Architecture-Centric Design of Complex Message-Based Service Systems

Christoph Dorn, Philipp Waibel, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology, Austria
{dorn,dustdar}@infosys.tuwien.ac.at, philipp.waibel@gmail.com

Abstract. Complex, message-based service systems discourage central execution control, require extremely loose coupling, have to cope with unpredictable availability of individual (composite) services, and may experience a dynamically changing number of service instances. At the topmost level, the architecture of such a complex system often follows a messaging style most naturally. A major problem during the design of these systems is achieving an overall consistent configuration (i.e, ensuring intended message routing across producers, consumers, and brokers). While orchestration or choreography-based approaches support the design of individual composite services along a workflow-centric paradigm, they are an awkward fit for specifying a message-centric architecture. In this paper, we present an architecture-centric approach to designing complex service systems. Specifically we propose modeling the system's high-level architecture with an architecture description language (ADL). The ADL captures the message-centric configuration which subsequently allows for consistency checking. An architecture-to-configuration transformation ensures that the individual deployed services follow the architecture without having to rely on a central coordinator at runtime. Utilizing our provided tool support, we demonstrate the successful application of our methodology on a real world service system.

Keywords: Decentralized Composite Services, Architecture Description Language, Consistency Checking, Message-based Style.

1 Introduction

The last two decades have witnessed the emergence of various techniques for composing complex service systems. Composition approaches based on orchestration languages such as BPEL [12] and YAWL [1] or those based on choreography languages such as WS-CDL[1] share a common assumption on the underlying system architecture: namely workflow-like control and data flow among services. Not all application scenarios, however, fit this workflow-centric scheme and hence existing approaches are cumbersome to apply. A publish-subscribe architecture is a better match for a complex service system which (i) discourages centralized

[1] Web Services Choreography Description Language
http://www.w3.org/TR/ws-cdl-10/

X. Franch et al. (Eds.): ICSOC 2014, LNCS 8831, pp. 184–198, 2014.

execution control, (ii) consumes and provides data rather than method invocations, (iii) experiences unpredictable service availability, and (iv) must support a dynamically changing number of service instances.

In this paper we address challenges emerging from the design and configuration efforts of such decentralized, highly decoupled, event-based (composite) services. A system architect following a naive approach would specify the individual (composite) services and wire them up in an ad-hoc manner via message queues. The resulting message flow might be documented somewhere but the overall consistency of the ultimately developed services and the deployed message brokers cannot be guaranteed. The ground truth message flow remains implicit in the configuration of individual services and the utilized message-oriented middleware (MOM). It is only a matter of time and complexity before the design and configuration of such a composite system becomes inconsistent. An engineer engaging in example tasks such as restructuring the message flow, integrating additional services, deploying additional instances, or adapting services has little means to ensure that a particular change leaves the updated system in a coherent state. Enterprise Application Integration (EAI) patterns [10] guide the architect in how to structure the overall system but cannot guarantee correct implementation. Consequently high costs occur in terms of time and invested resources when attempting to maintain consistency, as well as for detecting and repairing inconsistencies.

We propose to address this problem through a combination of architecture-centric composite service specification, separation of message routing aspects from local invocation-centric message processing, and architecture-to-configuration transformation. Specifically, our approach applies a component and connector view for describing the high-level, overall complex service system's architecture. The components represent individual, composite services while the connectors represent message brokers. The resulting centralized system architecture serves as the authoritative source for configuring the MOM and each service's publish/subscribe endpoints. Individual services leverage the advantages of proven technologies such as Enterprise Service Buses (ESB) and workflow engines for processing messages locally. This cleanly separates the responsibility of designing the overall, distributed architecture from designing its constituent components. Constraint checks ensure that the architecture itself is consistent. Ultimately transformations derive the actual technology configuration automatically from the architecture description and thus guarantee consistency.

In support of this approach, our contribution in this paper is four-fold. We provide (i) an message-centric extension for the Architecture Description Language xADL [5] (Sec. 5.2), (ii) message-centric architecture consistency checking (Sec. 5.3), (iii) tool support through extension of ArchStudio4 [4] (Sec. 6), and (iv) proof-of-concept architecture-to-configuration transformations for the ActiveMQ JMS server and the Mule ESB (Sec. 6.1). We applied our approach and techniques to an industry case study, demonstrating that our methodology is not only feasible, but also easily applicable in real world situations (Sec. 7).

2 Motivating Scenario

A parking management system consists of a high number of distributed services. Figure 1 depicts a simplified, typical system configuration. Data services at parking sites provide primarily static and highly dynamic information about the parking sites structure (e.g., structure layout, spots per vehicle type), properties (e.g., location, typical occupation level at a given time), the current capacity, and reserved spots. Filter services obtain these details and bring all messages to a uniform format for structural data and dynamic change events (i.e., EAI message translator pattern and the normalizer pattern). Aggregator services maintain a coherent, property-specific view on the parking structures. For example, one aggregator provides details on all parking structures in a particular region, another specializes on caravan parking. Ultimately, Point-of-Sales (POS) services serve particular business cases such as hotels, airports, train stations, rental-car companies, or car-sharing initiatives for reserving parking spots. These POS services obtain the structural data, and changes thereof from aggregator services but receive dynamic updates from filter services directly.

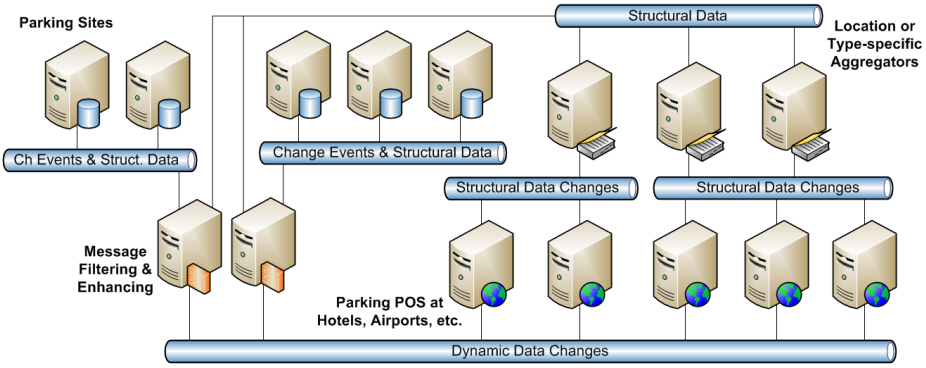

Fig. 1. Parking Management Complex Service System comprising, Data Services, Filter Services, Aggregator Services, and POS Services, as well as message brokers. (Note that icons are meant to depict services and not servers.)

This scenario reflects the challenges from the introduction. The overall system relies primarily on asynchronous message exchange. There is no single service that would logically serve as a central point of control. Individual service participants may be disconnected or briefly overloaded and thus temporarily unavailable. New services may be introduced anytime but must not affect the remaining service participants' interaction nor require their extensive reconfiguration. Our approach aims at preventing (respectively detecting) following example problems: a Data service publishing its updates to the wrong topic, respectively a Filter service reading from the wrong topic; an Aggregator service expecting an incompatible message type from a filter. Multiple POS services using a single,

shared reply queue for asynchronous requests, or a POS service connecting to a non-existing request topic.

3 Related Work

Choreography and Orchestration are the two main contemporary paradigms for addressing design and configuration of complex service systems. Orchestration languages such as BPEL [12], JOpera [13], or YAWL [1] represent centralized approaches and thus need a single coordinating entity (i.e., the workflow engine) at runtime. Decentralized orchestration approaches (e.g.,[11,16]) mitigate this shortcoming through distributing control flow among the participating services. While orchestration takes on a single process view including all participating services, choreography specification languages such as *BPEL4Chor* [6], *Let's Dance* [17], or *MAP* [3], on the other hand, aim at a holistic, overarching system view. Both choreography and orchestration, however, presume a workflow-like system style, with services playing fixed roles, and being highly available (respectively easily replaceable on the fly). It is rather cumbersome to model complex service systems that experience dynamically fluctuating service instances, multiple (a-priori unknown) instances of the same service type, and temporal unavailability with the languages and approaches outlined above. Our work caters predominately to systems that more naturally rely on one-way events and less on request/reply style information exchange. In addition, our approach offers more flexibility on where to locate and manage coordinating elements by strictly separating them from services concerned with business logic as well as modeling them as first class entities. Enterprise Application Integration patterns (EAI) [10] demonstrate the benefits of message-centric service interaction. Scheibler et al. [14] provide a framework for executing EAI-centric configurations; however, by means of a central workflow engine.

At no point are we suggesting that our approach is superior to any of these existing approaches, methodologies, or technologies. We rather see our work as focusing on different service system characteristics. We believe that integrating these existing technologies are well worth investigating as part of future work. This holds also true for existing research efforts that focus on other qualities than high-level architectural consistency. Work on integrating QoS or resource allocation is highly relevant but currently not applicable to our scenarios. Such approaches [7] typically rely on centralized control and/or exclusively employ the request/reply invocation pattern.

Our work takes inspiration from significant contributions in the software architecture domain. Zheng and Taylor couple architecture-implementation conformance with change management in their 1.x mapping methodology [18]. 1.x mapping focuses primarily on maintaining consistency between an architecture specification and its underlying Java implementation and how changes are propagated from the architect to the software developer. We follow a similar procedure by separating high-level architectural design and configuration decisions from the engineers that implement the actual (composite) services.

Garcia et al. investigate the issues of architecture-centric consistency bottom-up [9]. In contrast to our top-down specification of event handling, Garcia et al. identify message flows from source code of event-based systems implemented in Java or Scala. Their technique appears also very suitable for recovering the messaging architecture of an already existing complex service system.

Baresi et al. model publish/subscribe systems for rigorous verification [2]. Their approach requires modeling of a component's internal publishing/subscribing behavior in order to evaluate message reliability, ordering, filtering, priorities, and delays. On the one hand, we do not assume knowledge of precise service internal behavior at the architectural level, and on the other hand, such analysis is significantly more fine-grained than required for our purpose.

The SASSY framework [8] targets service system specification by domain experts through the Service Activity Schema (SAS) language. Inspired by BPMN, SAS provides OR, XOR, AND gateways, loops, activities, input/output elements, and external services for specifying the system's data and control flow. The resulting specification lacks first class connectors and primarily lends itself to workflow-style systems.

4 Approach

Our approach to designing and configuring a complex service system consists of four phases (depicted in Figure 2). First, a high-level architectural component and connector view identifies the main (composite) services (architecture-level components), and their interactions via messages (architecture-level interfaces). Explicit message channels (architecture-level connectors) enable the clear separation of interaction concerns from (service) logic concerns [15]. An architect may model connector-specific properties, configurations, simplify N:M links (become N:1:M), interaction monitoring, etc when connectors become first class model elements. In our specific context, the high-level architecture also separates the responsibility of the overall service system architect from engineers tasked with the internal design and wiring of the individual services (incl. applied tools such as ESBs). We apply an existing extensible Architecture Description Language (xADL) [5] for expressing the high-level architecture. Subsection 5.1 below provides a short introduction of xADL and its main modeling elements.

At any stage in the architecture modeling, the architect may choose to specify messaging-specific configuration properties. For connectors, these properties define messaging middleware-specific details such as channel name, applicable protocol, or deployment host. For interfaces, these properties include messaging endpoint related details such as reply channel references, event-centric request endpoint references, as well as framework-centric properties.

Upon triggering consistency checking, our algorithm iterates through all components, connectors, and links that exhibit messaging-specific configurations. It verifies allowed link cardinalities, missing configuration values, and matching interface details. For a detailed description of constraints see Subsection 5.3.

For all elements that passed these constraint checks, the system architect may then trigger architecture-to-configuration transformations. Distinct tool-centric

transformations exist for connectors and components. Connectors plus messaging-centric configuration become a message broker configuration; in our case a set of ActiveMQ configurations. Components plus messaging-centric configuration translate into Mule workflow skeletons (see Subsection 6.1).

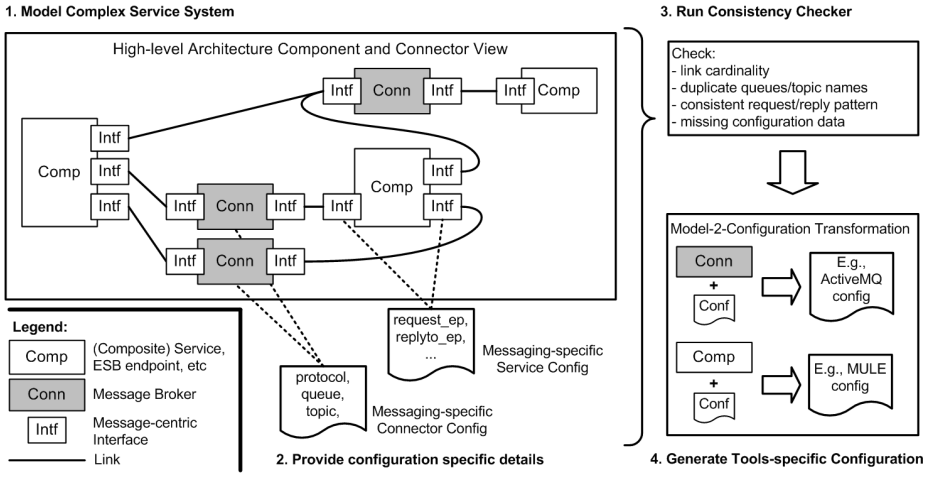

Fig. 2. A methodology for designing and configuring complex service systems

Figure 2 displays the four phases in a sequential manner. The system architect and her co-worker, however, will typically progress through these phases in an iterative manner. An initial configuration may sufficiently serve for checking overall consistency and for identifying core individual services. This approach addresses the four properties of complex service systems outlined earlier in the introduction:

No centralized execution control. The high-level system architecture constitutes a central, authoritative specification only at design-time. The top-down specification of message infrastructure and message-centric service endpoints ensures that the decentralized elements remain true to the architecture at runtime.

Publish/Subscribe interaction. The high-level composite services in this system are primarily concerned with their business logic and not how many sources they receive information from or how many destination in turn are interested in their processed information. Hence, an event-driven interaction style best reflects this loose coupling.

Unpredictable service availability. The message-oriented architecture enables reasoning on the effect of unavailable services. As a durable subscriber, individual services may process at their own pace without affecting simultaneous subscribers. Explicit connector modeling also enable reasoning on where to host which channels, further decoupling message routing from processing.

Dynamically fluctuating service instances the system architect may specify in the architecture that there may exist multiple instances of particular composite service types, and control their impact on the overall system through selection of publish-subscribe versus point-to-point channel connectors.

5 Architecture-Centric Design and Configuration

5.1 Background

The extensible Architecture Description Language xADL 2.0 [5] comprises a set of XML schemas (XSD) explicitly aimed at encouraging simple, domain-specific model element refinements, extensions, and constraints.[2] We will briefly describe xADL's main elements that are relevant for our purpose and outline where our extensions plug into the overall schema hierarchy. The interested reader will find an extensive discussion of the language's features in [5].

At its core, xADL provides a component and connector view where connectors are treated as first class entities (i.e., components and connectors are wired up via links). It introduces a simple type systems, thus differentiating between *Component* and *ComponentType*, *Connector* and *ConnectorType*, as well as *Interface* and *InterfaceType*. This type-instance hierarchy allows reasoning on common component (i.e. service) and connector (i.e., message middleware) behavior or implementation. For our purpose, within the scope of the xADL *types* schema, the architect defines general publish-subscribe and point-to-point channel connector types in addition to all the various component types foreseen in the complex service system. *ComponentTypes* and *ConnectorTypes* expose *Signatures* which in turn may refer to interface types. It is thus possible to distinguish between provided and required interfaces. A messaging connector type typically exhibits a signature for sending messages and a signature for retrieving messages, both according to the same interface type. The xADL *structure* schema subsequently exhibits all the component and connector instances including their specific wiring. Type inheritance is optional.

A second set of schemas target the specification of implementation details. The *abstract implementation* schema identifies the plug-in locations where concrete implementations subsequently provide technology specific details. Natively xADL provides only modeling constructs for Java-based implementations deployed on a single JVM. Figure 3 depicts the xADL modeling constructs, their relations, and our extensions.

5.2 Message-Centric ADL Extension

The main architecture modeling concerns in a message-centric complex service system are configuration of the messaging middleware, definition of message channels, direction and type of messages, message request/reply correlation beyond generic one-way messages, and messaging middleware access properties.

[2] Throughout this paper any reference to xADL always implies xADL version 2.0.

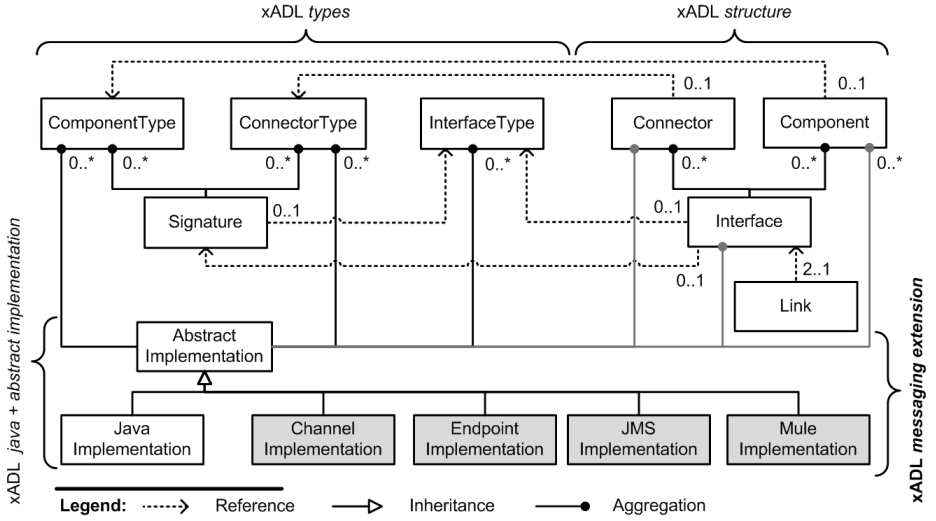

Fig. 3. Simplified xADL schema excerpt including the messaging extensions (dark grey)

To this end, we provide a set of four implementation extensions (see also Figure 3 bottom). The schemas for *Channel Implementation* and *Endpoint Implementation* provide general messaging properties, while *Mule Implementation* and *JMS Implementation* express technology specific properties for Mule and ActiveMQ respectively. The separation into four schemas also reflect the fact that each schema applies only to a particular core architecture element.

Channel Implementation (aka EAI message channel pattern) applies to a Connector element and specifies whether the Connector behaves as a publish-subscribe channel or point-to-point channel and provides the respective name.

Endpoint Implementation (aka EAI message adapter pattern) applies to an Interface element associated with a Component (and will be ignored when the Interface is associated with a Connector). The *Durable_ Name* properties identifies the subscriber of a durable subscription towards the channel connector. When a component dispatches a message for which is expects a reply, its sending interface must identify the point-to-point channel where it expects to eventually receive the reply message from via the *Reply_ To_ Queue* property. To completely specify a request/reply pattern, the requesting component exhibits a receiving interface that signals its role via the *Connection_ To_ Request_ Endpoint* property. A component may thus exhibit multiple, unambiguously defined request/reply interface pairs. On the replying component (within a request/reply pattern), the receiving *in* interface points to the replying *out* interface via the *Reply_ To_ Queue* property, which in turn completes the bi-directional references via the *Connection_ To_ Request_ Endpoint* property. The *Endpoint_ Position_ No* property allows for specifying an ordering of interfaces.

Mule Implementation applies to a Component element, indicating that the component is a composite service, specified by a Mule workflow. The configuration properties comprise the *file_ id* where to save the Mule workflow skeleton and generic parameter/value *AdditionalConfig* properties targeted at Mule. All components with the same *file_ id* end up in the same configuration and thus will be collocated on the same Mule instance.

JMS Implementation applies to a ConnectorType element and configures an ActiveMQ instance. The *Transport_ Configuration* property specifies at least one ActiveMQ connection endpoint URL. The optional *JMS_ Specification_ Version* property holds the JMS protocol version, by default 1.1. The optional *Persistence_ Configuration* property captures persistence adapter (default is kahaDB) and storage directory. Finally, the *file_ id* property determines which JMS endpoints are hosted on the same ActiveMQ instance.

5.3 Consistency Checks

We have devised an initial set of soft and hard consistency checks that issue warnings and recommendations on how to mitigate the inconsistency. The checks are restricted to Component, Connectors, ConnectorTypes, and Interfaces refined with our xADL extensions. Basic checks such as interface direction and type compatibility are already available in the ArchStudio4 (see Sec. 6).

Most messaging-centric checks apply at the architecture level. We detect when there exists a link directly between two components or two connectors. Two connector instances of the same connector type cannot share the same channel name. Subscriber and publishers would otherwise share the same channel which is in conflict with the architecture-prescribed distinct channels. Every component interface can only link to a single connector interface as the interface represents the message channel at the service side. We recommend that every connector has exactly one *in* and one *out* interface. The use of *inout* interfaces is discouraged. Instead a set of separate request and reply queues, thus implying separate *in* and *out* interfaces, unambiguously document the intent of the *inout* direction. Our checks warn when multiple message consumers link to the out interface of a point-to-point channel. Only one nondeterministic subscriber will be able to obtain the message. We also warn when a publish-subscribe channel (rather than a point-to-point channel) is used within the scope of a request/reply pattern. With multiple subscribers to the request topic, multiple responses may occur.

Complementary component level checks ensure the proper use of the message-centric request/response pattern. A response endpoint (interface) must refer to its respective, initiating request endpoint (interface), both must exist, reside on the same component, and may not be identical. Additional tool-centric checks ensure that the architect provided all required information for transforming the model to message broker and ESB configuration (see following section 6).

6 Tool Support

6.1 Architecture-to-Configuration Transformation

For the purpose of this paper, we focused on two architecture-to-configuration transformations. As example for configuring a message-oriented middleware, we generate the XML-based ActiveMQ Server configuration. A ConnectorType's JMS implementation is sufficient for deriving a server's configuration which consists of two parts. The *Persistence_ Configuration* determines the *persistenceAdapter* and all *Transport_ Configuration*s determine the set of available *transportConnector*s. The transformation also ensures traceability by adding the connector type's id as a comment to the configuration files *broker* element. Note that the transformation ignores any connectors and thus doesn't specify what queues or topics the server will eventually manage as ActiveMQ creates these on the fly. Ultimately, every connector type with a JMS implementation results in a separate *transportConnector* element. Configurations for connector types that share a *file_ id* are aggregated into a single configuration file and subsequently end up collocated on the same ActiveMQ server instance.

As example for configuring a service endpoint, we provide the complete message specification for a Mule workflow, i.e., a workflow designer may neglect any message-broker related details and can focus purely on the local message processing. The Mule workflow configuration captures components, interfaces, and their wiring to the various connectors, while the ActiveMQ service configuration represents only the connector types in complex service system's architecture. Each component results in a separate mule workflow specification. The transformation places all workflow specifications from component with the same *file_ id* in the same file, and thus collocates them on the same Mule ESB instance. To this end, the transformation first retrieves all connectors (with channel implementation) and obtains the JMS configuration from the corresponding connector type. Each distinct connector type becomes an *activemq-connector* element. For each interface, a new *[inbound|outbound]-endpoint* element obtains the configuration properties from the endpoint implementation, the channel name from the linked connector's channel implementation, and the respective *connector-ref* to the *activemq-connector*. Our transformation treats two interfaces coupled via a *Connection_ To_ Request_ Endpoint* property and *Reply_ To_ Queue* property differently depending on whether they represent the requesting component or the replying component. In case of the former, the respective two mule endpoints become wrapped in a *request-reply* element and a preceding *message-properties-transformer* element. In case of the latter, the receiving interface becomes a *inbound-endpoint* with an *exchange-pattern="request-response"* property while the outgoing interface is ignored. The respective reply endpoint information arrives embedded in the request message at runtime.

6.2 ArchStudio Integration

We realized our approach as a prototype on top of ArchStudio 4 [4], a visual, Eclipse-based IDE for editing xADL documents. ArchStudio comes with two

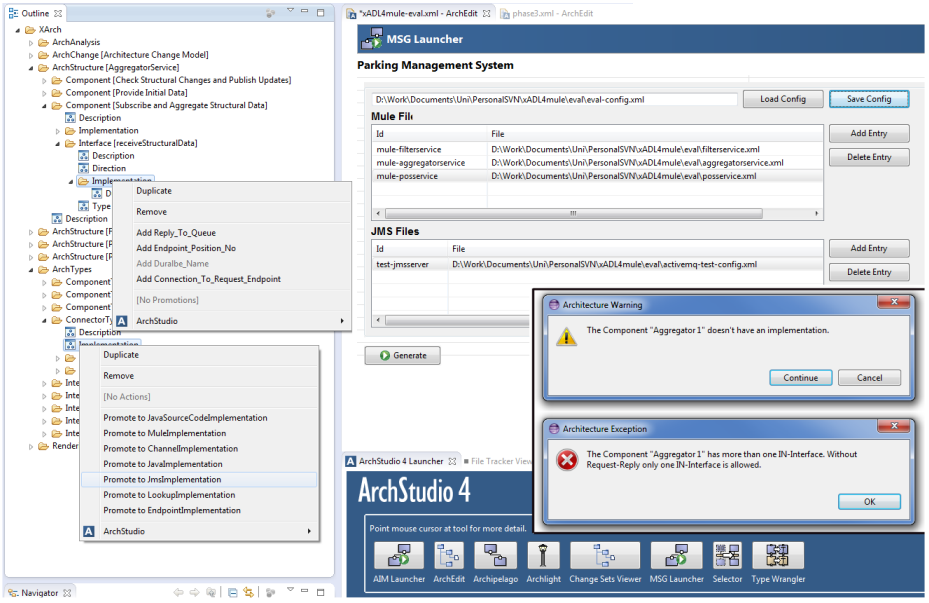

Fig. 4. ArchStudio extension screenshot: schema extension (left), transformation configuration file mapping (top), exemplary inconsistency alerts (inset)

main editors: *ArchEdit* provides access to the underlying xADL document (including all extensions) as a tree, while *Archipelago* offers a drag-and-drop, point-and-click interface for placing and wiring up components and connectors. ArchStudio foresees the integration of additional functionality through extensions.

Schema Extensions. For the purpose of our approach, it proved sufficient to extend xADL at the implementation schema level. The additional elements (recall subsection 5.2) blend in smoothly with the existing user interface, merely appearing as new implementation options (see Figure 4 left). An existing ArchStudio 4 user won't have to learn any new steps for utilizing our schema extensions. Under the hood, ArchStudio applies its *Apigen* tool for creating a data binding library for each xADL schema. *ArchEdit* and *Apigen*'s limitations combined result in configuration properties being limited to strings, references, and complex data structures thereof.

Consistency Checking. We implemented the consistency checker as a dedicated component within ArchStudio. The checker raises warnings and errors during execution, depending on the consistency rule severity. The user may decide to ignore warnings and still continue to configuration transformation later on. Transformation is disabled, however, in the presence of consistency errors (see Figure 4 inset). In general, consistency checking is cheap. The consistency algorithm's runtime complexity is $\Theta(comp + l)$ for architecture components ($comp$) and links (l) as rules are either local (e.g., interface properties, interface link cardinality) or access only a link's two referenced elements (e.g., compatible

interface direction). Checking the uniqueness of channel names of connectors deriving from the same connector type is slightly less efficient: the algorithm's complexity is $\Theta(n\ log\ n)$ in the number of connectors.

Transformation. As mentioned above, the actual transformation component becomes only available after passing all consistency checks. The only user interaction with the transformation consists of mapping a component's and connector type's file id to an actual file location (see Figure 4 top).

We are currently in the process of open-sourcing our tool as an ArchStudio4 add-on and will update this paper with a link to the tool website as soon as we have put together sufficient installation documentation. For now, the extending xADL schema documents, example architecture model, and corresponding generated configuration files are available as Supporting Online Material (SOM) at `http://wp.me/P1xPeS-5H`.

7 Proof-of-Concept Case Study

We utilized our prototype tool support for developing a parking management complex service system. The system is similar to the one introduced in the motivating scenario but for reasons of confidentiality we cannot disclose the actual system architecture. Any depicted and described architecture excerpts, hence, closely match the system in structure but exhibit generalized element names and properties. We briefly report on the development process and respective application of our approach to demonstrate not only feasibility but also actual benefit in a representative, real-world development environment.

Our approach and tool support allows for an iteratively refining system design methodology. As the architecture goes through various iterations, the architect gradually assigns implementation elements to components, connectors, and interfaces. Mule workflow developers pick the various components and generated workflow specification and implement the internal composite service behavior. Specification and changes need not necessarily flow only from architecture to configuration. Due to page constraints, we are unable to describe our additional tool capabilities such as generating the messaging endpoints within pre-existing Mule workflows (rather than from-scratch workflow generation), non-destructive change propagation of architecture updates into workflow configurations, and consistency checking upon changing message-centric elements conducted within the mule workflow editor. These aspects are subject to future publications. The architect runs consistency checks in any development phase, after any update to the architecture or Mule workflow and thus can guarantee that inconsistencies are immediately detected, respectively that the prescribed architecture and system are in a consistent configuration.

Ultimately, the architect arrives at a model similar to the excerpt in Figure 5. It contains one composite Filter Service, Aggregator Service, and POS Service each (for a total of eight Mule workflows), along with the five intermediate message queues/topics. The Filter Service comprises of two components: one Mule workflow for filtering and enhancing dynamic changes events from parking sites

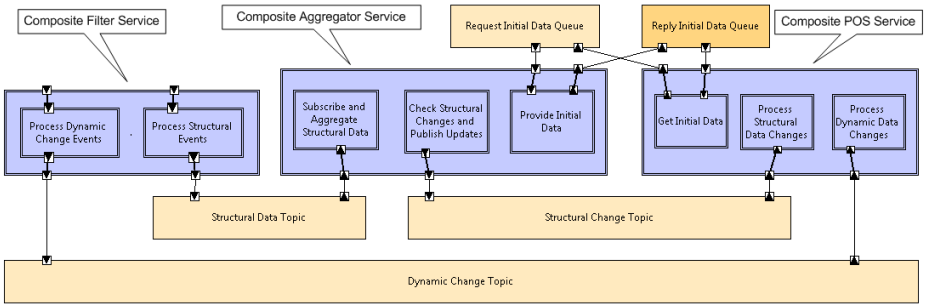

Fig. 5. Evaluation system architecture except: depicting composite service components (dark/blue) and message broker connectors (light/beige) - ArchStudio screenshot (colors online)

(not shown), and one workflow processing structural events. The composite Aggregator Service comprises three workflows: one for obtaining structural data (typically provided by more than one Filter Service), one for checking the structural data for changes relevant to POS services and dispatching those changes, and one for providing POS services with initial complete state information. A generic POS Service contains at minimum flows for (i) obtaining initial data (in this case, the POS service is aware from which Aggregator Service it receives such initial information), for (ii) receiving structural updates and for (iii) dynamic data updates. Further locally relevant flows which contain the actual business logic are irrelevant at this architectural level. Similarly, shared databases serving multiple flows within a single Mule instance need not be configured at this level but instead are within the scope of a Mule configuration file. Note also that the architectural substructures are included for sake of better understanding. Collocation of mule workflows depends solely on specifying the same implementation file id property.

Tool supported consistency checking pays of even for this small architecture excerpt. A single execution of the all consistency checks outlined in Section 5.3 on the architecture in Figure 5 results in four architecture-level checks, four link-specific checks, three connector checks, one connector type check, and six component checks (including respective interfaces). Remember that an architecture would need to conduct many more checks when conducting the same analysis on Mule and ActiveMQ files alone. The ActiveMQ configuration is void of any topic and queues definitions, thus there exists no authoritative, explicit connector element. Observing a simple example such as ensuring that a queue has only a single receiver or that queue/topic names are unique: the architect needs to traverse the Mule configuration for each queue and topic definition first to the corresponding Mule messaging endpoint definitions (requiring a detailed understanding of the configuration file) and then pairwise compare this information across all included mule workflows (i.e., $n * (n-1)/2$ comparisons, thus 45x for our use case's 10 connected component interfaces); a tedious and error-prone task, especially for larger systems.

Discussion and Limitations

Complex service systems do not necessarily need to exhibit all the challenging properties listed in the introduction: prohibiting centralized execution control, consuming and providing data rather than invocations, experiencing unpredictable service availability, and supporting a dynamically changing number of service instances. Systems that encounter only a subset will equally benefit from our approach and tools.

Currently, our architecture-to-configuration transformation produces only Mule workflows and ActiveMQ configurations. The underlying real world development project underlying our evaluation scenario identified these technologies as sufficient and providing a good balance between a light-weight messaging framework and the expressive and extensible Mule workflows for composite service design. Our approach remains valid for other messaging protocols or frameworks as well as for other service design methodologies. The architecture-level consistency checking mechanism remain applicable. Ultimately, supporting other runtime frameworks does not necessarily require adapting our ArchStudio add-on. For small deviating tasks, such as generating an OpenJMS server configuration, access to the architecture model via *Apigen*'s data binding libraries, or directly via the xADL XML file will be sufficient.

8 Conclusions

We made the case for architecture-centric design of complex, message-based service systems. Our approach targets the specification of systems comprising dynamically fluctuating instances of message-driven, highly decoupled composite services with uncertain availability. Our extension to xADL provides the basis for central specification and consistency checking at design-time, subsequently achieving consistent run-time configuration without having to rely on a central coordinator. Our prototypical tool integrated with ArchStudio4 produces configurations for the ActiveMQ message broker and Mule workflow endpoints.

Our future work focuses on following two aspects: on the one hand, we plan to extend the set of supported protocols and tools (e.g., the advanced message queue protocol AMQP or WS-Notification). It will be especially worthwhile evaluating how the EAI patterns (currently modeled internally in Mule) may be explicitly supported by our ADL and subsequently mapped to EAI frameworks such as Apache Camel. On the other hand, we will investigate additional analysis aspects such as optimal channel allocation across message-broker instances and their location in proximity to the various services.

Acknowledgment. This work is partially supported by the European Union within the SIMPLI-CITY FP7-ICT project (Grant agreement no. 318201).

References

1. van der Aalst, W., Hofstede, A.H.M.T.: Yawl: Yet another workflow language. Information Systems 30, 245–275 (2003)
2. Baresi, L., Ghezzi, C., Mottola, L.: On accurate automatic verification of publish-subscribe architectures. In: Proc. of the 29th International Conference on Software Engineering, ICSE 2007, pp. 199–208. IEEE Computer Society, Washington, DC (2007)
3. Barker, A., Walton, C., Robertson, D.: Choreographing web services. IEEE Transactions on Services Computing 2(2), 152–166 (2009)
4. Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., Taylor, R.: Archstudio 4: An architecture-based meta-modeling environment. In: Companion to the Proc. of the 29th International Conference on Software Engineering, pp. 67–68. IEEE Computer Society, Washington, DC (2007)
5. Dashofy, E.M., Van der Hoek, A., Taylor, R.N.: A highly-extensible, xml-based architecture description language. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, pp. 103–112. IEEE (2001)
6. Decker, G., Kopp, O., Leymann, F., Weske, M.: Bpel4chor: Extending bpel for modeling choreographies. In: IEEE 20th International Conference on Web Services, pp. 296–303. IEEE Computer Society, Los Alamitos (2007)
7. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web Grid Serv. 1(1), 1–30 (2005)
8. Esfahani, N., Malek, S., Sousa, J.P., Gomaa, H., Menascé, D.A.: A modeling language for activity-oriented composition of service-oriented software systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 591–605. Springer, Heidelberg (2009)
9. Garcia, J., Popescu, D., Safi, G., Halfond, W.G.J., Medvidovic, N.: Identifying message flow in distributed event-based systems. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 367–377. ACM, New York (2013)
10. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, Reading (2003)
11. Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. SIGPLAN Not 39(10), 170–187 (2004)
12. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language (WS-BPEL) Version 2.0 (April 2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
13. Pautasso, C., Heinis, T., Alonso, G.: Jopera: Autonomic service orchestration. IEEE Data Eng. Bull. 29(3), 32–39 (2006)
14. Scheibler, T., Leymann, F.: A framework for executable enterprise application integration patterns. In: Mertins, K., Ruggaber, R., Popplewell, K., Xu, X. (eds.) Enterprise Interoperability III, pp. 485–497. Springer, London (2008)
15. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley (2009)
16. Yildiz, U., Godart, C.: Information flow control with decentralized service compositions. In: IEEE Int. Conf. on Web Services, pp. 9–17 (July 2007)
17. Zaha, J.M., Barros, A., Dumas, M., ter Hofstede, A.: Let's dance: A language for service behavior modeling. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 145–162. Springer, Heidelberg (2006)
18. Zheng, Y., Taylor, R.N.: Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In: Proc. of the 34th Int. Conf. on Software Engineering, ICSE 2012, pp. 628–638. IEEE Press, Piscataway (2012)