ports. Two primitive ports are connected together when all the interfaces of the first port are connected to interfaces of the second port (and reciprocally). A composite port is connected when all the primitive ports it is composed of (directly or indirectly) are connected. Component architectures can then be built by connecting together component ports (what entails interface connections). Next section details how ports, and more precisely composite ports, make the building of architectures easier.

2.2 Validity of an Architecture

An architecture is said to be valid if it is both correct and complete.

Correctness. Stating the correctness of an architecture relies on techniques that verify the coherence of connections, to check whether they correspond to

[4] We choose Fractal mainly because it is a hierarchical composition model that supports component sharing, its structure is simple but extensible and respects the separation of concerns principle and an open-source implementation exists.

possible collaborations between the linked components. These verifications use various kinds of meta-information (types, protocols, assertions, etc.) associated with various structures (interfaces, contracts, ports, etc.).

A first level of correctness, called **syntactic correctness**, can be verified by comparing the types of the connected interfaces [5, 7]. This ensures that components can "interact" because the signatures of the functionalities to be called through the required interface match the signatures of the functionalities of the provided interface. A second level of correctness, called **semantic correctness** [15, 9], can then be verified to determine if the connected components can "collaborate" i.e. exchange sequences of messages that are coherent with each other's behavior. Semantic verifications require that **protocols** – valid sequences of messages – be defined. The semantic correctness of the connection between two ports is handled as a classic comparison of their associated protocols. This is a time-consuming process because of the highly combinatorial complexity of the algorithms used to compare all the possible message sequences [13].

Completeness. A component architecture is built to achieve some **functional objectives** [1, 15, 16]. Functional objectives are defined as a set of functionalities to be executed on selected components. The set of connections in the architecture must be sufficient to allow the execution of collaborations that reach (include) all the functional objectives. Such an architecture is said to be **complete**.

Starting from a set of components corresponding to the functional objectives, a naive algorithm can be to try to build an architecture where all the interfaces of all the components are connected, so that all the execution scenarios may be executed. When no solution exists in the current architecture to connect an interface, the repository is searched for a component that has a compatible interface. If one exists, it is added to the architecture and the interfaces are connected. If several connections are possible, they represent alternative building paths to be explored. In case a dead end is reached, the construction is backtracked to a previous configuration, in order to try alternative connection combinations. The problem with this building process is the size of the solution space to be explored. It is amplified by the cost of the semantic verifications that must be calculated for any candidate connection between two components. Therefore, the automatic construction of valid architectures still is an open problem. We then have studied different ways to reduce the complexity of the building process.

3 Taming the Complexity of Automation

3.1 Using Composite Ports to Connect Components

To reduce the complexity, the building process can try to connect only the interfaces that are useful to reach the functional objectives. However, the proper use of a functionality of a component is not independent from other functionalities. The behavior protocol of a component specifies the different valid execution scenarios where a functionality is called. The execution of a scenario requires

the connection of all the interfaces that it uses: regarding the scenario, these interfaces are said to be **dependent**. Thus, a given functional objective can be reached only when precise sets of (dependent) interfaces, corresponding to valid scenarios, are connected. An analysis of the behavior protocol of a component could be used to determine those scenarios but a means is required to capture and to express this information in an explicit and simple way, in order to ease the connection process. Ports are introduced as a kind of structural meta-information, complementary to interfaces, that group together the interfaces of a component corresponding to a given valid scenario. Ports could be produced automatically, by the analysis of behavior protocols or be manually added by the designer in order to document a given usage of the component.

Port connections make the building process more abstract (port-to-port connections) and more efficient (no useless connections). Considering a port that needs to be connected, the availability of a compatible port is an important issue. The more numerous interfaces are in a given port, the more specific the port type is and the less chances exist to find compatible ports. Composite ports are used to solve this issue: they allow short scenarios, composed of few interfaces, to be described as small primitive ports that are then composed together to describe more complex scenarios. Large flat primitive ports can then be replaced by small primitive ports hierarchically structured into larger composite ports. The result is that smaller ports are less specialized and thus provide more connection possibilities. From a different point of view, a primitive port can be considered as the expression of a constraint to connect a set of interfaces both at the same time and to a unique component. A composite port is the expression of a constraint to connect a set of interfaces at the same time but possibly to different components. As they relax constraints, composite ports increase the amount of possible connection combinations. Moreover, composite ports provide a means to precisely specify how interfaces must be connected: to a unique component – for functionality calls to produce cumulative effects – or to distinct components.

3.2 Building Quasi-valid Architectures

Semantic verifications are very expensive. Our approach keeps semantic verifications separated from the building process so as not to waste time verifying the semantics of connections as long as the completeness of the architecture cannot be guaranteed. To achieve this, a **quasi-valid** architecture is first built. A quasi-valid architecture is a syntactically correct and complete architecture. The connection of a port enforces the completeness of an architecture, regarding the execution of a scenario. Once all the ports corresponding to the functional objectives are connected, an architecture is quasi-valid. Quasi-validity is a precondition for an architecture to be valid.

We wrote an algorithm that automatically builds quasi-valid architectures. The building process uses a set containing the ports that still have to be connected – the functional objective set (FO-set). The FO-set contains only primitive ports: composite ports are systematically decomposed into the set of primitive ports they are directly or indirectly composed of. The FO-set is initialized

with the ports that correspond to the functional objectives. One of the primitive ports is picked up from the FO-set and a compatible port is searched for. If a compatible unconnected port is found, the ports are connected together. If the compatible port belongs to a component that does not yet belong to the architecture, the component is added to the architecture. If the chosen compatible port belongs to a composite port, all the other primitive ports that composed the composite port are added to the FO-set. This way, no port dependencies – and therefore no interface dependencies – are left unsatisfied. The building process is iterated until the FO-set is empty. All the initial primitive ports that represent functional objectives are then connected along with all ports they are recursively dependent upon: the resulting architecture is quasi-valid.

Figure 1 shows the example of an architecture built by our algorithm. It starts with a FO-set that contains the *Money_Withdraw* primitive port of the *Client* component. This port is taken out of the FO-set and a connection is searched for. It is connected to the compatible *Money_Dialogue* primitive port of the *ATM* component. As this latter port belongs to the *Money_Withdraw* composite port, it depends on the *Money_Transaction* primitive port which is thus added to the FO-set before the building process iterates. The *Money_Transaction* primitive port of the *ATM* component is now considered for connection. It is compatible with the *Money_Transaction* primitive port of the *Bank* component which belongs to the composite port *Money_Withdraw*. After connection, the other primitive port of this composite port, *Request_Data*, is in turn added to the FO-set. At the next iteration, the *Request_Data* primitive port of the *Bank* component is connected with the compatible primitive port *Provide_Data* of the *Database* component. As this primitive port does not belong to a composite port, no primitive port is to be added to the FO-set. The FO-set is now empty: the architecture of Fig.1 is quasi-valid.

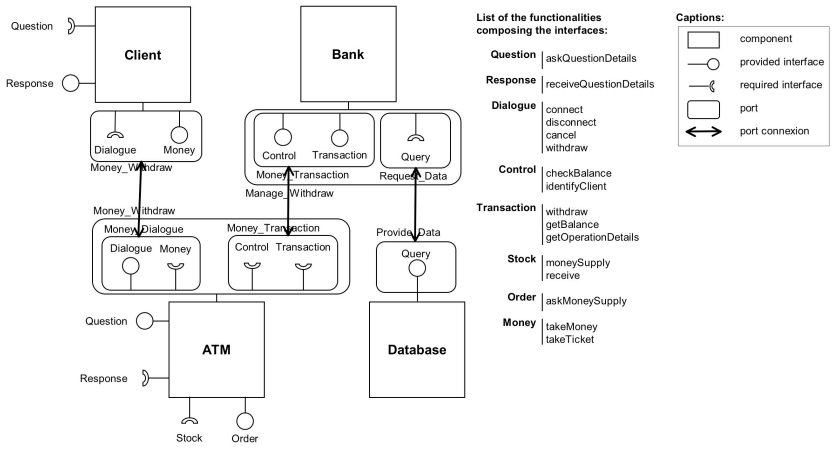

Fig. 1. A quasi-valid architecture built with the support of composite ports

Several special situations can occur during this process. When several free compatible ports are candidate for connection, they correspond to alternate solutions that are to be explored. Conversely, when no free compatible port is found the building algorithm has reached a dead end. The construction is then backtracked to a previous situation where unexplored connection possibilities exist. Our algorithm is implemented as the searching of a construction tree using a depth-first policy. Breadth search is used to explore all the alternate construction paths. This complete exploration of the construction tree is used to guarantee that any possible solution is always found.

3.3 Strategies, Heuristics and Experiments

The performance of the building algorithm has been measured. For this purpose, we have implemented a small environment that generates random component sets which provide different building contexts, in size and complexity. Once a component set is generated, an arbitrary number of ports can be chosen as functional objectives and the building algorithm be launched. Our experiments show that the combinatorial complexity of the building process is very high. To be able to use our approach in demanding situations, such as the deployment and configuration of components at runtime, we have studied various heuristics that speed up the building process.

Building Minimal Architectures. A first strategy is to try to find not all the possible architectures but only the most interesting ones. Minimality is an interesting metrics for the quality of an architecture [17]. We apply this minimality criterion to the number of connection. Less connections entail less semantic verifications, less interactions and therefore less conflict risks. Less connections also entail more evolution capabilities (free ports). To efficiently search for minimal architectures, we have added a branch-and-bound strategy to our building algorithm. The bound is the maximum number of connections allowed for the construction of the architecture. When this maximum is reached when exploring a branch of the construction tree, the rest of the branch can be discarded as any new solution will be less optimal than the previously found (pruning).

Min Domain Heuristic. This heuristic is used to efficiently choose ports from the FO-set. The port for which a minimum of free compatible ports exists is chosen first. This minimizes the effort to try all the connection possibilities: in case of repeated failures, this allows impossible constructions to be detected sooner.

Minimum Effort Heuristic. In the branch-and-bound strategy, every time the bound is lowered, the traversal of the tree is speeded up. To connect a primitive port, the algorithm first chooses the free compatible primitive port that belongs to the "smallest" composite port. It corresponds to the choice of the less dependent ports, that minimize future efforts to connect them.

No New Dependency Heuristic. When a compatible port can be found in the FO-set its connection will add no new dependency, and furthermore, satisfy two dependencies at once. Indeed, when a port belongs to the FO-set, the other primitive ports it depends on are already in the FO-set.

Look-ahead Strategy. Calculi can be used to predict if the traversal of the current construction branch can lead to a minimal solution. They are based on an estimate of the minimum number of connections required to complete the building. A soon as the sum of the existing connections with this estimate is greater than the bound, the current branch can be pruned. A simple example of this estimate is the number of ports in the FO-set divided by two.

Experimental Results: an Outline. Experiments show that performance mainly depends on the number of initial functional objectives. This is logical since more functional objectives implies not only a larger search space but also more constraints, thus more failures and backtracks. For example, series of experiments have been run with a library of 38 generated components. Each component had at most 4 primitive ports and at most 2 composite ports. Each primitive port had at most 5 interfaces. Starting with 5 initial functional objectives, the following typical results are obtained. A basic construction algorithm, implemented in Java and executed on a standard computer, without any of the above optimizations, is able to find 325 000 quasi-valid architectures, when stopped after 15 hours. This gives an idea of the gigantic size of the search space. Among those quasi-valid architectures, the largest ones are composed of 48 connections. The smallest architecture found is composed of 18 connections. As a comparison, the optimized construction algorithm finds the only minimal architecture composed of 7 connections in less than a second. This motivates our proposal for an efficient building approach. It is difficult to build quasi-valid architectures, because the more frequent ones are rather large (around 40 connections in the above example). It is even more difficult to build minimal ones, because they are scarce in a large search space.

4 Conclusion and Perspectives

While other works focus on the validation of complete architectures, our work studies the building process of architectures and proposes a practical solution to automate it. It enables the candidate architectures, on which validation algorithms are to be applied, to be systematically searched for. Besides the many optimization strategies and heuristics used for the traversal of the construction space, the use of ports, and particularly of composite ports, is prominent in our approach. As they express the dependencies that exist between interfaces, ports provide a simple means to evaluate the completeness of an architecture. Finally, being composed of interfaces, they provide means to abstract the many connections of interfaces to single connections and thus reducing the combinatorial complexity of the building.

A perspective for this work is to integrate it to a component-based development framework, for example as part of a trading service, to provide a means to manage the self-assembling of components in open, dynamic systems (autonomic computing).

References

1. Crnkovic, I.: Component-based software engineering - new challenges in software development. Software Focus (2001)
2. Garlan, D.: Software Architecture: a Roadmap. In: The Future of Software Engineering. ACM Press (2000) 91–101
3. Brown, A.W., Wallnau, K.C.: The current state of CBSE. IEEE Software **15**(5) (1998) 37–46
4. OMG: Unified modeling language: Superstructure, version 2.0 (2002) http://www.omg.org/uml/.
5. OMG: Corba components, version 3.0, http://www.omg.org/docs/formal/02-06-65.pdf (2002)
6. Traverson, B.: Abstract model of contract-based component assembly (2003) ACCORD RNTL project number 4 deliverable (in french).
7. Bruneton, E., Coupaye, T., Stefani, J.: Fractal specification - v 2.0.3 (2004) http://fractal.objectweb.org/specification/index.html.
8. Plásil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: Proceedings of the Int. Conf. on Configurable Distributed Systems, Washington, DC, USA, IEEE Computer Society (1998) 43–52
9. Plásil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28**(11) (2002) 1056–1076
10. Hacklinger, F.: Java/A - Taking Components into Java. In: IASSE. (2004) 163–168
11. Farías, A., Sudholt, M.: On components with explicit protocols satisfying a notion of correctness by construction. In Meersman, R., Tari, Z., et al., eds.: On the Move to Meaningful Internet Systems: Int. Conf. CoopIS, DOA, and ODBASE Proc. Volume 2519 of LNCS., Springer (2002) 995–1012
12. de Boer, F.S., Jacob, J.F., Bonsangue, M.M.: The OMEGA component model. Deliverable of the IST-2001-33522 OMEGA project (2002)
13. Inverardi, P., Wolf, A.L., Yankelevich, D.: Static checking of system behaviors using derived component assumptions. ACM Trans. Softw. Eng. Methodol. **9**(3) (2000) 239–272
14. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: Proceedings of ICSE, Orlando, Florida, USA, ACM Press (2002) 187–197
15. Dijkman, R.M., Almeida, J.P.A., Quartel, D.A.: Verifying the correctness of component-based applications that support business processes. In Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K., eds.: Proc. of the 6th Workshop on CBSE: Automated Reasoning and Prediction, Portland, Oregon, USA (2003) 43–48
16. Inverardi, P., Tivoli, M.: Software Architecture for Correct Components Assembly. In: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture. Volume 2804 of LNCS. Springer (2003) 92–121
17. Cechich, A., Piattini, M., Vallecillo, A., eds.: Component-Based Software Quality: Methods and Techniques. Volume 2693 of LNCS. Springer (2003)