AN ALGEBRAIC SERVICE COMPOSITION MODEL FOR THE CONSTRUCTION OF LARGE-SCALE IOT SYSTEMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

Damian Isaid Arellanes Molina

Department of Computer Science

Contents

Word Count: 68892

List of Tables

List of Figures

Abstract

AN ALGEBRAIC SERVICE COMPOSITION MODEL FOR THE CONSTRUCTION OF LARGE-SCALE IOT SYSTEMS

Damian Isaid Arellanes Molina

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2019

The Internet of Things (IoT) is an emerging paradigm that envisions the interconnection of (physical and virtual) objects through innovative distributed services. With the advancement of hardware technologies, the number of IoT services is rapidly growing due to the increasing number of connected things. Currently, there are about 19 billion connected things, and it is predicted that this number will grow exponentially in the coming years. The scale of IoT systems will hence surpass human expectations as such systems will require the composition of billions of services into complex behaviours. Thus, scalability in terms of the size of IoT systems becomes a significant challenge.

Existing service composition mechanisms (i.e., orchestration, choreography and dataflows) were primarily designed for the integration of enterprise services, not for the physical world. For that reason, they do not provide the requisite semantics and hence properties for tackling the scalability challenge that future IoT systems pose. In this thesis, we identify crucial scalability requirements for IoT systems, and propose an algebraic service composition model for the construction of large-scale IoT systems. The resulting model, DX-MAN, has been validated with a software platform and evaluated against the scalability requirements. A comparison with the related work shows that DX-MAN advances the state of the art on IoT service composition and it is, therefore, promising for the construction of future large-scale IoT software systems.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Kung-Kiu Lau for the endless discussions, guidance and support throughout this wonderful journey. Thank you for many hours of fascinating discussions and pertinent criticisms that, certainly, have a lot to do with the present form of this research. I would also like to thank my examiners, Prof. Gordon Blair and Prof. Rizos Sakellariou, for their invaluable feedback and suggestions that improved the final version of this document.

To my mom for all her moral support, listening and patience. Thank you for being there when I needed you most.

To my father for the financial support throughout my entire education. Thank you for making my life easier and helping me to follow the seemingly unending path of research.

None of this could have been possible without the financial support of the National Council of Science and Technology (CONACyT, no. 280863/411891) which is one of the most remarkable institutions I have ever known. I would also like to thank the Secretary of Public Education (SEP) for providing me with a complementary scholarship. Thank you all for making this dream come true.

Chapter 1

Introduction

"The art of asking questions is more valuable than solving problems."

— George Cantor, *1867*

With the advent of the Internet of Things (IoT), we are entering a world where software is becoming more and more pervasive. IoT is an emerging paradigm that promises the interconnection of practically every (physical and virtual) object through innovative distributed services. Like traditional enterprise services, IoT services interact in many different ways via the Internet to realise a global system behaviour. However, unlike traditional enterprise systems, IoT systems will require the interaction of billions of services since the number of connected things (and therefore services) is rapidly growing. Currently, there are around 19 billion devices connected to the Internet infrastructure [IoT18, SNP+15] and it is estimated that this number will increase by a factor of 1.92 in the next six years (see Figure 1.1). Hence, scalability becomes a crucial concern for the full realisation of future IoT systems.

Scalability is typically considered as a system capability to handle increasing workloads [HN18, VN17, ARJ18, MVT17, SA16, LLZ14]. In particular, vertical scalability [SBAB19, CPS17, RMBG18] refers to the addition or removal of computing resources in a single IoT node, while horizontal scalability [CSB19, WLB09, SNP+15] involves the addition or removal of IoT nodes. These kinds of scalability have been extensively investigated [CGK+11, MSR17, XH16, GMA17, CSC+18, CPS17, RMBG18, CdCSR+15, BD16, VN17], unlike scalability in terms of the number of services composed in an IoT system, which this thesis refers to as *functional scalability*.

Existing service composition mechanisms were primarily designed for the integration of static enterprise services, not for the physical world. For that reason, they may

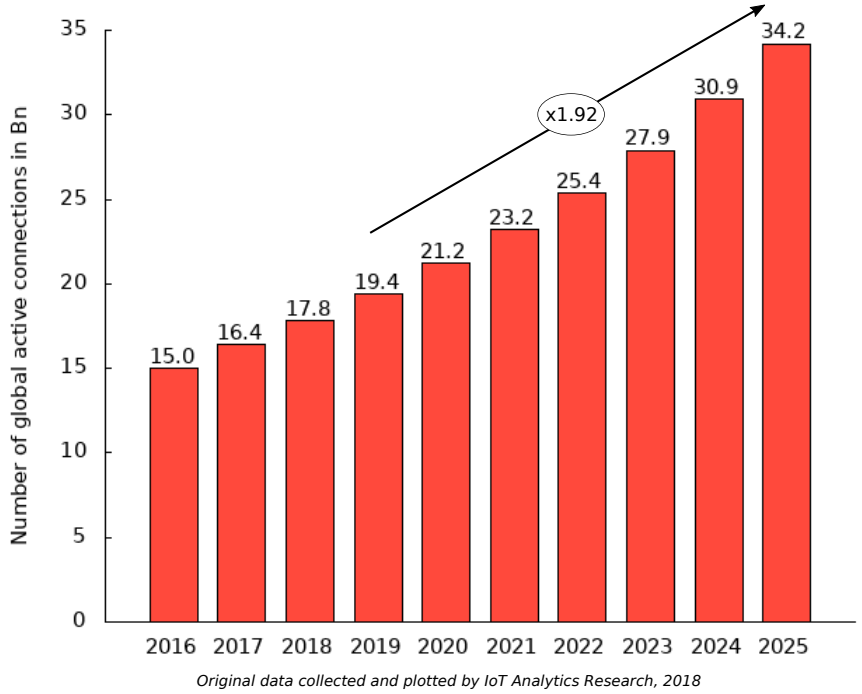


Figure 1.1: Total number of active device connections worldwide [IoT18].

not address the functional scalability challenges that IoT systems pose. Early IoT systems were deployed in closed environments using private Application Programming Interfaces (APIs) and private data. However, future IoT systems will be deployed in open environments (also known as software ecosystems [MS03]) with an overwhelming number of available services, as a result of the vast amount of connected things [WSJ15]. Hence, billions of IoT services will be composed into complex IoT systems [FGG+06, AIM10, DPB17, RNN+16, Kop11, TG18], thereby raising the challenging question of *How to construct IoT systems composed of a large number of services?*

This thesis proposes a service composition model for the construction of large-scale IoT systems. The model advances the state of the art on IoT service composition and it is the first one to address the functional scalability problem.

1.1 IoT Service Composition

Compositionality is the ability to combine services into complex behaviours via a *service composition mechanism* [LDC17]. This thesis particularly focuses on service composition mechanisms that define behaviour by workflow control flow.

Workflows are increasingly important for IoT systems because they allow the integration of IoT services into complex tasks that automate a specific context [XV14, PKS+15, MCS16, SKNS15, SHHA19]. For example, a smart home can be automated with a workflow that regulates the temperature of a room according to environmental changes. In the domain of smart agriculture, a workflow can be defined to analyse data coming from harvest sensors, predict diseases and react accordingly.

Defining future IoT workflows is a challenging task that requires a proper composition mechanism able to handle any number of services. As a single entity cannot build systems of such a magnitude in one go, this thesis envisions that those workflows will emerge from the collaborative interaction of many individuals, organisations and inanimate objects worldwide.

Large-scale physical systems are not science fiction. The most prominent example is the Internet which was created in a collaborative effort over many years. Now, with the emergence of IoT, we are moving towards a generation of large-scale "software" systems composed of billions of interacting distributed services. This time the scale will be not only a particular property of networking systems, but also an inherent characteristic of software systems. For that reason, a service composition mechanism should enable the management of individual workflow segments to preserve global system properties while allowing the addition, removal, maintenance and replacement of any number of services. As a large-scale workflow cannot be depicted and even visualised in full, it is infeasible to show a complete example in this thesis (and everywhere else). Nonetheless, Chapter 4 provides an inductive insight on such workflows.

1.2 Problem Statement

Current service composition mechanisms were designed for enterprise systems that comprise relatively few services. However, with the advent of IoT, new challenges arise as a consequence of the overwhelmingly large number of available services.

Large-scale IoT systems will potentially be composed of billions of interacting distributed services that span multiple geographically dispersed administrative domains [GLL18, GBLL15, HTM+14, RCL14]. Hence, no single entity should govern an entire composition workflow to ensure that every participant has control over its workflow part [ATS14, FYG09, AL18b]. Consequently, a service composition mechanism must support interoperability by defining *distributed cross-domain workflows*.

Large-scale IoT systems could also exhibit a high degree of heterogeneity in many

different forms [AL18b, DPB17, NAG19]. For instance, there may be different service providers (e.g., Amazon AWS IoT and IBM Watson), a wide variety of programming languages (e.g., Swift and embedded C), multiple operating systems (e.g., Contiki and TinyOS) and different network communication protocols (e.g., CoAP and MQTT). The *separation of control flow, data flow and computation* enables a flexible composition of services in such heterogeneous environments, as it allows independent maintenance, validation, verification, reuse and evolution of such functional concerns. Also, services are loosely coupled because control flow is never embedded in the computation of many services. For that reason, a service composition mechanism must separate control, data and computation [NAG19].

As the number of composed services becomes massive, execution failures become unavoidable, more challenging to manage and may potentially unleash catastrophic consequences (for individuals or societies) [SA11]. As it allows the visualisation of workflow logic, *explicit control flow* becomes crucial for monitoring, tracking, verifying, maintaining and evolving complex, large-scale IoT systems. Therefore, it is a vital need for any IoT service composition mechanism.

Large-scale IoT systems will also be inherently dynamic and uncertain due to the presence of churn in the operating environment [BD16, BS16, WYG+17, DPB17, IGH+11, ZGLB10, Kop11]. Churn means that things (and their services) dynamically connect and disconnect from the network, as a result of auto-scaling, software upgrades, failures, poor connection and mobility. It is particularly evident when an IoT system uses resource-constrained things with a poor connection [HHP15] or when there are mobile things involved [WYG+17, TJLG17]. Hence, churn results in physical service locations (i.e., IP addresses) changing over time frequently. For that reason, a composition mechanism must support *location transparency* for dealing with highly dynamic environments.

Highly dynamic environments are also subjected to variability caused by external or internal factors [SS15, GZW+17, MASS08, HGR12]. External factors are beyond the scope of the system, and include changes in requirements and increasing workloads. Internal factors are associated with the system operation, and include system failures and sub-optimal behaviours. A service composition mechanism must then support the definition of alternative workflows that adapt a composite to changes in both the external and the internal environment. As manually choosing them is a costly and inefficient management process, workflows must be selected

with minimal or without human intervention [CBF+16]. Thus, *workflow variability* is a crucial desideratum for the realisation of large-scale autonomous IoT systems [FGG+06, GZW+17, WRS18, HGR12, BMGG+16, SKNS15].

Network performance is also crucial for large-scale IoT systems that potentially exchange vast amounts of data continuously. Such an exchange must be as efficient as possible to avoid performance bottlenecks, achieve better response time and improve throughput [KLS+10]. Since they require only one network hop to pass data directly from a service producer to a service consumer, *decentralised data flows* provide the best QoS and are, therefore, crucial for a composition mechanism that supports the construction of data-intensive IoT systems [AL19a, GKB12, BWVH12, BCF09, Liu02, BWR09, SDSB19, CSGD+14].

Overall, this thesis addresses the problem of how to compose future large-scale IoT systems that may potentially be multi-domain, complex, heterogeneous, highly dynamic and data-intensive.

1.3 Aim and Research Questions

This thesis is grounded on the hypothesis that *algebraic service composition can be leveraged for tackling the functional scale of future IoT systems*. To verify this, the following questions are addressed:

RQ1 What are the functional scalability requirements of IoT systems?

RQ2 What are the existing IoT service composition mechanisms that define workflows at design-time for the construction of IoT systems?

RQ3 What is the degree of satisfaction of existing IoT service composition mechanisms with respect to the functional scalability requirements identified?

RQ4 How could IoT services interact at run-time in a large-scale IoT system?

RQ5 Are algebraic service interactions suitable for large-scale IoT systems in terms of the functional scalability requirements identified?

RQ6 Does the proposed model fulfil all the functional scalability desiderata identified?

Figure 1.2 shows the three-level dependency tree among the research questions of this thesis. The hypothesis is at the top-level of the hierarchy and requires the study of *RQ3*, *RQ6* and *RQ5* which, in turn, need the investigation of *RQ1*. The research questions *RQ3* and *RQ5* also depend on the study of *RQ2* and *RQ4*, respectively.

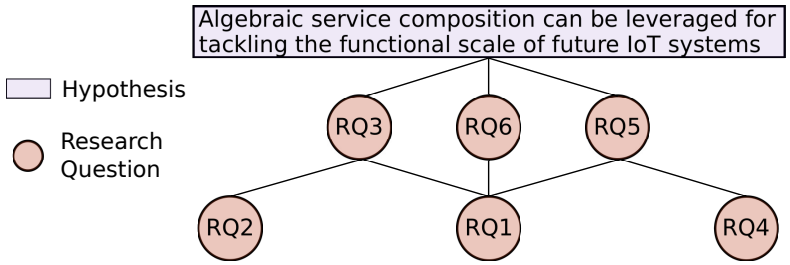


Figure 1.2: Dependency tree among the research questions of the thesis.

1.4 Research Methodology

This section describes the research methodology which is defined with BPMN 2.0 notation [OMG11] and consists of seven fundamental stages (see Figure 1.3).

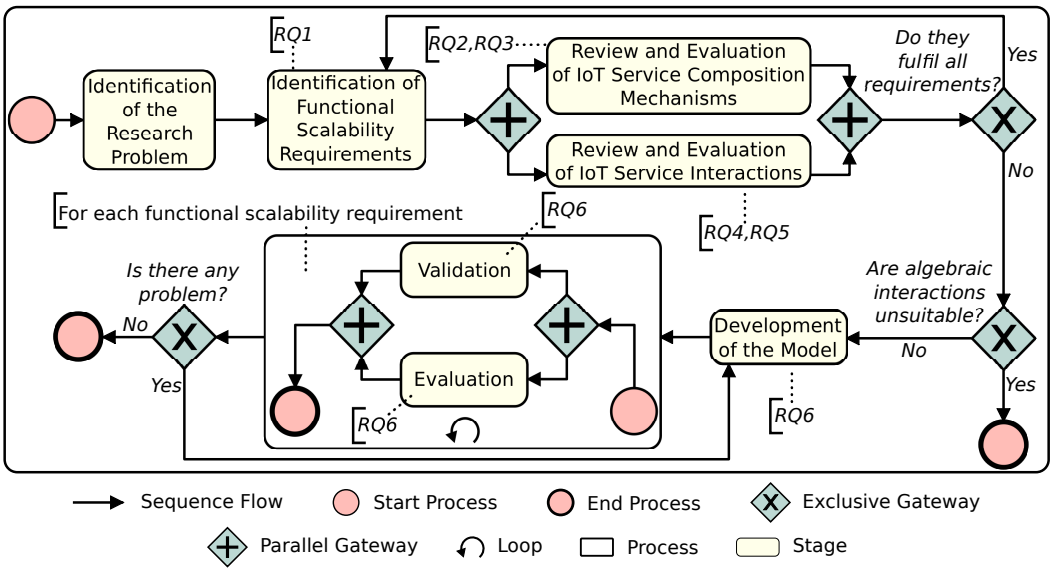


Figure 1.3: Research methodology.

The investigation starts with the identification of the research problem and the consequent discovery of functional scalability requirements. The requirements are then used to ascertain if there is at least one mechanism that fulfils all the desiderata. If

this is the case, further requirements are identified. In parallel, IoT service interactions are reviewed and evaluated also in terms of the desiderata identified. If and only if algebraic service compositions are suitable for large-scale IoT systems and no composition mechanism satisfies all the desiderata, a novel service composition model is developed, whose desiderata support is validated and evaluated in parallel. The investigation finishes if no problems arise during validation and evaluation. Otherwise, the model is revised and changed accordingly. For completeness, the remaining of this section briefly describes the individual stages of the methodology.

1.4.1 Identification of the Research Problem

This stage identifies the research problem via a comprehensive study of both large-scale software systems and service composition. The problem is systematically tackled by investigating the research questions (presented in Section 1.3) according to the temporal logic formula 1.1 [Ven17]. The precedence binary relation $\prec$ means that the study of one question precedes the investigation of another one. For example, $x \prec y$ means that the question x is studied before the question y. On the other hand, the relation $\preceq$ means that a question is addressed before or at the same time as another one. For example, $x \preceq y$ means that the question x is investigated before or at the same time as the question y.

$$RQ1 \prec ((RQ2 \prec RQ3) \preceq (RQ4 \prec RQ5)) \prec (RQ6 \preceq RQ7) \tag{1.1}$$

1.4.2 Identification of Functional Scalability Requirements

This stage addresses the research question *RQ1* by analysing research papers, magazines, websites and experiences from companies (that deal with "large-scale" software systems). The idea is to iteratively identify different functional scalability requirements and the relationship between them. As IoT service composition is an abstraction rather than a concrete implementation, the requirements cannot be quantitative, but only qualitative. Figure 1.4 shows that this stage identifies six crucial desiderata: (i) *explicit control flow*; (ii) *distributed workflows*; (iii) *location transparency*; (iv) *decentralised data flows*; (v) *separation of control, data and computation*; and (vi) *workflow variability*.

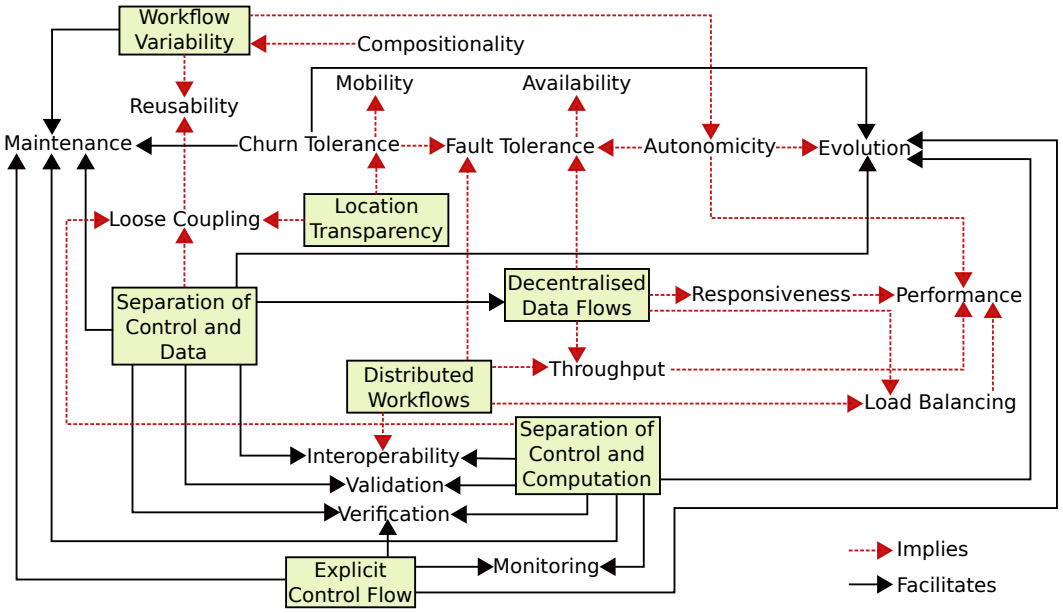


Figure 1.4: Functional scalability requirements of IoT systems.

A glance at Figure 1.4 reveals the association between requirements and large-scale system qualities. For instance, *workflow variability* implies the *reusability* of multiple behaviours in different contexts, while *loose coupling* entails the *reusability* of services in systems that exhibit different requirements. As another example, the *separation of control and data* facilitates the *evolution* of large-scale systems because workflow logic and data flow logic can evolve separately. The requirements and their association are explained later in this dissertation. Further details can also be found in [AL19b, AL18b].

1.4.3 Review and Evaluation of IoT Service Composition Mechanisms

This stage addresses the research question *RQ2* by conducting a systematic review of the literature on service composition mechanisms that define IoT workflows. The survey reveals that there are three primary mechanisms, namely (centralised and distributed) *dataflows*, (centralised and distributed) *orchestration* and *choreography*. Once the literature review is done, the research question *RQ3* is studied to ascertain if there is at least one mechanism that fulfils all the scalability requirements identified. As the answer is negative, this stage concludes that it is meaningful to investigate the development of a novel composition model (with the desiderata in mind).

1.4.4 Review and Evaluation of IoT Service Interactions

This stage addresses the research question *RQ4* to discover how services could interact in large-scale IoT systems. To do so, it systematically reviews and classifies service interactions that happen at run-time in current IoT systems. This activity reveals that there are four primary interaction schemas, namely *direct message passing*, *indirect message passing*, (P2P and broker-based) *event-driven interactions* and (one- and multi-level) *exogenous interactions*. Once the schemas have been identified, this stage addresses *RQ5* by conducting a systematic comparison of IoT service interactions in terms of the functional scalability requirements identified. The conclusion is that algebraic composition results in *multi-level exogenous interactions* which turns out to be the most promising schema.

1.4.5 Development of the Model

This stage studies the research question *RQ6* by investigating the development of a novel algebraic service composition model, DX-MAN, that satisfies the functional scalability requirements to a degree of 100%. To do so, this stage particularly:

- Determines if control flow is visible in algebraic composite services.
- Investigates the possibility of partitioning workflow control flow over multiple algebraic composite services that reside on different network nodes.
- Analyses if algebraic compositions provide location transparency.
- Investigates a decentralised data exchange approach that leverages algebraic composition for the analysis of data dependencies.
- Investigates the possibility of separating control, data and computation.
- Investigates the resulting type of algebraic composition (i.e., *compositionality*) and the possibility of using algebraic composition operators as variability operators.

1.4.6 Validation

This stage also works on the research question *RQ6* and it is crucial for ensuring model soundness because it validates that specific functional scalability requirements truly

exist in the DX-MAN model. In particular, it iteratively implements and tests a set of requirements in a distributed software platform [AL17a], and then validates them. Figure 1.5 illustrates this sequential process.

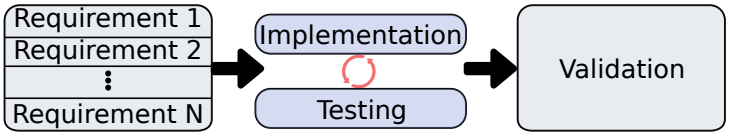


Figure 1.5: Validation strategy.

Note that the platform is causally connected to the model so that it is continuously refined throughout the thesis to reflect model updates.

1.4.7 Evaluation

This stage qualitatively and empirically evaluates the DX-MAN model using the strategy depicted in Figure 1.6. It particularly addresses the research question *RQ6* by conducting a comparative evaluation of DX-MAN with existing IoT service composition mechanisms in terms of *functional scalability requirements* and *compositionality*. For further analysis, *explicit control flow*, *workflow variability*, *compositionality* and *location transparency* are evaluated through pertinent case studies. *Decentralised data flows* is the only requirement that is evaluated empirically with a controlled experiment.

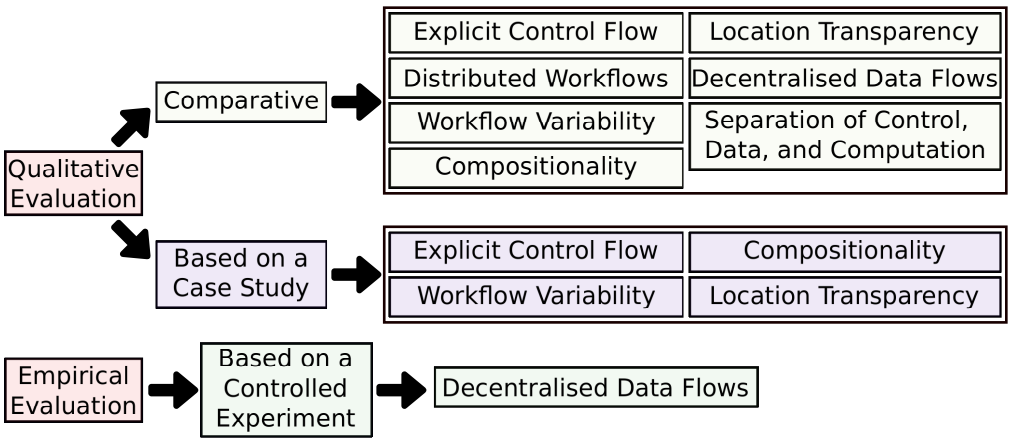


Figure 1.6: Evaluation strategy.

1.5 Contributions

The main contributions of this thesis are:

1. **A survey on IoT service composition mechanisms**, which focuses on fundamental semantics (i.e., how to compose services) underlying so-called "composition algorithms," programming frameworks, programming languages and software platforms. The survey simplifies the vast amount of work on IoT service composition under a common classification that consists of three primary mechanisms: (centralised and distributed) dataflows, (centralised and distributed) orchestration and choreography. The findings show that IoT service composition is just another name for composition of Service-Oriented Architectures (SOA).

2. **An evaluation framework to measure the functional scalability degree of IoT service composition mechanisms**, which considers six crucial desiderata: (i) explicit control flow; (ii) distributed workflows; (iii) location transparency; (iv) decentralised data flows; (v) separation of control, data and computation; and (vi) workflow variability. The framework provides a useful starting point towards the construction of large-scale IoT systems, which can be refined to consider further scalability requirements.

3. **An analysis and classification of service interactions for the scalability of IoT systems**, where interactions are simplified into four major schemas: (i) direct message passing; (ii) indirect message passing; (iii) (P2P and broker-based) event-driven interactions; and (iv) (one- and multi-level) exogenous interactions. The analysis is a qualitative evaluation that uses an early version of the proposed framework to determine which schema best fulfils the functional scalability requirements of IoT systems. The results show that exogenous multi-level interactions are the most promising ones. This research contribution serves as a guideline for future research on large-scale IoT service interactions.

4. **A novel service composition model** referred to as DX-MAN, which was designed to fulfil all the functional scalability requirements of the proposed framework. The model uses algebraic composition to enable a systematic hierarchical bottom-up construction of service-based software systems, which is a well-known technique for tackling scale and complexity. Unlike existing service composition mechanisms, it provides total compositionality to enable workflow variability. Furthermore, it is the only composition approach that separates control,

data and computation. The most notable characteristics of the proposed model result in the following sub-contributions:

4.1 **A workflow variability theory** for the design-time definition of a (potentially) infinite family of workflow control flows, which combines variability with behaviour and provides workflows that are both non-deployable and executable only. At deployment-time, the theory allows system designers to manually choose the workflow that best fulfils the system requirements. At run-time, a self-adaptive mechanism (i.e., a feedback control loop) autonomously decides and executes the optimal workflow for the current system conditions. Thus, the theory is a cornerstone for the adaptation of service behaviour to different contexts.

4.2 **An approach for the realisation of decentralised data flows and the exogenous coordination of control flow**. It leverages the separation of concerns to avoid passing data alongside control, which contrasts with existing exogenous composition mechanisms (e.g., orchestration) where data is tightly coupled with control and passes through (multiple) mediators. Overall, the proposed approach is beneficial for loosely-coupled, large-scale IoT systems that exchange huge amounts of data continuously.

1.6 Thesis Overview

This thesis is presented according to the guiding principles of a *Journal Format*. The core contribution is a collection of six published peer-reviewed papers and one manuscript submitted, which I have produced during my PhD studies:

[AL17b] **Damian Arellanes** and Kung-Kiu Lau. Exogenous Connectors for Hierarchical Service Composition. In *International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 125–132. IEEE, 2017.

[AL17a] **Damian Arellanes** and Kung-Kiu Lau. D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures. In *International Symposium on Cloud and Service Computing (SC2)*, pages 283–286. IEEE, 2017.

[AL18a] **Damian Arellanes** and Kung-Kiu Lau. Algebraic Service Composition for User-Centric IoT Applications. In Dimitrios Georgakopoulos and Liang-Jie Zhang, editors, *Internet of Things – ICIOT 2018*, volume 10972 of Lecture Notes in

Computer Science, pages 56–69, Cham, Switzerland, 2018. Springer. **Best Paper Award**.

[AL19c] **Damian Arellanes** and Kung-Kiu Lau. Workflow Variability for Autonomic IoT Systems. In *International Conference on Autonomic Computing (ICAC)*, pages 24-30. IEEE, 2019.

[AL19a] **Damian Arellanes** and Kung-Kiu Lau. Decentralized Data Flows in Algebraic Service Compositions for the Scalability of IoT Systems. In *World Forum on Internet of Things (WF-IoT)*, pages 668–673. IEEE, 2019.

[AL18b] **Damian Arellanes** and Kung-Kiu Lau. Analysis and Classification of Service Interactions for the Scalability of the Internet of Things. In *International Congress on Internet of Things (ICIOT)*, pages 80–87. IEEE, 2018. Runner-up for the **Carole Goble medal** for outstanding doctoral paper in Computer Science.

[AL19b] **Damian Arellanes** and Kung-Kiu Lau. Evaluating IoT Service Composition Mechanisms for the Scalability of IoT Systems. *Manuscript submitted to the Future Generation Computer Systems Journal*, 2019.

In all the first-authored publications, I contributed to the main ideas proposal, research development, research planning, literature review, writing, evaluation and analysis of the results. My supervisor, Kung-Kiu Lau, also contributed to the ideas, proofread the papers and approved the results. To comply with the *Journal Format* policy, the self-contained papers are presented as they appear in print, with their respective abstracts, figures and references. Hence, there is a reasonable amount of repetition.

The first paper [AL17b] presents an initial version of the proposed model, DX-MAN, in the context of SOA. It describes the main semantic constructs and introduces exogenous connectors as service composition operators. The paper presents and discusses the concept of *algebraic composition* which is also referred to as *total compositionality* and enables a systematic hierarchical bottom-up construction of SOA systems. A qualitative evaluation shows that composition operators define *explicit control flow* for the coordinated execution of such systems. It also evaluates the *compositionality* of the model and the support for *location transparency*. Both the *separation of control, data and computation*, and the *distribution of workflow* control flow (over the network) are discussed throughout the paper.

The second paper [AL17a] presents a platform that implements the semantic constructs of the DX-MAN model. The platform delivers the necessary programming abstractions and the runtime environment to design, deploy and execute distributed service-based software systems. It is a useful starting point for the validation of explicit control flow, compositionality and the separation of control, data and computation. As it is causally connected to the model, the platform is updated whenever a new feature is added into DX-MAN. For that reason, it is a fundamental tool for the validation of the model as the thesis progresses, which later evolved into an IoT platform that is available online at `http://github.com/damianarellanes/dxman`.

The third paper [AL18a] extends the semantics of the DX-MAN model with *workflow variability* by combining variability with behaviour. It introduces the notions of abstract workflow tree and concrete workflow tree, which allow the selection of workflows at run-time. The evaluation is done using a case study in the domain of IoT end-user applications, where workflow variability is leveraged to mitigate change of run-time user requirements. The paper briefly discusses how existing IoT service composition mechanisms allow the definition of only one workflow at a time, which is a detrimental issue as the number of services increases.

The fourth paper [AL19c] advances the theory of *workflow variability* by extending the semantics of DX-MAN with both autonomic capabilities and the notion of *workflow spaces*. Exogenous connectors are presented as composition operators that define workflow variants, and new operators for (exclusive and inclusive) branching are introduced. The new semantics endow composite services with feedback control loops that autonomously select workflows at run-time whenever the context changes. As workflows are non-deployable and executable only, a feedback control loop changes a composite service behaviour by executing the selected variant, instead of dynamically reconfiguring the whole system's workflow. A comparison with the state of the art on workflow variability is presented and a qualitative evaluation is conducted using a case study in the domain of autonomous smart homes.

The fifth paper [AL19a] describes the data flow dimension of DX-MAN, which allows the definition of data flows per workflow variant, and proposes an approach that leverages the *separation of control and data* for the realisation of *decentralised data flows*. The algebraic semantics of the model allows a well-defined data dependency graph (at design-time) which is analysed (at deployment-time) to form a direct mapping between data consumers and data producers. Such a mapping prevents composition operators from passing data during workflow execution. The approach is validated

on top of the DX-MAN platform [AL17a] using the Blockchain as the underlying data space. An empirical evaluation is done by comparing decentralised data flows versus distributed data flows within the model. Results confirm that the approach scales well with the size of IoT systems. The solution is also compared qualitatively with the state of the art on service composition approaches that support "decentralised data flows."

The sixth paper [AL18b] classifies and analyses IoT service interactions into four schemas: direct interactions, indirect interactions, (P2P and broker-based) event-driven interactions and (one- and multi-level) exogenous interactions. The schemas are described with a case study in the domain of smart cities and evaluated with a framework that considers four functional scalability requirements: *explicit control flow*, *separation of control and computation*, *distributed workflows* and *location transparency*. The evaluation aims to determine how well the schemas fulfil the framework desiderata. Results show that multi-level exogenous interactions best meets the requirements. As DX-MAN results in such run-time interactions, the paper [AL18b] suggests that algebraic composition can be suitable for tackling the functional scale of IoT systems.

The last paper [AL19b] systematically reviews and evaluates existing IoT service composition mechanisms, namely (centralised and distributed) dataflows, (centralised and distributed) orchestration and choreography. Unlike [AL18b], it focuses on service composition mechanisms that define composite services at design-time, not on service interactions that happen at run-time. Thus, the papers complement each other. In particular, [AL19b] presents a qualitative evaluation based on an extension of the framework initially discussed in [AL18b]. It additionally considers *compositionality*, *separation of data*, *decentralised data flows* and *workflow variability*. Compositionality is further considered because it is the key enabler of workflow variability. Overall, [AL19b] describes functional scalability requirements in terms of a large-scale IoT scenario in the smart parking domain, and presents a systematic comparison of DX-MAN with the related work by showing the degree of satisfaction of such requirements. The results show that DX-MAN is promising for the construction of large-scale IoT systems since it fulfils all the framework desiderata.

Figure 1.7 illustrates the relationship between the included papers, the research questions and the thesis contributions. It shows that paper [AL19b] addresses the questions *RQ1*, *RQ2*, *RQ3* and *RQ6*, where *Contribution1* derives from *RQ2* and *Contribution2* results from *RQ1*. Paper [AL18b] also works on *RQ1* and results in *Contribution3* since it additionally addresses *RQ4* and *RQ5*. Notably, the question *RQ6* is the most complex one because it is investigated by [AL17a, AL17b, AL18a,

AL19a, AL19b, AL19c] in a collaborative endeavor to develop, validate and evaluate the novel service composition model that this thesis proposes, i.e., *Contribution4*. For concreteness, Figure 1.7 also shows that *Contribution4.1* and *Contribution4.2* derive from *Contribution4*.

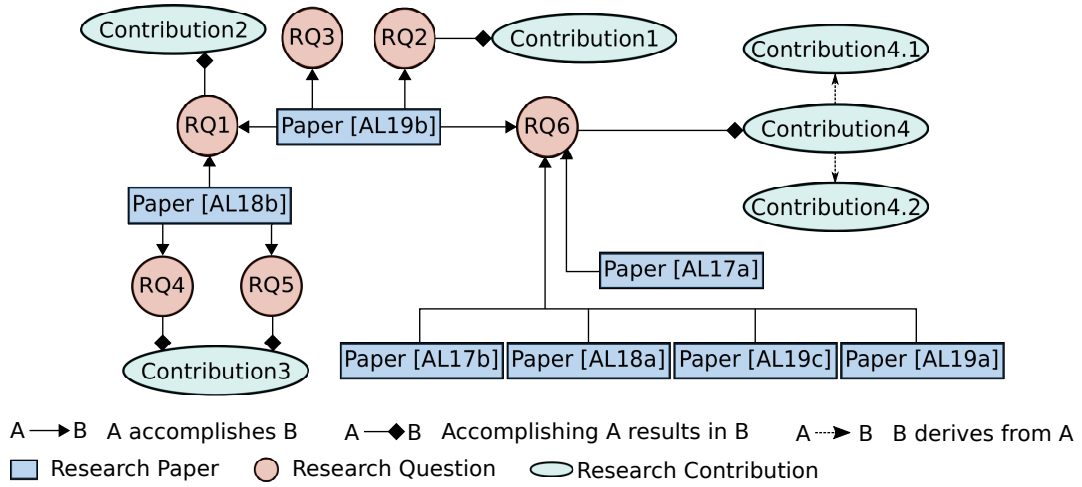


Figure 1.7: Relationship between the included papers, research questions and contributions.

Figure 1.8 illustrates the relationship between the included papers and functional scalability requirements. In particular, paper [AL17b] evaluates *explicit control flow*, *distributed workflows*, *location transparency* and the *separation of concerns* in the DX-MAN model. Papers [AL18a] and [AL19c] collaboratively analyse *workflow variability*, while article [AL19a] individually focuses on *decentralised data flows*. The DX-MAN platform [AL17a] validates all the functional scalability requirements.

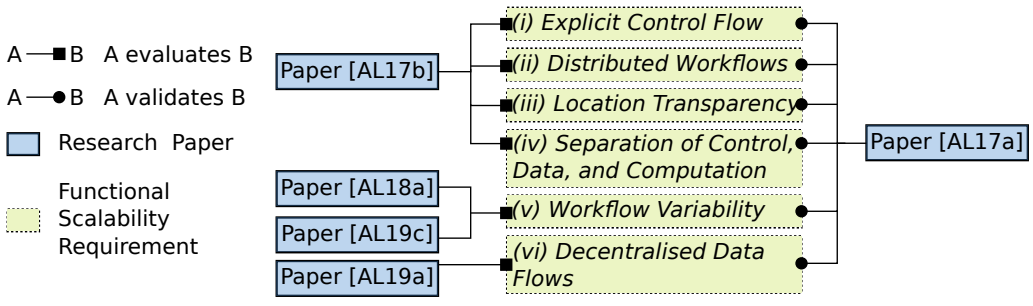


Figure 1.8: Relationship between the included papers and functional scalability requirements.

The rest of the dissertation is structured as follows. *Chapter 2* presents an overview

of IoT services, scalability and service composition. *Chapter 3* comprehensively describes the state of the art on IoT service composition by discussing orchestration, choreography and dataflows. Since the thesis borrows the notion of exogenous connectors from the X-MAN component model, Chapter 3 also describes such a model. *Chapter 4* presents the core contributions of the thesis in the form of seven original research papers by myself and my supervisor. Finally, *Chapter 5* draws the conclusions by bringing the thesis together, discussing limitations, providing a critical evaluation of the findings and setting out open challenges for future work.

Chapter 2

Universe of Discourse

"We do not learn, and that what we call learning is only a process of recollection."

— Plato, *385 B.C.E.*

This chapter outlines the universe of discourse for the rest of the dissertation. It presents an overview of IoT services, scalability and service composition.

2.1 IoT Services

Kevin Ashton coined the term *Internet of Things* (IoT) in a presentation made in 1999 at Procter and Gamble [Ash09], referring to the interconnection of everything via the Internet for the creation of a ubiquitous computing environment [Wei91]. As per the recommendation of ITU-T Y.4000, IoT has been recently redefined as *"a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies"* [ITU12].

A *thing* is practically a physical or virtual construct of the real world, which is capable of being identified and integrated into communication networks through specific protocols. The difference between physical and virtual things lies in their tangibility [Pop78]. A physical thing is a tangible object of the physical world, which is capable of being sensed, actuated and connected, e.g., home appliances, robots, buildings, plants and people. Contrastingly, a virtual thing is a non-tangible construct formed from a human idea which only exists in the information world, e.g., Clouds and enterprises.

The broad range of available things inevitably requires dealing with a high degree of heterogeneity in a distributed environment. Accordingly, SOA represents the best

way of dealing with this issue [CGB16], as it is *"a logical way of designing a software system to provide services either to end-user applications or other services distributed in a network, via published and discoverable interfaces"* [PTDL07]. Thus, it is expected that physical and virtual things will provide services to expose behaviour via interfaces [AHGZ16, BWN16, NPCA16, XHL14, GTK+10, ARJ18].

According to the Oxford dictionary, the word *service* can be a noun or a verb referring to the *action of helping or doing work for someone*. Considering a service as a noun allows the encapsulation of behaviour (i.e., actions) in the form of operations described as verbs. Thus, *a software service is a distributed component that provides a set of operations through network-accessible endpoints* [GGKS02, CDK+02, PZL08]. In general, IoT services are virtual representations of the behaviour of things, which can be combined with other services into more complex behaviours to yield complex, service-oriented IoT systems [PKGZ08, GTMW11, GTK+10, CGD14]. Thus, things are integrated through the composition of the services they provide (see Section 2.2).

Resource-constrained things (e.g., pulse sensors) typically provide fine-grained services for basic functionality (e.g., fetching sensor data), whilst non-resource constrained things (e.g., a Cloud) may offer coarse-grained services in addition (e.g., services for geolocation or complex industrial processes). Enterprise services are typically coarse-grained as they are deployed on infrastructures that have a lot of resources, whilst services of physical things are often fine-grained because they are usually deployed on resource-constrained things.

Figure 2.1 shows the relationship between things, services and operations. Figure 2.1(a) depicts a *washing machine* (i.e., a resource-constrained physical thing) that offers the *Washing* and *Drying* fine-grained services with two operations each (for starting and stopping the respective processes). Figure 2.1(b) shows a *City Council Cloud* (i.e., a non-resource constrained virtual thing) that offers the services *CouncilTax* and *Parking*. The *CouncilTax* service provides the operations *pay* (for paying a tax bill) and *check* (to query council tax information). The *Parking* service offers the operations *getNearest* (for getting the closest parking space from a driver's location) and *registerDisabled* (for registering an impaired driver).

The rest of the dissertation uses the notation *S.O* to denote an operation *O* in service *S*, e.g., *Parking.getNearest* refers to the *getNearest* operation provided by the *Parking* service.

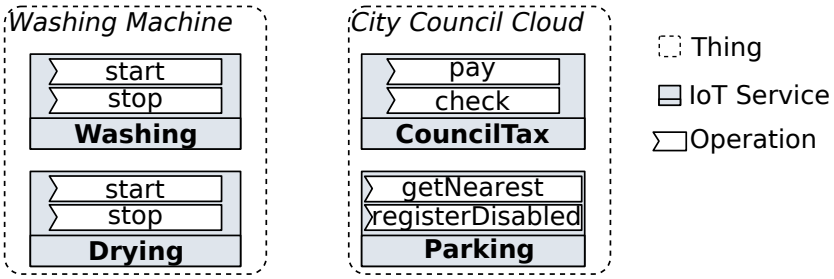


Figure 2.1: Relationship between things, IoT services and operations.

2.2 IoT Service Composition

An IoT service is a distributed unit of composition, which constitutes the virtual representation of a thing's behaviour, and can be either atomic or composite. An *atomic service* is a well-defined and self-contained piece of behaviour that cannot be divided into other services [BD13, LDC17, Bel10]. A *composite service*, on the other hand, is a more complex unit that provides value-added functionality and is formed by the combination of (atomic or composite) services [BD13, JGB17, LDC17, PKGZ08, Gui09, GTK+10, CGD14]. For example, a humidity sensing service can be combined with a temperature service into an air conditioning composite [BBDL+13].

The ability of combining services is referred to as *compositionality* and is realised by a *composition mechanism* [LDC17]. Thus, an IoT system requires a things infrastructure, the definition of what a service is and the selection of a composition mechanism [RAF+17]. In any scenario, composition is done regardless of both the technologies being used and the things infrastructure. Service technologies include REST [Pau09, Fie00], WS-* [PZL08, GIM12], OSGi [LWKH17, RVC+07] and many others.[1] Figure 2.2 illustrates the concept of IoT service composition.

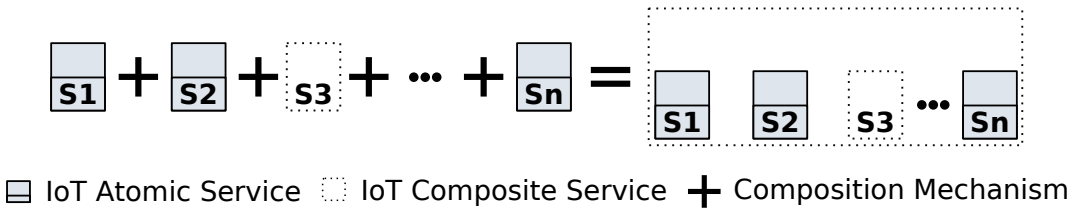


Figure 2.2: IoT service composition.

A service composition mechanism defines a meaningful interaction between services [LDC17] by considering two functional dimensions: control flow and data flow

[1] In RESTful services, operations are exposed as resources [Pau09].

[PLB+17, LDB15]. Control flow refers to the order in which interactions occur [DD04, AL18b], whilst data flow defines how data is moved among services [PLB+17]. This thesis focuses on service composition mechanisms that define behaviour by workflows.

A workflow describes a series of discrete steps for the realisation of a computational activity, which can be control-driven, data-driven or hybrid [Shi07]. In a control-driven workflow, steps (also known as tasks [Ama19a], actors [LAB+06], transitions [Pet62], procedures [ZWF07], thorns [GAL+03], activities [OAS07] and units [TSWH07]) are executed according to explicit control flow constructs that define sequencing, looping, branching or parallelising. A data-driven workflow invokes steps whenever data becomes available without explicitly defining any control flow constructs [Ope19]. In a hybrid workflow, some parts are control-driven while others are data-driven [TSWH07]. Figure 2.3 illustrates a generic workflow that executes the operation op_1, then decides to invoke either op_2 or op_3 and, finally, triggers the operations op_4 and op_5 in parallel. Operation invocations happen regardless of the workflow kind.

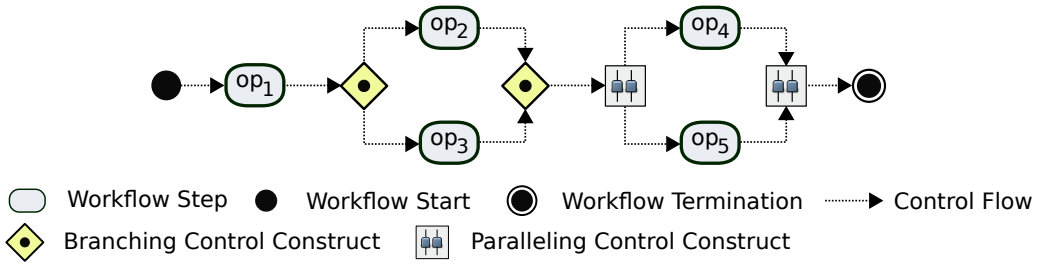


Figure 2.3: A generic workflow.

2.3 Algebraic Service Composition

Mathematical functions can be composed into other (higher-order) functions [Dev04], and two mathematical functions are composable if they have compatible input/output types. For example, $f : X \to Y$ and $g : Y \to Z$ can be composed into a higher-order function $g \circ f : X \to Z$ s.t. $X \to Y$ and $Y \to Z$ are type signatures. This composition can also be expressed as $g(f(x))$ or $(xf)g$ where $x \in X$ [Gal11]. Figure 2.4 shows that, intuitively, the input of the outer function g is the output of the inner function f.

Mathematical functions are composed algebraically and, therefore, hierarchically in a bottom-up fashion. Algebraic service composition is thus the process by which a

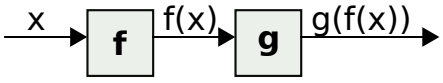


Figure 2.4: Mathematical function composition.

composition operator hierarchically composes multiple services of type $\mathbb{S}$ into a composite service of type $\mathbb{S}$. The resulting composite can be further composed into even more complex composites, and has an interface constructed from the sub-service interfaces [LDC17, AL17b, AL19c]. Formally, an algebraic service composition operator is a function $\circ$ with the following type:

$$\circ : \mathbb{S} \times \mathbb{S} \times \cdots \times \mathbb{S} \rightarrow \mathbb{S} \tag{2.1}$$

Applying $\circ$ does not require any glue code that has to be defined manually for the systematic construction of (hierarchical) composite services [LDC17]. Figure 2.5 shows that algebraic composition produces a service of type $\mathbb{S}$ at every level of the construction hierarchy. In particular, composing the services $S_1 \in \mathbb{S}$ and $S_2 \in \mathbb{S}$ results in the composite $S_5 \in \mathbb{S}$, i.e., $\circ(S_1, S_2) = S_5$. Likewise, composing the services $S_3 \in \mathbb{S}$ and $S_4 \in \mathbb{S}$ produces the composite $S_6 \in \mathbb{S}$, i.e., $\circ(S_3, S_4) = S_6$. At the top level, $S_5 \in \mathbb{S}$ and $S_6 \in \mathbb{S}$ are further composed into $S_7 \in \mathbb{S}$ which is the most complex composite in the hierarchy construction, i.e., $\circ(S_5, S_6) = \circ(\circ(S_1, S_2), \circ(S_3, S_4)) = S_7$. It is important to note that this notion of composition only considers homogeneous algebras, not heterogeneous algebras where a composite would be of the same type as at least one of the composed services [LDC17, LR10].

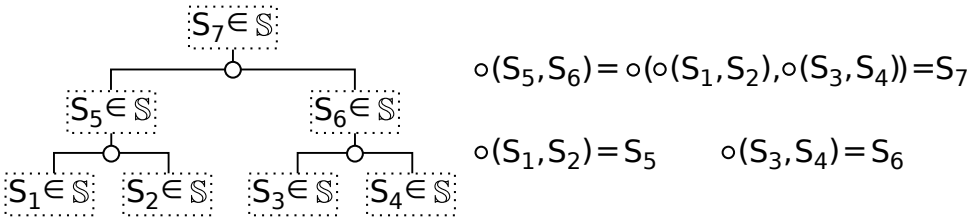


Figure 2.5: Algebraic service composition.

2.4 Scalability of IoT Systems

Typically, scalability is the capability to handle increasing workloads in an IoT system [HN18, VN17, ARJ18, MVT17, SA16, LLZ14]. In this case, it is a metric that

indicates how system performance improves over time. Workloads are usually measured in terms of either the number of requests dispatched [HN18] or the data streams generated [SA16]. The overall goal of scalable solutions is to enhance the Quality of Service (QoS) for guaranteeing a certain level of performance under the presence of high workloads, e.g., by minimising bandwidth, energy, latency and response time while maximising throughput. To quantitatively measure QoS, several network aspects of a service are considered such as jitter, packet loss and availability [LLZ14].

Currently, there are two kinds of scalability: vertical and horizontal.[2] Vertical scalability (or scaling up) [SBAB19, CPS17, RMBG18] refers to the addition or removal of computing resources in a single thing. For example, adding more memory to increase the buffer size or adding more processor capacity to speed up processing. On the other hand, horizontal scalability (or scaling out) [CSB19, WLB09, SNP+15] involves the addition or removal of things that participate in an IoT system. Its goal is to distribute the workload over multiple things so as to decrease individual loads, minimise the response time and enhance concurrency. Figure 2.6 depicts the contrast between vertical and horizontal scalability.

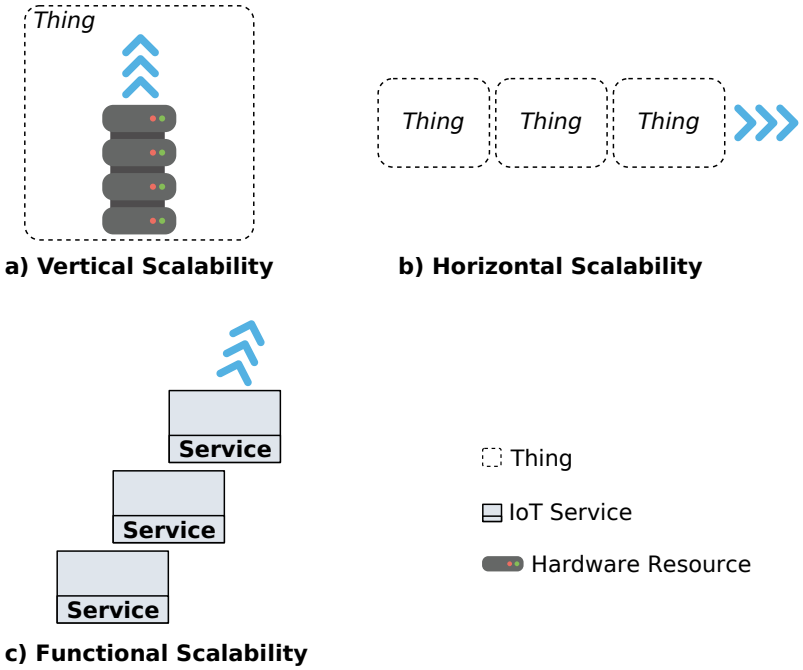


Figure 2.6: Scalability of IoT systems.

Both vertical and horizontal scalability have been extensively addressed in the literature [CGK+11, MSR17, XH16, GMA17, CSC+18, CPS17, RMBG18, CdCSR+15,

[2] IoT cloud environments benefit from dynamically scaling vertically, horizontally or both.

BD16, VN17], unlike *functional scalability* which this thesis refers to as the capability to accommodate growth in terms of the number of services composed in an IoT system (see Figure 2.6(c)). Functional scalability enables the composition of any number of services, without severely impacting both system behaviour and global system properties such as performance, maintenance, evolution and monitoring. Figure 2.7 shows that functional scalability is orthogonal to vertical and horizontal scalability.

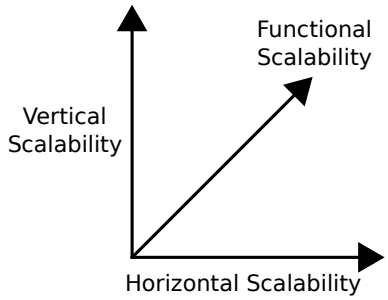


Figure 2.7: Scalability dimensions.

Like the other kinds of scalability, functional scalability requires the definition of metrics to measure the degree of satisfaction for accommodating new services. This thesis proposes six qualitative metrics discussed in Chapter 4. It does not claim that such metrics are complete since quantitative metrics for QoS, identified for vertical and horizontal scalability, can also be important. However, quantitative metrics are only applicable to specific implementations. As service composition is an abstraction rather than a concrete implementation, this thesis strongly argues that qualitative metrics are the best ones to measure the degree of satisfaction of functional scalability in service composition mechanisms. For the rest of the dissertation, the terms scalability and functional scalability are used interchangeably.

Chapter 3

Related Work

> *"Simplicity is a great virtue, but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."*
>
> — Edsger Dijkstra, *1984*

This chapter reviews the fundamental semantics of current IoT service composition mechanisms that define workflows: (centralised and distributed) dataflows, (centralised and distributed) orchestration and choreography. The semantics refers to how to compose services and underlies so-called composition algorithms [WZY+19, LLZ14, JM15, KAC+16, DXY+18, YLC14, SSC+17, AMC18, HW16, ZZLH18, LDCN14, ASC14, BAA19, BYDA18, UGBBM+17], programming frameworks [SBWS+17, LLW15, XV14, KDB15, VGS+13, CGV+18, KKMK16], languages [ÅHNM19, MGZ14] and platforms [KHDB+17, NGM+17, MBB18, ASM+19, RVHTGGMC14, KSRD14, CSGD+14], which have been somehow confusingly included in "IoT service composition" surveys [HN18, AIM10, ARJ18, HS19, ARJ19].

It is essential to mention that, in the literature, IoT service composition is just another name for traditional SOA composition and is done regardless of service "architectures." Microservice Architecture [LSCPE18, CS16, New15] has gained considerable attention in the last few years and is becoming increasingly important and popular for the development of IoT systems [FPSC18, KHLL18, CCG+18, QNG+18, TVS18]. Every Microservice Architecture is an SOA, but not the other way round [Zim17]. Hence, the composition mechanisms presented in this chapter can be used interchangeably in both Microservices and traditional SOA services. In fact, there are no composition mechanisms specifically aimed for Microservices.

Although it does not have any support for IoT services, this chapter also presents an overview of the X-MAN component model from which the proposed model borrows the notion of exogenous connectors.

3.1 Dataflows

Dataflows, or *Flow-Based Programming* [Mor10, JHM04, Mor78], is a composition mechanism that defines a workflow using data transformations (e.g., filter, split, union and sort) as well as exogenous data exchange between services [PLB+17, ACKM04]. A dataflow description is a directed graph where vertices are asynchronous data processing units (invoking service operations), and edges are connections for passing data streams between vertices via the network (by message passing or events). A vertex explicitly defines input ports and output ports. When it receives data from all inputs, it performs some computation and writes results in output ports. The resulting data is then moved to other vertices via an edge. This process is illustrated in Figure 3.1.

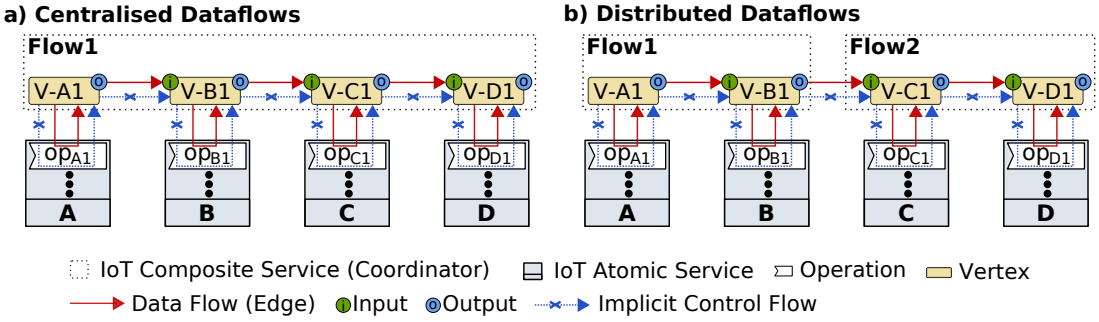


Figure 3.1: Composition by dataflows.

Dataflows can be centralised or distributed. A centralised dataflow [Ope19, KSRD14] defines a single coordinator for managing an entire graph and exogenously invokes service operations. A distributed dataflow [GBLL15, GLL18, GLL19, NTGS19] partitions and distributes a complex graph over multiple coordinators that interact directly by exchanging data between vertices. Figure 3.1(a) shows a centralised dataflow for a pipeline of services *A*, *B*, *C* and *D*. When the coordinator *Flow1* is triggered, vertex *V-A1* invokes $A.op_{A1}$ and passes the result to the vertex *V-B1* which, in turn, executes $B.op_{B1}$. Next, the result of *V-B1* is passed to the vertex *V-C1* which executes $C.op_{C1}$. Finally, the data of *V-C1* is moved to the vertex *V-D1* and then processed by $D.op_{D1}$. A distributed version of the same pipeline is shown in Figure 3.1(b), where

there is an edge between *V-B1* and *V-C1* for moving data from the *Flow1* composite to the *Flow2* composite.

Dataflows are increasingly popular for composing IoT systems. In particular, they are widely used for the Internet of Data (IoD) [FCXC12] which involves data collection from multiple sources (e.g., sensors), data analysis and control of the physical world. This paradigm has been referred to as Sense-Compute-Control (SCC) [TMD09].[1] Currently, there are many platforms for composing IoT services by dataflows. Examples include Node-RED [Ope19], COMPOSE [PVC+14], Glue.Things [KSRD14], LabVIEW [Nat19], Paraimpu [PCP12], Virtual Sensors [KHDB+17], SpaceBrew [Spa19], FogFlow [CSC+18], ASU VIPLE [DLLMC18], ThingNet [QNG+18], Calvin [PA15], IoT Services Orchestration Layer [Int19], NoFlo [NoF19] and many others [PBT+19, TKY+17, MVF+15, HLR17].

As the Web 2.0 became more data-centric and user-friendly [Pau09, PLB+17], dataflows have gained popularity for IoD through mashups. Mashups [PG17] are realised by dataflows [PLB+17, CBZF16], and they allow the composition and visualisation of data streams on a graphical user interface displayed on the Web [PLB+17, DM14, Pau09]. Examples of IoT mashup tools include WoTKit [BL12], IoTMaaS [IKK13] and Clickscript [GTMW11].

Dataflows have been accepted as coordination languages because a graph is defined in a coordinator that exogenously invokes services according to a dataflow description [JHM04, GBLL15, Mor10]. A dataflow graph is typically created with a graphical editor and executed by an engine (i.e., the coordinator), and it is triggered by either timing constraints or events.

Formally, flow-based programming is not algebraic because it can be defined as a function DFL with the following type:

$$DFL : \mathbb{OP} \times \mathbb{OP} \times \cdots \times \mathbb{OP} \mapsto \mathbb{WF} \tag{3.1}$$

where $\mathbb{OP}$ is the service operation type and $\mathbb{WF}$ is the type of workflows that invoke a set of operations of type $\mathbb{OP}$.

[1] Do not confuse data analysis tools [NPPZ18, GDG12] with dataflows. A data analysis tool is a software that allows the collection, storing, indexing, processing, monitoring and visualisation of data. On the other hand, dataflows specify how data is passed between services, according to a dataflow graph specification.

3.2 Orchestration

Orchestration can be centralised or distributed. Centralised orchestration [Pel03, BDO05, SQV+14, LDB15] describes interactions between services from the perspective of a central coordinator (also known as orchestrator) which has control over all parties involved. It explicitly defines workflow control flow to coordinate the invocation of service operations, in order to realise some complex function that cannot be achieved by any individual service [LWKH17, DP06, RVC+07]. In a distributed orchestration, also known as "decentralised orchestration" [HSPDB14, JDB16, NCS04, CCMN04, PPT14, WML08, MWL10, SGK+10, KLS+10, FDGGB14], multiple coordinators collaboratively define workflow control flow.

An orchestration is typically defined using a workflow language such as BPEL [OAS07, SKG+09, RVC+07, PCPG10, GEPF11, Ove08, CRWS12, DMMS10] or BPMN [OMG11, MCS16]. The resulting workflow has tasks for passing control among services according to explicit control flow constructs (for sequencing, parallelising, branching and looping) [PLB+17]. In distributed orchestration, the interaction between coordinators can be done in three different ways: (i) *with an extra task* [JDB16], (ii) *with two additional tasks* [NCS04, CCMN04, FDGGB14] or (iii) *without any extra task* [MWL10, WML08, HSPDB14]. In the former interaction, an orchestration invokes the interface of another orchestration using an external task to the system's workflow control flow. In the second interaction, there are two different tasks for receiving and passing control (and data) between two orchestrations. Finally, in the last interaction style, two orchestrations interact by moving control (and data) directly between the tasks of the system's workflow control flow.

An orchestration engine is responsible for executing a workflow process by invoking service operations in a given order. Although traditional engines can be used (e.g., Camunda BPM workflow engine [Cam19, MCS16], Activiti [Rad12, PKAK18] and AWS Step Functions [Ama19b]), recently we have seen the emergence of orchestration engines particularly designed for IoT systems (e.g., PROtEUS [SHS15] and [CZCC18]).[2]

Figure 3.2(a) illustrates a centralised orchestration for the services A, B, C and D, where the coordinator *Orch1* defines a "composite service" for the sequential invocation of $A.op_{A1}$, $B.op_{B1}$, $C.op_{C1}$ and $D.op_{D1}$. Three distributed versions are depicted in Figure 3.2(b). In the former, *Orch1* defines a "composite service" for the sequential

[2] A workflow engine can be deployed on either a specialised server [Cam19] or a service bus like a Gateway [SHLP05, Jos07, NGM+17].

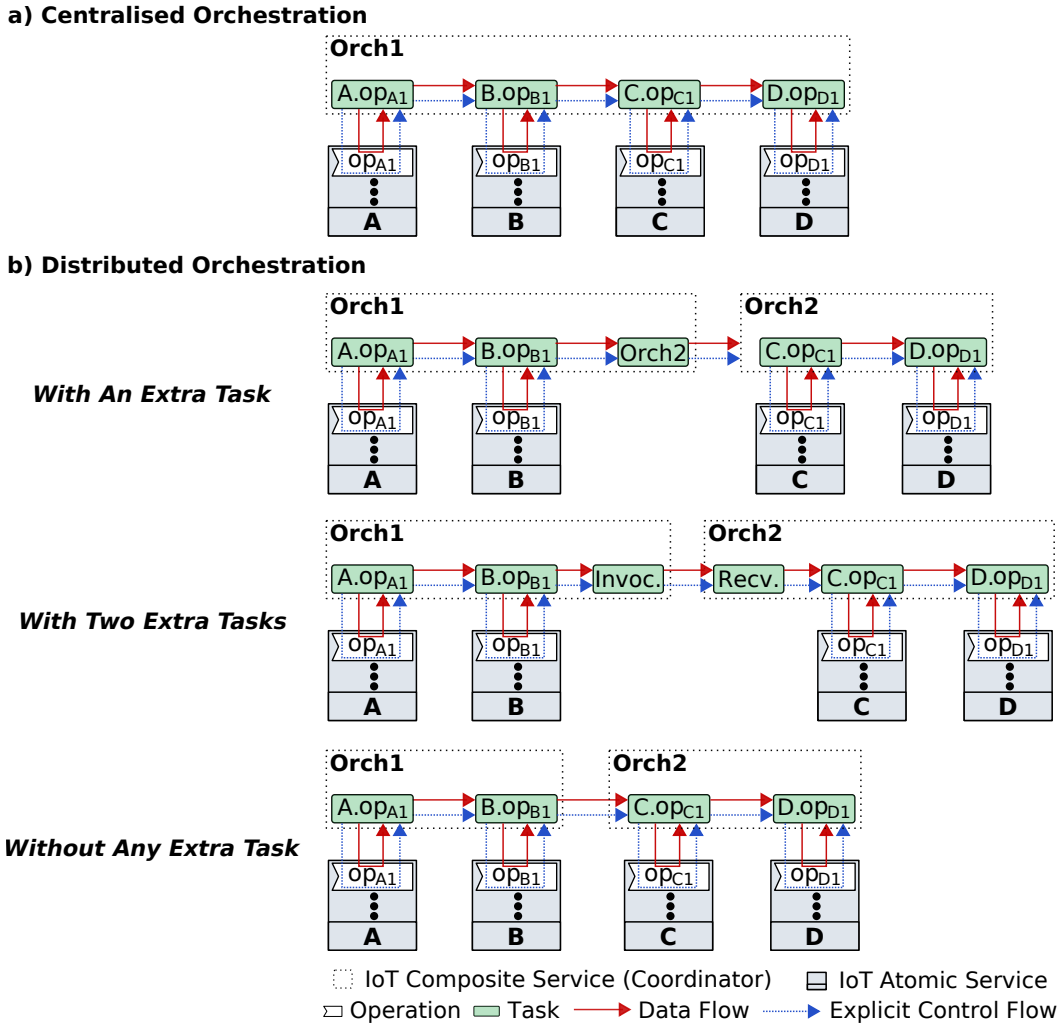


Figure 3.2: Composition by orchestration.

execution of $A.op_{A1}$ and $B.op_{B1}$, and then uses an extra task to pass control (and data) to *Orch2*. *Orch2* defines another "composite service" to sequentially invoke $C.op_{C1}$ and $D.op_{D1}$. The second distributed version uses two extra tasks for passing and receiving control (and data) between *Orch1* and *Orch2*. Finally, in the last distributed version, control and data are passed directly from $B.op_{B1}$ to $C.op_{C1}$.

Formally, orchestration is not algebraic because it can be defined as a function *ORCH* with the following type:

$$ORCH : \mathbb{OP} \times \mathbb{OP} \times \cdots \times \mathbb{OP} \mapsto \mathbb{WF} \tag{3.2}$$

where $\mathbb{OP}$ is the service operation type and $\mathbb{WF}$ is the type of workflows that invoke a set of operations of type $\mathbb{OP}$.

3.3 Choreography

A *choreography* describes service interactions from a global perspective using a public contract (also known as protocol) [BWR09, Pel03, BDO05, SQV+14, DKB08]. The contract specifies a "conversation" among participants via decentralised message exchanges, and it can be modelled by a global observer using a choreography modelling language [SQV+14, DKB08, New15].[3] An interaction-based modelling language allows the definition of event-driven or request-response messages by connecting required and provided interfaces. Examples include WS-CDL [RTF06] and Let's Dance [ZBDH06]. An interconnected interface model, on the other hand, allows the specification of explicit control flow per participant. Examples include BPEL4Chor [DKLW07], Web Service Choreography Interface (WSCI) [AAF02] and BPMN [DB07].[4]

A public contract defines roles for the collaborative realisation of a global workflow, where there is no control over the internal details of the participants involved [DP06]. A role explicitly describes a participant workflow control flow in terms of expected and produced messages. When a concrete service instance plays a role, it must behave accordingly by exchanging messages with other instances, using either direct message passing (e.g., invoking REST APIs) or events [AL18b, New15, CSGD+14, BBDL+13].[5] This process is known as choreography enactment.

IoT is moving towards a more decentralised environment to reduce the bottleneck caused by centralised environments. As choreographies represent a natural way of dealing with such a decentralisation, there are currently some platforms for composing IoT services by choreographies, e.g., CHOReVOLUTION [OW2], ChorSystem [WAS+16], Actorsphere [Act19], *BeC*^{3} [CSGD+14] and TraDE [HBKL18].

Figure 3.3 shows a sequential choreography for the services A, B, C and D, where a protocol (defined with standard BPMN 2.0 notation [OMG11]) specifies that $B.op_{B1}$ expects a message from $A.op_{A1}$, $C.op_{C1}$ a message from $B.op_{B1}$ and $D.op_{D1}$ a message from $C.op_{C1}$. When the choreography is enacted, there is a chain reaction that starts with the invocation of $A.op_{A1}$ and finishes with the execution of $D.op_{D1}$.

[3] A Microservice architecture prefers choreography over orchestration to support decentralised workflows [New15].

[4] The choice of the contract depends on the type of participants involved which can be either atomic services or orchestrations.

[5] Service participants are tightly coupled in terms of dependencies. In choreographies based on direct message-passing, services hardcode invocation calls in service computation. In event-driven choreographies, services are tightly coupled because senders and receivers agree on a topic queue in advance [ZBL17].

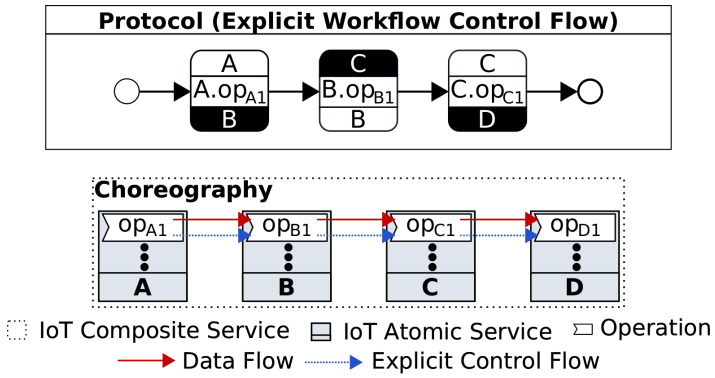


Figure 3.3: Composition by choreography.

Formally, choreography is not an algebraic composition mechanism because it can be defined as a function *CHOR* with the following type:

$$CHOR : \mathbb{OP} \times \mathbb{OP} \times \cdots \times \mathbb{OP} \mapsto \mathbb{WF} \tag{3.3}$$

where $\mathbb{OP}$ is the service operation type and $\mathbb{WF}$ is the type of workflows that invoke a set of operations of type $\mathbb{OP}$.

3.4 X-MAN Component Model

The X-MAN component model [LVEW05, LT12] considers components and exogenous connectors as first-class entities (see Figure 3.4). A component can be either atomic or composite, and it is a passive unit of composition that exposes behaviour via an interface of provided services. Exogenous connectors encapsulate control flow to initiate and coordinate the execution of a component-based software system. The coordination of control occurs from outside components.

An atomic component is the most primitive composition unit in the X-MAN component model. It is constructed by connecting an invocation connector with a computation unit which encapsulates the implementation of some behaviour in a chosen programming language (see Figure 3.4(a)). For ensuring component encapsulation, computation units cannot call one another.

A composite component is formed by connecting a composition connector with multiple (atomic or composite) sub-components (see Figure 3.4(c)). A composition connector coordinates the execution of sub-components by passing control and data, and can be defined for exclusive branching and sequencing (see Figure 3.4(b)). In

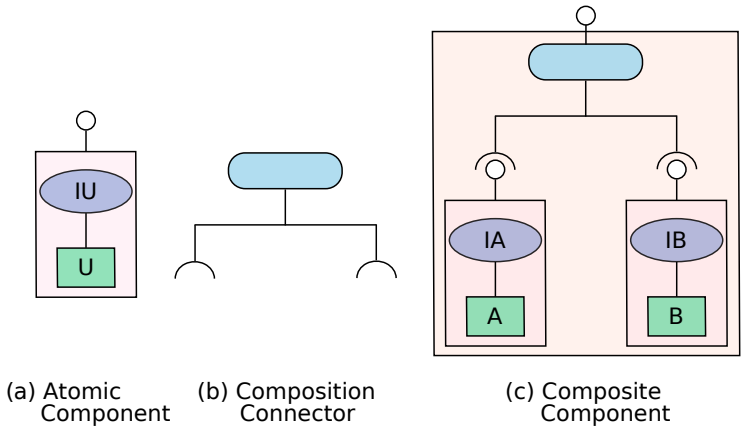


Figure 3.4: The X-MAN component model.

particular, a sequencer connector allows the composition of $C_1, \ldots, C_n$ components and executes them in a predefined sequential order. An exclusive branching connector enables the composition of components $C_1, \ldots, C_n$ and chooses which component to invoke according to a fixed condition. There are also exogenous connectors that are not used for composition, but for adapting the behaviour of individual components. These special connectors are called adapters and can define looping or guarding.

The X-MAN component model enables a hierarchical, incremental, bottom-up construction of software systems, where every composition produces a composite that can be further composed with other components. This process is also referred to as algebraic composition by which a software system exhibits a self-similar composition structure (see Figure 3.5). Thus, the execution of an X-MAN composition starts at the top-level connector and traverses the rest of the connectors according to control flow constructs. When control reaches an invocation connector, a computation is triggered and then control returns upwards.

X-MAN services are interfaces that require inputs and provide outputs to the external world. To define a data flow, such a component model allows the connection of a data channel between two services. During a system execution, each exogenous connector reads and writes data values on associated data channels. Horizontal data routing happens between the sub-components within a composite, whereas vertical data routing occurs between the interface of a composite and its sub-components. See [LT12] for further details on this matter.

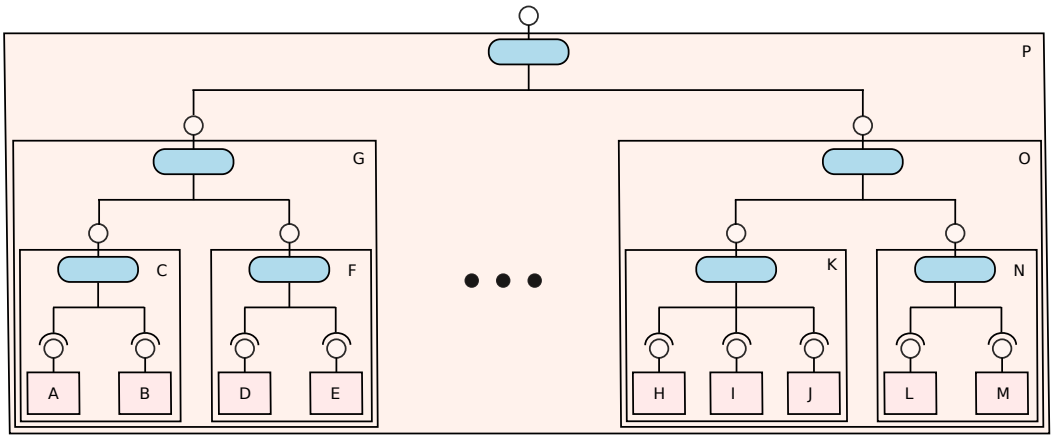


Figure 3.5: Self-similarity in the X-MAN component model.

3.4.1 X-MAN vs. DX-MAN

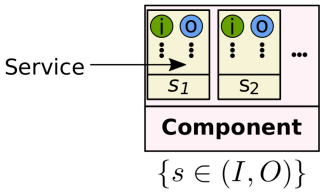
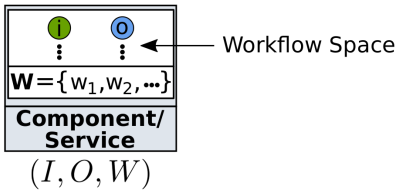
The proposed model, DX-MAN, borrows the notion of exogenous connectors from the X-MAN component model. However, there are substantial differences between them.

X-MAN is a general-purpose component model for building single-process software systems. Contrastingly, DX-MAN is a distributed, multi-process model particularly designed for the construction of service-oriented IoT systems. For that reason, DX-MAN components are services that reside on different things so exogenous connectors exchange control in a distributed fashion via the network. In addition to sequencing, exclusive branching, looping and guarding, DX-MAN provides exogenous connectors for parallelising and inclusive branching. Furthermore, adapters are not limited to looping and guarding only, since their semantics allow any control flow structure supported by the model.

As DX-MAN follows SOA principles for the definition of services and service composition, the fundamental semantics of what a service is and how to compose are completely different. A DX-MAN service is a component semantically equivalent to a workflow space that defines a family of workflow control flow variants. On the other hand, an X-MAN service is just a component interface, with inputs and outputs, which defines only one workflow at a time. Formally, the interface of an X-MAN component is a set of services $\{s \in (I,O)\}$ s.t. each service s is a tuple consisting of a (non-empty) finite set of data inputs I and a (non-empty) finite set of data outputs O. Remarkably, the interface of a DX-MAN service is a tuple (I,O,W) consisting of a finite set of data inputs I, a finite set of data outputs O and a (finite or infinite) set of workflows W.

In terms of data management, DX-MAN supports decentralised data flows for an

optimal QoS in data-intensive IoT systems. This exchange is by no means possible in the X-MAN component model where data always follows control. Table 3.1 summarises the comparison.

	X-MAN	DX-MAN
Component Interface		
Control Flow Constructs Supported by Composition Connectors	- Sequencing - Exclusive Branching	- Sequencing - Exclusive Branching - Parallelising - Inclusive Branching
Control Flow Constructs Supported by Adapters	- Looping - Guarding	- Sequencing - Exclusive Branching - Inclusive Branching - Looping - Guarding - Parallelising
Distributed Workflows	✗	✓
Number of Workflows per Composite	1	[1, ∞]
Decentralised Data Flows	✗	✓
Separation of Control/Data/ Computation	Control/Computation	Control/Data/Computation

i Input Data o Output Data ✓ Supported ✗ Not Supported

Table 3.1: X-MAN vs. DX-MAN.

3.5 Analysis of Related Work

Table 3.2 summarises the results of the analysis of existing IoT service composition mechanisms with respect to the functional scalability desiderata identified [AL18b, AL19b]: (i) explicit control flow; (ii) service location transparency; (iii) distributed workflows; (iv) decentralised data flows; (v) separation of control, data and computation; and (vi) workflow variability. Requirements (i), (ii), (iii), (iv) and (vi) are binary because they can be either supported (i.e., a tick mark) or not supported (i.e., a cross mark).

	Centralised Dataflows	Distributed Dataflows	Centralised Orchestration	Distributed Orchestration	Choreography
Explicit Control Flow	✗	✗	✓	✓	✓
Location Transparency	✓	✓	✓	✓	✗
Distributed Workflows	✗	✓	✗	✓	✓
Decentralised Data Flows	✗	✗	✗	✗	✓
Separation of Control/Data/Computation	Data/ Computation	Data/ Computation	Control/ Computation	Control/ Computation	None
Workflow Variability	✗	✗	✗	✗	✗

Table 3.2: Analysis of of existing IoT service composition mechanisms w.r.t. functional scalability desiderata of IoT systems.

Centralised dataflows is the worst mechanism because it only supports one binary requirement (i.e., location transparency) and separates two concerns (i.e., data and computation). Distributed dataflows fulfils scalability requirements with a slightly higher degree because it additionally offers distributed workflows. Although centralised orchestration provides the same satisfaction degree as distributed dataflows, it supports different requirements (i.e., explicit control flow and location transparency) and separates different concerns (i.e., control and computation). Distributed orchestration is similar since it supports distributed workflows in addition. Finally, choreography covers three binary requirements (i.e., explicit control flow, distributed workflows and decentralised data flows) and does not provide any separation of concerns since control and data are mixed in service computation. The X-MAN component model is not considered in the analysis because it does not have any support for IoT services.

Overall, none of the existing IoT service composition mechanisms fulfil all the functionality scalability desiderata. The included paper [AL19b] presents a detailed and extended analysis of the results.

Chapter 4

Commented Collection of Original Publications

"If the doors of perception were cleansed then
everything would appear to man as it is, Infinite."
— William Blake*, 1793*

This chapter presents the core contributions of the thesis in the form of six published peer-reviewed papers and one submitted manuscript. The published papers appear in international conference proceedings, and the other article was submitted to a journal with a high impact factor. The contributions are presented in the order discussed in Section 1.6, and Appendix A shows the formal permissions for reusing them.

For each publication source, this chapter shows impact measurements in terms of ranking (if available), acceptance rate (if available in the publication year), impact factor, citation ranking (if there are citations) and awards (if any). A conference impact factor is estimated for the most recent (possible) year by analysing academic citations from Google Scholar [Goo19]. The rankings are obtained from CORE [Com18] and Qualis [Coo16] (which assess major conferences and journals in Computer Science). The citation ranking is determined by retrieving and sorting paper citations also from Google Scholar [Goo19]. For the journal submitted, we obtained the most recent impact factor, the SCImago Journal Rank and (SJR) and the Source Normalised Impact per Paper (SNIP) from the official website [Els19].

4.1 Exogenous Connectors for Hierarchical Service Composition

Damian Arellanes and Kung-Kiu Lau

In proceedings of the **10th Conference on Service-Oriented Computing and Applications (SOCA 2017)**.

Published by IEEE,
pages 125-132,
ISBN 978-1-5386-1326-9,
2017.

Impact Indicators:
• Conference Core ranking (2018): C,
• Conference impact factor (2018): 2.74,
• First most cited article amongst 34 publications in SOCA 2017 (as of 2019).

The final authenticated version [AL17b] is available online at `https://doi.org/10.1109/SOCA.2017.25`.

Summary: This paper presents an initial version of the proposed model, referred to as DX-MAN. It discusses the concept of algebraic service composition and presents a qualitative evaluation of *explicit control flow* and *compositionality*. The paper uses a case study in the domain of smart retail to further discuss *location transparency*, *workflow distribution* and the *separation of control, data and computation*.

Comments on authorship: I proposed the main idea of the paper, developed and validated the model, conducted a qualitative evaluation, analysed results, investigated related work, provided and edited all graphics, participated in the entire writing process and addressed the reviewer's comments. My supervisor, Kung-Kiu Lau, also contributed to the idea, proofread the paper and approved the model. He also guided the whole research process.

Key contributions: Contribution 4 (see Section 1.5).

Exogenous Connectors for Hierarchical Service Composition

Damian Arellanes and Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

***Abstract*—Service composition is currently done by (hierarchical) orchestration and choreography. However, these approaches do not support explicit control flow and total compositionality, which are crucial for the scalability of service-oriented systems. In this paper, we propose exogenous connectors for service composition. These connectors support both explicit control flow and total compositionality in hierarchical service composition. To validate and evaluate our proposal, we present a case study based on the popular MusicCorp.**

***Index Terms*—hierarchical service composition, scalability, orchestration, choreography, microservices, exogenous connectors**

I. Introduction

In Service-Oriented Architectures (SOA) [26], service composition is increasingly challenging as SOA systems get ever larger [10]. Therefore, the *de facto* approaches for service composition, namely (hierarchical) orchestration and choreography, need to address scalability.

Microservice architecture [7], [18] is the leading trend in SOA [26]. It prefers choreography over orchestration so as to avoid a single point of failure and attack, as well as performance bottlenecks. However, Netflix, a pioneer of this architectural style, has recently expressed that they found it difficult to scale with growing business needs by using choreographies because the implicit control flow therein is hard to visualize. For this reason, Netflix now prefers service orchestration [16].

Apart from explicit control flow, we believe that total compositionality is also crucial for scalability since it enables hierarchical construction of SOA systems. By total compositionality we mean algebraic composition, which is not present in choreography, orchestration or even hierarchical orchestration.

In this paper, we propose exogenous connectors [13], [12] for hierarchical service composition. These connectors are architectural elements that coordinate the execution of an SOA system by passing only control. Like orchestration, exogenous connectors define explicit control flow, but unlike (hierarchical) orchestration and choreography, they enable total compositionality.

The rest of the paper is organized as follows. Sect. II briefly revisits the paradigms for service composition. Sect. III describes our approach. Sect. IV presents a case study to demonstrate the suitability of our approach. Sect. V outlines a qualitative evaluation of our approach and presents a discussion of the results. Finally, Sect. VI presents the conclusion and the future work.

II. Orchestration, Hierarchical Orchestration and Choreography Revisited

In this section, we review the paradigms for service composition, namely (hierarchical) orchestration and choreography, rather than reviewing methods [24], [14], languages [9], tools or platforms [2] using these paradigms.

We believe that explicit control flow and total compositionality are crucial for the scalability of SOA systems. Explicit control flow means that an architectural entity explicitly defines the order in which individual services are executed. On the one hand, in orchestration, control flow is defined in the central coordinator [21], [9]; similarly, in hierarchical orchestration, control flow is defined in nested (inner and outer) orchestrations [8], [4], [25]. On the other hand, choreography defines control flow only implicitly, in the collaborative exchange of messages [20], [6], [21], [23]. Implicit control flow is hard to monitor, track, maintain and evolve since it is hard to visualize entirely [16], [5], [3], [18], [10].

Compositionality is assumed to be present in orchestration, hierarchical orchestration, and choreography, as a coordination of service invocations [15]. However, this is not total compositionality, by which we mean *algebraic composition*: two or more services can be composed into a new (composite) service of the same type, that preserves all the operations provided by the composed services.[1] Total compositionality implies a hierarchical composition structure but not the other way round; in fact, an orchestration can be hierarchical but not compositional.[2] Totally compositional architectures are more tractable than non-compositional architectures ones because they make it easier to evaluate the individual parts [17]. Furthermore, hierarchical construction is a well-known technique for tackling scale and complexity.

Table I shows that orchestration, hierarchical orchestration, and choreography do not define a composition of entire services, but a workflow of invocations of selected and named

[1] See 6 in Appendix A for a formal definition of total compositionality.
[2] See Appendix B.

TABLE I
COMPOSITIONALITY IN SOA.

	Resulting type of composition	Number of operations preserved from the composed services	Compositionality
Orchestration	Workflow	Number of selected and named operations	Partial
Hierarchical Orchestration	Workflow	Number of selected and named operations	Partial
Choreography	Workflow	Number of selected and named operations	Partial
Our Approach	Service	All	Total

operations in the composed services [19], [1], [20], [21], [22].[3] (Selecting and) Naming a specific set of operations results in a partial composition in which individual workflows are required for the invocation of operations in the composed services; thus, the operations that are not (selected and) named are lost so they cannot be invoked. Of course, all the operations could be included; however, the resulting workflow would be potentially complex as the number of operations increases, leading to combinatorial explosion. In contrast, our approach enables total compositionality, since the resulting type of composition is another service with all the operations of the composed services (not a workflow with selected and named operations). In a total composition, any operation of any composed service can be invoked without the need of individual workflows.

Consider two services: $S1$ which provides the operations $op11$ and $op12$, and $S2$ which provides the operations $op21$ and $op22$. Fig. 1 shows a possible composition of these services using orchestration, our approach, hierarchical orchestration and choreography. In this figure, it is clear that orchestration, hierarchical orchestration, and choreography results in a partial composition, i.e., a workflow that loses operations of the composed services. For instance, the operation $op22$ cannot be invoked in any of these approaches; any such change would require an entirely new workflow. In contrast, in our approach, the symbol # is a wildcard indicating that any operation of the service involved can be invoked, e.g., both operations $op11$ and $op12$ are available to be invoked in service $S1$.

Table II shows that orchestration, hierarchical orchestration and choreography does not support total compositionality. Only our approach supports total compositionality and, like orchestration (and hierarchical orchestration), exogenous connectors also define explicit control flow.

TABLE II
ORCHESTRATION VS. HIERARCHICAL ORCHESTRATION VS. CHOREOGRAPHY VS. OUR APPROACH.

	Total Compositionality	Explicit Control Flow
Orchestration	✗	✓
Hierarchical Orchestration	✗	✓
Choreography	✗	✗
Our Approach	✓	✓

[3]See 2 in Appendix A.

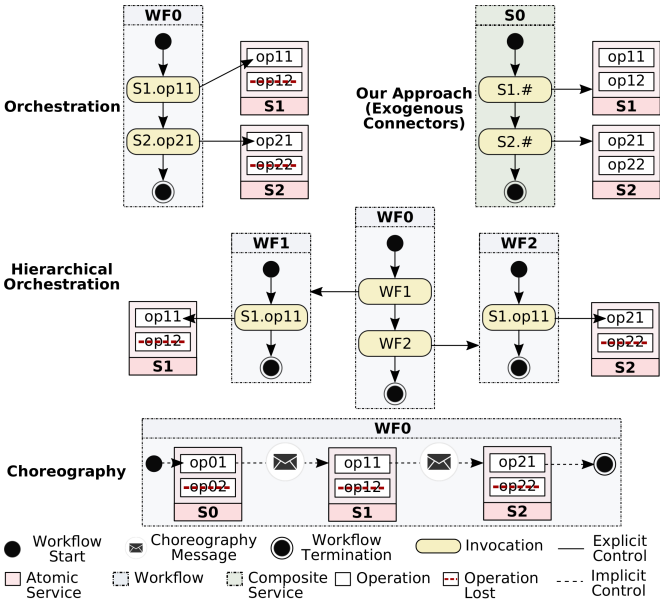


Fig. 1. Compositionality and control flow in SOA.

III. EXOGENOUS CONNECTORS FOR SERVICE COMPOSITION

We propose exogenous connectors for service composition, which define explicit control flow and enable hierarchical construction of SOA systems. Our notion of total compositionality is akin to mathematical function composition where two or more functions can be composed into a new function of the same type that can be further composed with other functions. Mathematical functions are composed algebraically and, hence, hierarchically. With this in mind, a service composition results in a new service that can be composed into even bigger services. Fig. 2 shows that, unlike (hierarchical) orchestration and choreography,[4] at every level of the hierarchy the result of composition is a service (of type $\mathbb{S}$). The operator $\circ$ denotes a service composition.

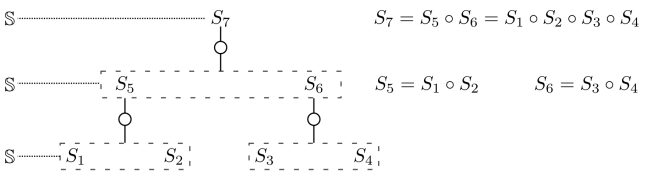


Fig. 2. Hierarchical service composition using exogenous connectors.

[4]See Appendix B.

A. Design of Exogenous Connectors

Our notion of exogenous connectors is borrowed from the X-MAN component model [11], [12], [13], but our approach is significantly different from X-MAN, especially in the semantics of distribution,[5] services and service composition. A detailed comparison with X-MAN is out of scope, but we will briefly discuss the main differences in Sect. V.

Exogenous connectors are architectural elements that mediate the interaction between services. They originate control and coordinate the execution of an SOA system; to this end, they encapsulate a network communication mechanism in general and control in particular. There are three kinds of connectors: (i) *invocation*, (ii) *composition* and (iii) *adaptation*.

An invocation connector is connected with a computation unit which encapsulates the implementation of some behaviour and is not allowed to call other computation units (see Fig. 3(a)). An invocation connector provides access to the operations implemented in the computation unit.

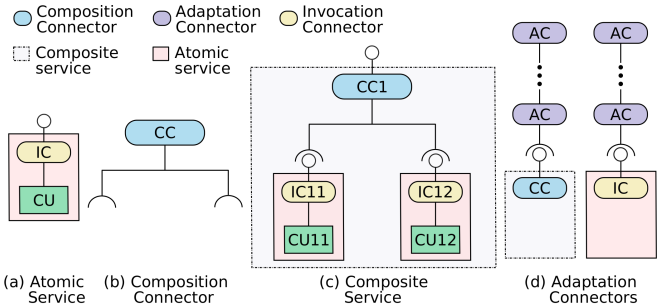


Fig. 3. Exogenous connectors and services.

A composition connector is a composition operator ($\circ$) that defines explicit control flow and coordinates the execution of $n > 1$ (atomic and/or composite) services (see Fig. 3(b)). Composition connectors can be defined for the usual control structures in SOA for sequencing, branching, and parallelism. The sequencer connector allows the composition of services $S_1, \ldots, S_n$ and executes them in sequential order. The selector connector allows the composition of services $S_1, \ldots, S_n$ and can choose the services out of them to be executed, according to a predefined condition. The parallel connector composes $S_1, \ldots, S_n$ services and executes all of them in parallel.

Fig. 3(d) shows that $n \geq 0$ adaptation connectors can be connected with either a composition connector or an invocation connector. Adaptation connectors can be defined for complementary control structures in SOA such as looping and guarding. They do not require the composition of services as they only operate, if a predefined condition is true, over an individual service. The control structure for looping defines a number of iterations, while a guard connector provides gating.

Our approach is then a Turing complete set for defining explicit control flow for sequencing, branching, and looping. Composition connectors can define (and encapsulate) workflows for the set of composed services. Composition connectors and adapters are able to receive, initiate and return control; whereas invocation connectors are only able to receive and return control.

Services only provide operations and do not call directly operations provided by other services. Fig. 3 shows that there are two kinds of services: (i) *atomic* and (ii) *composite*. An atomic service is formed by connecting an invocation connector with a computation unit (see Fig. 3(a)), whose interface has all the operations implemented in the computation unit.

A composite service consists of a set of (atomic and/or composite) services composed by a composition connector (see Fig. 3(c)). Its interface is constructed from the interfaces of the composed services; thereby, a composite has available all the operations of the composed services (see Fig. 4).

Services are decoupled from the hierarchical control flow structure provided by connectors. Fig. 4 shows that composite services are self-similar as exogenous connectors enable compositional and, therefore, hierarchical construction in a bottom-up fashion. The connectors of the top- and middle-levels are of variable arities and types since they can be connected to any number of connectors. At the bottom-level, there are unary invocation connectors which connect to single connectors. We use the master-slave pattern so higher-level connectors are the masters of the lower-level connectors they are connected to.

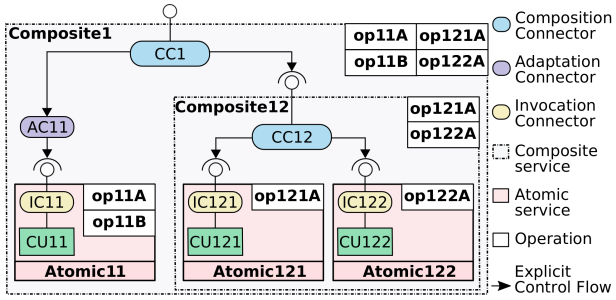


Fig. 4. Total compositionality and explicit control flow in our approach.

The precise choice of connectors, the number of levels of the hierarchy and the connection structure, depend on the relationship between the behaviour of the individual services and the behaviour that the system is intended to achieve. The control structure is always hierarchical, which means that there is always one connector at the top-level (the top-level composite can represent an SOA system *per se*). This connector initiates the control flow in the whole system. For instance, the connector $CC1$ initiates the generic SOA system presented in Fig. 4.

Fig. 5 shows a possible data flow for the service composition presented in Fig. 4, where we can see that data is represented by parameters and data flow is orthogonal to control flow. Input parameters are the required data by either an operation or a (composition or adaptation) connector, while output parame-

[5] X-MAN is not distributed.

ters are data resulting from an operation's computation.[6] Connectors read input parameter values and write output parameter values on data channels [11]. Composition and adaptation connectors read input parameters to achieve their purpose, e.g., a selector may define a condition $price < 2000$ that requires the input parameter $price$. An invocation connector reads input parameters and writes output parameters for the operations of the computation unit it is connected to.

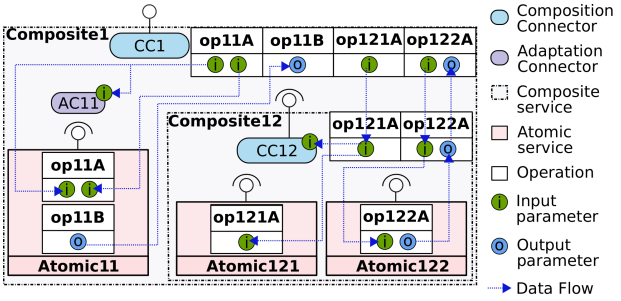


Fig. 5. Data flow in our approach.

A data channel connects two endpoints: an origin parameter $from$ with a destination parameter to. There is a set of data channels for each operation of a composite service; for instance, in the composite $Composite1$ in Fig. 5, the operation $op121A$ has only one data channel, whereas the operation $op11A$ has three data channels. Data channels are automatically created (on service composition) for each operation of a composite service; nevertheless, composite service operations can be customized so as to add and/or remove data channels. In Fig. 5, the data channels connected to the input parameters of connectors $CC12$ and $AC11$, respectively, were added manually.

Fig. 6 illustrates that both composite and atomic services can be potentially mapped onto different nodes over a network. In particular, Fig. 6 shows a possible mapping of the services and connectors presented in Fig. 4. Exogenous connectors reside in the same network address as the service they belong to. For instance, the atomic service *Atomic11*, its invocation connector *IC11*, and its adaptation connector *AC11* reside in *203.0.113.7*. Services are location- and workflow-agnostic as exogenous connectors encapsulate service location and define explicit control flow. The workflow of an SOA system is distributed among the involved exogenous connectors.

B. Implementation of Exogenous Connectors

We implemented the meta-model of our proposal in Java (see Fig. 7).[7] The purple section encompasses the classes for exogenous connectors, the green section includes the classes for network communication and the rest of the classes are concerned with services and data representation. Services and exogenous connectors were defined as a hierarchy of Java

[6]Connectors do not have output parameters because they do not perform any computation.

[7]https://gitlab.cs.man.ac.uk/mbaxrda2/ExogenousConnectors

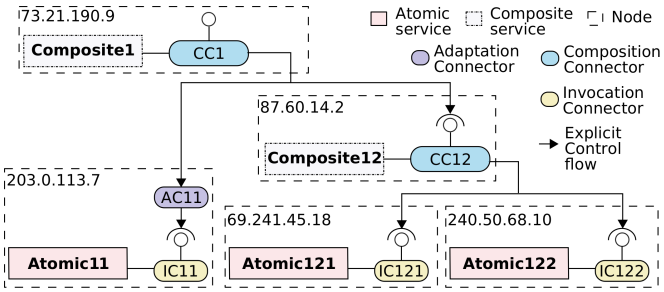


Fig. 6. Connector and service mappings over a network.

classes. The superclasses *ConnectorType* and *Service* allow the definition of any connector and any service, respectively, at any level of the hierarchy.

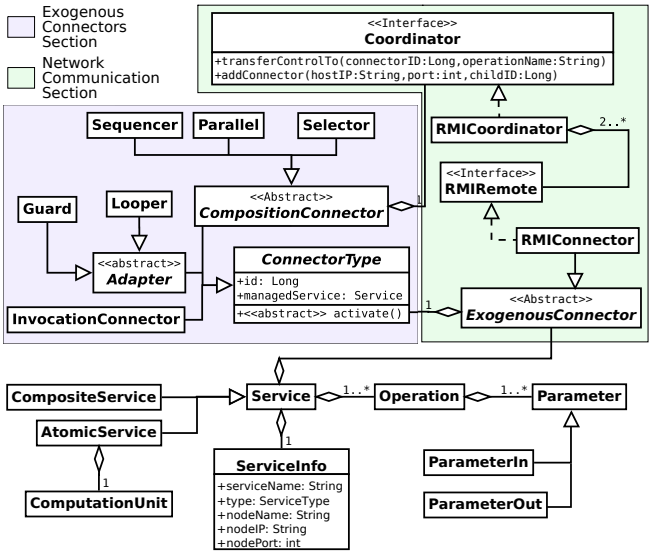


Fig. 7. Meta-model of our proposal.

The *ServiceInfo* class encapsulates the service name and the service kind as well as the details (i.e., name, IP address and listening port) of the node wherein the service is deployed. A service provides at least one operation with at least one parameter. Parameters and data channels have unique IDs within a network.

The *ExogenousConnector* class encapsulates a network communication mechanism for the interaction between exogenous connectors via the network. Although we particularly use Remote Method Invocation (RMI), it is possible to replace it with any other mechanism such as HTTP/REST. Thus, exogenous connectors use RMI to coordinate an SOA system execution (by passing only control) via the network. As we rely on hierarchical composition, a composition connector contains an *RMICoordinator* instance which provides the *transferControl*() method to pass control to the remote connectors of the composed services. Exogenous connectors have unique IDs within a network.

Our exogenous connectors are synchronous so they are always listening for remote invocations from higher-level connectors. The *ConnectorType* class has the abstract method *activate*() which is invoked remotely by other connectors. This method is implemented according to the intended control structure of the exogenous connector involved.

A selector connector associates each lower-level connector with a condition by which these connectors are invoked. An adaptation connector associates a single lower-level connector with a single condition. Sequencer connectors remotely invoke, in a given order, a list of lower-level connectors. A parallel connector creates a Thread pool of n threads, where n is the number of composed services; hence, the parallel execution of services is performed by Java threads. An invocation connector uses the *invoke*() method provided by Java reflection to execute an operation in the connected computation unit.

Total compositionality does not require any glue that has to be constructed manually; therefore, invocation connectors dynamically invoke an operation (provided by the atomic service they belong to) by reading an invocation map from a data space. An invocation map associates a service ID (i.e., an entry key) with the ID of the operation (i.e., an entry value) to be invoked in that service. During the deployment of a composite service CS, Algorithm 1 generates an invocation map M_i for each operation Op_i provided by CS. For each data channel dc_i of the operation Op_i, Algorithm 2 analyzes the respective endpoints (i.e., the origin $from$ and the destination to). Only data channels connected to service operation parameters are analyzed[8] and we particularly assume that the given data channels are valid. Invocation maps are written in the data space DS with the operation ID as the key.

Algorithm 1 Algorithm for the generation of invocation maps

Input: The data space DS and the composite service CS being deployed
  ▷ Op_i:An operation provided by the composite CS
for all $OP_i \in CS$ **do**
    ▷ M_i:Invocation map for the operation Op_i
  $M_i \leftarrow newInvocationMap()$
    ▷ dc_i:A data channel for the operation Op_i
  for all $dc_i \in OP_i$ **do**
    $analyzeEndpoint(DS, CS, M_i, dc_i.from)$
    if $dc.to.notInConnector()$ **then**
      $analyzeEndpoint(DS, CS, M_i, dc_i.to)$
    end if
  end for
  $DS.write(Op_i.id, M_i)$
end for

When a data channel is connected to a parameter of an operation provided by a composite service (different to the one being deployed), the invocation map $M_{endpoint}$ (previously generated by that composite) is retrieved from the data space DS and combined with the invocation map M_i.[9] Otherwise, if the data channel is connected to an atomic service's operation and the invocation map M_i does not have an entry for that service, the association between the atomic service ID and the operation ID is created in the invocation map M_i.

Algorithm 2 Algorithm for the analysis of a data channel endpoint

Input: The data space DS, the composite service CS being deployed, the invocation map M_i being generated and the data channel $endpoint$ to analyze
if $endpoint.service$ is a composite **then**
  if $endpoint.service$ is not the composite CS **then**
    $M_{endpoint} = DS.read(endpoint.operationId)$
    for all $key, value \in M_{endpoint}$ **do**
      $M_i.putIfAbsent(key, value)$
    end for
  end if
else
  $M_i.putIfAbsent(endpoint.serviceId,$
    $endpoint.operationId)$
end if

We also developed an algorithm for reading and writing data efficiently. However, we do not present this algorithm due to space constraints.

C. Platform Support

We implemented a platform in Java for the development of SOA systems based on exogenous connectors. A central service repository was implemented to store and retrieve services. Data is managed by a shared data space: MozartSpaces 2.3.[10]

Fig. 8. Platform support.

System instances sit above the *Platform API* which provides the constructs for designing and deploying services as well as executing systems. *Platform Core* provides the functionality for repository, data and deployment management. *Network Mgmt* contains the communication mechanisms to perform actions over the network such as passing control between connectors. Our platform requires every node to have support for *Java Runtime Environment (JRE) 1.8*.

IV. Case Study

Our case study (see Fig. 9) is based on the popular MusicCorp [18]. It is focused on the creation of customers which get a new record in a loyalty points bank and receive a welcome

[8]Data channels connected to connector parameters are not analyzed because connectors do not provide operations.

[9]The invocation maps for the operations of sub-composite services are generated in advance as composite services are deployed in a bottom-up way.

[10]http://www.mozartspaces.org

pack/email. We do not show data flow as services are composed by composition connectors which rely on control flow. The source code was generated using the platform API and it is available at https://gitlab.cs.man.ac.uk/mbaxrda2/MusicCorp.

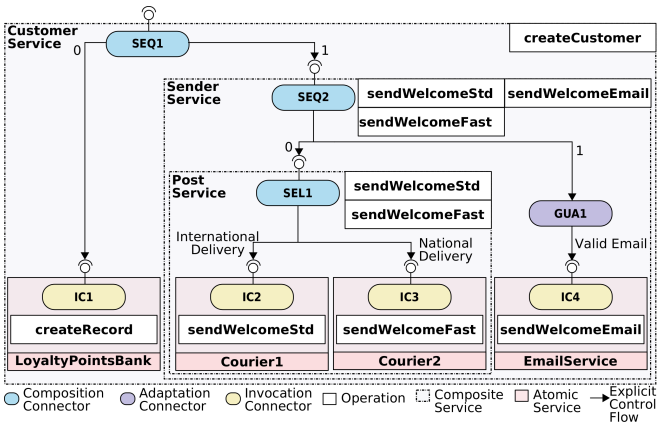


Fig. 9. Compositionality and explicit control flow in our case study.

The atomic services *LoyaltyPointsBank*, *Courier1*, *Courier2* and *EmailService* offer primitive operations to achieve the intended behavior of our case study. *LoyaltyPointsBank* has the operation *createRecord* to store customer details in a database. *Courier1* and *Courier2* provide the operations to send a welcome pack by standard and fast delivery, respectively. *EmailService* exposes the *sendWelcomeEmail* operation to send a welcome email to new customers.

Our case study is constructed in a hierarchical bottom-up fashion. First, *Courier1* and *Courier2* are composed into *PostService* by the selector connector *SEL1*. Then, the composite *SenderService* uses the sequencer connector *SEQ2* to compose *PostService* and *EmailService*. Finally, *LoyaltyPointsBank* and *SenderService* are composed into *CustomerService* by the sequencer connector *SEQ1*.

At the bottom-level, we have the invocation connectors *IC1*, *IC2*, *IC3*, and *IC4*. In the next level, we have the adapter *GUA1*. Then, we have the selector *SEL1*. Next, we have the sequencer *SEQ2*. Finally, at the top-level, we have the sequencer *SEQ1*.

The execution of our case study is control-driven. The top-level connector *SEQ1* starts the execution by passing control to the invocation connector *IC1* and the sequencer *SEQ2*, in that order. Next, *SEQ2* invokes the selector *SEL1* which activates either the invocation connector *IC2* or the invocation connector *IC3*, depending on the customer address. Then, *SEQ2* invokes the adapter *GUA1* (which denies the invocation of *EmailService* if the customer email is invalid). Finally, *SEQ2* returns the control to the top-level connector *SEQ1* and the execution terminates.

In general, *SEQ1* defines a sequential invocation of *LoyaltyPointsBank* and *SenderService*. Similarly, *SEQ2* defines a sequential execution of *PostService* and *EmailService*. *SEL1* explicitly defines a condition for invoking either *Courier1* or *Courier2*. *GUA1* defines gating for *EmailService*.

We implemented a client to remotely execute the operation *CreateCustomer* in *CustomerService*. Our case study was tested in *localhost* with each service running in a separate process (to simulate different nodes in the Local Area Network). We mapped a service per node.

Fig. 10 displays a screenshot of the standard output for the composite *SenderService*, resulting from the execution of our case study. A glance at the bottom of Fig. 10, reveals the explicit control flow defined by the sequencer *SEQ2* (with ID *4684084166367832649*): *SEQ2* passes control to the selector *SEL1* (with ID *-5878785820492134700*) of *PostService* and, then, to the adapter *GUA1* (with ID *84636168467804098*) of *EmailService*.

To achieve total compositionality, the composition of two or more services must yield another service that (1) preserves all the operations provided by the composed services and (2) can be composed into even bigger services. Our composite services inherit all the operations from their respective composed services. Thus, primitive operations are initially defined in atomic services and inherited on composition. For instance, as shown in Fig. 9 and 10, *PostService* and *EmailService* are composed into *SenderService* by the sequencer *SEQ2* (with ID *4684084166367832649*); thereby, the composite *SenderService* has available the operations *sendWelcomeStd*, *sendWelcomeFast*, and *sendWelcomeEmail*.

V. Evaluation and Discussion

Although our notion of exogenous connectors is borrowed from the X-MAN component model, there are important differences. X-MAN is a general-purpose and a single-process component model, whereas our approach is particularly focused on SOA systems. For this reason, unlike X-MAN, our approach is distributed (i.e., multi-process) so services are mapped onto different network addresses, and the control flow is distributed over a network. Moreover, in contrast to X-MAN, we followed SOA principles for the definition of services and service composition. We also changed the semantics of X-MAN so as to support (1) the parallel invocation of services and (2) the execution of multiple services that satisfy a particular condition in the selector connector. Finally, we developed an algorithm to dynamically invoke primitive operations in atomic services, so the manual mapping of operations (during the design phase) is not required anymore.

Total compositionality entails a strictly hierarchical way of constructing SOA systems by composing services. In our approach, atomic services form a flat layer and the entire control structure (of composition and adaptation connectors) sits on top of this. This hierarchical composition structure is split up among the exogenous connectors (which are distributed over a network). A hierarchical structure enables location transparency which is crucial for scalability since service locations may dynamically change. For instance, if *PostService* changes its location, only the connector of the composite *SenderService* is affected without requiring updates to other connectors or other services.

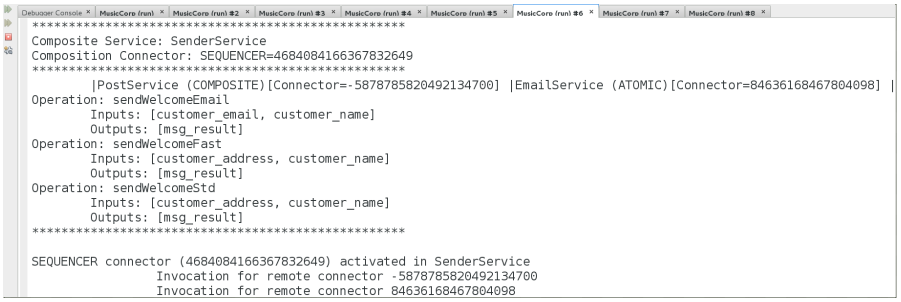


Fig. 10. Standard output for the composite *PostService*.

Having available all the operations in a composite service implies that any operation can be invoked in any composed service. In fact, adding new operations does not require any change in the workflow defined by our connectors. Conversely, (hierarchical) orchestration and choreography require n workflows for n different operations, leading to combinatorial explosion as the number of operations increases.

For instance, adding the operation $sendProduct$ in the composite *CustomerService* does not require changing the workflow defined by such a composite. Conversely, (hierarchical) orchestration and choreography will require two different workflows: one for the invocation of the operation $createCustomer$ and another one for the invocation of the operation $sendProduct$.

Of course, our composite services can be customized to add new operations, remove operations inherited on composition, or both. Fig. 9 shows that we customized the top-level composite *CustomerService* to expose the operation *createCustomer* (to the final users) rather than the operations inherited on composition.

Total compositionality results in a service type at every level of the hierarchy, leading to service reuse. For instance, the composite *CustomerService* can be reused in multiple e-commerce systems. In orchestration, it is possible to reuse a workflow whereas in our approach it is possible to reuse a service containing multiple workflows.

Invocation connectors use the Algorithms presented in Sect. III-B so as to dynamically find the operation to invoke. Therefore, exogenous connectors only pass control and explicitly define the order in which services are executed, rather than define the order in which operations are invoked. Explicit control is important for scalability since it enables monitoring, tracking and visualization of service interaction. It therefore leverages the maintenance and the evolution of SOA systems.

VI. Conclusion and Future Work

Total compositionality and explicit control flow are crucial for the scalability of SOA systems. In this paper, we presented exogenous connectors for service composition. Like orchestration, exogenous connectors define explicit control flow, but unlike (hierarchical) orchestration and choreography, they enable total compositionality. We were not able to get a real-world case study consisting of many services to perform quantitative evaluation on scalability, so we evaluated qualitatively our proposal from the popular MusicCorp. We plan to perform quantitative evaluation in the future.

Centralized execution of composite services is not desirable for scalability [21]. For this reason, currently, we are investigating novel ways of achieving decentralized service composition. Additionally, as our approach enables composition automation, we are working on a novel mechanism to dynamically reconfigure services in the presence of changes in the environment. We strongly believe that exogenous connectors will play an important role in the development of large-scale SOA systems. Indeed, we are currently in discussion with an industrial partner on this matter.

Acknowledgment

The first author would like to thank CONACyT for the financial support to carry out his research.

References

[1] A. Barros, M. Dumas, and P. Oaks, "Standards for Web Service Choreography and Orchestration: Status and Perspectives," in *Business Process Management Workshops*. Springer, Berlin, Heidelberg, Sep. 2005, pp. 61–74.

[2] E. Ben Hadj Yahia, I. Gonzalez-Herrera, A. Bayle, Y.-D. Bromberg, and L. Réveillère, "Towards Scalable Service Composition," in *Proceedings of the Industrial Track of the 17th International Middleware Conference*, ser. Middleware Industry '16. ACM, 2016, pp. 3:1–3:6.

[3] C. Carneiro and T. Schmelmer, "Microservices: The What and the Why," in *Microservices From Day One*. Berkeley, CA: Apress, 2016, pp. 3–18.

[4] G. Chafle, S. Chandra, and V. Mann, "Decentralized Orchestration of Composite Web Services," in *Proceedings of the 13th International WWW Conference*, 2004, pp. 134–143.

[5] S. Daya, N. V. Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain, and R. Vennam, *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*, Sep. 2016.

[6] P. D. Fensel, D. F. M. Facca, D. E. Simperl, and I. Toma, "The Web Service Execution Environment," in *Semantic Web Services*. Springer Berlin Heidelberg, 2011, pp. 163–216.

[7] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014.

[8] W. Jaradat, A. Dearle, and A. Barker, "Towards an autonomous decentralized orchestration system," *Concurrency Computat.: Pract. Exper.*, vol. 28, no. 11, pp. 3164–3179, Aug. 2016.

[9] S.-S. T. Q. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, and H. Afsarmanesh, "Orchestrating web services using Reo: From circuits and behaviors to automatically generated code," *Service Oriented Computing and Applications*, vol. 8, no. 4, pp. 277–297, Dec. 2014.

[10] M. Jung and J. Simon, "Microservices on AWS," https://aws-de-media.s3.amazonaws.com/images/AWS_Summit_Berlin_2016/sessions/pushing_the_boundaries_1300_microservices_on_aws.pdf, 2016.
[11] K. K. Lau and C. M. Tran, "X-MAN: An MDE Tool for Component-Based System Development," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, 2012, pp. 158–165.
[12] K.-K. Lau, L. Safie, P. Stepan, and C. Tran, "A Component Model That is Both Control-driven and Data-driven," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*. ACM, 2011, pp. 41–50.
[13] K.-K. Lau, P. Velasco Elizondo, and Z. Wang, "Exogenous Connectors for Software Components," in *Proceedings of the 8th International Conference on Component-Based Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 90–106.
[14] P. Leitner, W. Hummer, and S. Dustdar, "Cost-Based Optimization of Service Compositions," *IEEE Transactions on Services Computing*, vol. 6, no. 2, pp. 239–251, Apr. 2013.
[15] A. L. Lemos, F. Daniel, and B. Benatallah, "Web Service Composition: A Survey of Techniques and Tools," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–41, 2016.
[16] Netflix, "Conductor," https://netflix.github.io/conductor/, 2016.
[17] P. G. Neumann, "Principled Assuredly Trustworthy Composable Architectures," Tech. Rep., 2004.
[18] S. Newman, *Building Microservices*, 1st ed. Beijing Sebastopol, CA: O'Reilly Media, Feb. 2015.
[19] C. Peltz, "Web Services Orchestration and Choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct. 2003.
[20] S. Ross-Talbot and T. Fletcher, "Web Services Choreography Description Language: Primer," https://www.w3.org/TR/ws-cdl-10-primer/, 2006.
[21] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, Oct. 2014.
[22] K. A. Suji and S. Sujatha, "A Comprehensive Survey of Web Service Choreography, Orchestration And Workflow Building," *International Journal of Computer Applications*, vol. 88, no. 13, pp. 18–23, Feb. 2014.
[23] N. Taušan, J. Markkula, P. Kuvaja, and M. Oivo, "Choreography in the embedded systems domain: A systematic literature review," *Information and Software Technology*, Jun. 2017.
[24] N. Temglit, A. Chibani, K. Djouani, and M. A. Nacer, "A Distributed Agent-Based Approach for Optimal QoS Selection in Web of Object Choreography," *IEEE Systems Journal*, no. 99, pp. 1–12, 2017.
[25] W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. ter Hofstede, "Design and Implementation of the YAWL System," in *Advanced Information Systems Engineering*. Springer, Berlin, Heidelberg, Jun. 2004, pp. 142–159.
[26] O. Zimmermann, "Microservices tenets," *Comput Sci Res Dev*, vol. 32, no. 3, pp. 301–310, 2017.

APPENDIX A

A service exposes a set of operations through a well-defined WSDL interface. A service $S \in \mathbb{S}$, where $\mathbb{S}$ is the type of services, is a set of operations defined as follows:

$$S = \{op_i \mid i \in \mathbb{N}\} \tag{1}$$

An orchestration or a choreography can be defined as a function $ORCH$ with the following type:

$$ORCH : \mathbb{OP} \times \mathbb{OP} \times \cdots \times \mathbb{OP} \rightarrow \mathbb{WF} \tag{2}$$

where $\mathbb{OP}$ is the type of operations in the invoked services and $\mathbb{WF}$ is the type of workflows for invoking a set of such operations.

A workflow is a sequence of invocations of service operations whose permutation is defined by the designer of the orchestration. A workflow is then defined as follows:

$$wf = \langle inv(op_i) \mid i \in \mathbb{N} \rangle \tag{3}$$

where $inv(op_i)$ is an invocation to the operation op_i.

A conversion from a workflow type $\mathbb{WF}$ into a service type $\mathbb{S}$ (with one operation for invoking the workflow) can be defined as a function $CONV$ with the following type:

$$CONV : \mathbb{WF} \rightarrow \mathbb{S} \text{ where } |S| = 1 \tag{4}$$

A hierarchical orchestration is defined by concatenating workflow sequences. It can therefore be defined as a function $HORC$ as follows:

$$HORC(wf_1, wf_2, \ldots, wf_n) = wf_1 \frown wf_2 \frown \cdots \frown wf_n \tag{5}$$

Our notion of total compositionality is defined as a function $COMP$ with the following type:

$$COMP : \mathbb{S} \times \mathbb{S} \times \cdots \times \mathbb{S} \rightarrow \mathbb{S} \tag{6}$$

APPENDIX B

Although an orchestration can be hierarchical, it is not totally compositional since there is not a service type at every level of the hierarchy as in our approach (see Fig. 2 in Sec. III). Consider the formal definitions presented in Appendix A and four services: $S_1 = \{op_{11}, op_{12}\}$, $S_2 = \{op_{21}, op_{22}\}$, $S_3 = \{op_{31}, op_{32}\}$ and $S_4 = \{op_{41}, op_{42}\}$. Fig. 11 illustrates a hierarchical orchestration wf_{1234} constructed by the concatenation of the sub-workflows wf_{12} and wf_{34}:

$$HORCH(wf_{12}, wf_{34}) = wf_{12} \frown wf_{34} = wf_{1234} \tag{7}$$

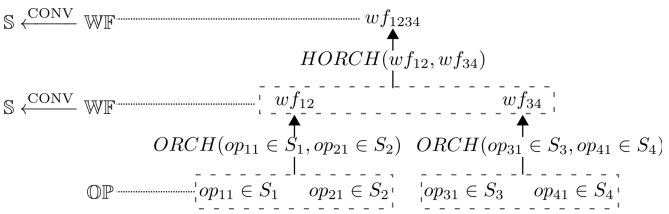


Fig. 11. Hierarchical orchestration.

The workflows wf_{12} and wf_{34} are sequences of type $\mathbb{WF}$ resulting from the functions:

$$ORCH(op_{11} \in S_1, op_{21} \in S_2) = \langle inv(op_{11} \in S_1), inv(op_{21} \in S_2) \rangle = wf_{12} \tag{8}$$

$$ORCH(op_{31} \in S_3, op_{41} \in S_4) = \langle inv(op_{31} \in S_3), inv(op_{41} \in S_4) \rangle = wf_{34} \tag{9}$$

Fig. 11 shows that in hierarchical orchestration, even if workflows can be converted into services (providing one operation for invoking the workflow) by applying the function $CONV$, there is not a service type at every level of the hierarchy.

4.2 DX-MAN: A Platform for Total Compositionality in Service-Oriented Architectures

Damian Arellanes and Kung-Kiu Lau

In proceedings of the **7th International Symposium on Cloud and Service Computing (SC2 2017)**.

Published by IEEE,
pages 283-286,
ISBN 978-1-5386-5862-8,
2017.

Impact Indicators:
• Impact factor (2018): 0.98,
• Fourth most cited article amongst 47 publications in SC2 2017 (as of 2019).

The final authenticated version [AL17a] is available online at `https://doi.org/10.1109/SC2.2017.55`.

Summary: This paper presents a platform that implements the semantics of the DX-MAN model. The platform is useful for the validation of functional scalability requirements in DX-MAN as the thesis progresses, which later evolved into an IoT platform that is available online at `http://github.com/damianarellanes/dxman`.

Comments on authorship: I proposed the main idea of the paper, designed and implemented the platform, investigated related work, provided and edited all graphics, participated in the entire writing process and addressed the reviewer's comments. My supervisor, Kung-Kiu Lau, also contributed to the idea, proofread the paper and approved the platform. He also guided the whole research process.

Key contributions: Contribution 4 (see Section 1.5).

D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures

Damian Arellanes and Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

***Abstract*—Current software platforms for service composition are based on orchestration, choreography or hierarchical orchestration. However, such approaches for service composition only support partial compositionality; thereby, increasing the complexity of SOA development. In this paper, we propose DX-MAN, a platform that supports total compositionality. We describe the main concepts of DX-MAN with the help of a case study based on the popular MusicCorp.**

***Index Terms*—service composition, platform, orchestration, choreography, scalability, microservices, exogenous connectors**

I. Introduction

Service-Oriented Architectures (SOA) are popular in the software industry because they enable high modularity. Many software platforms for service composition have been proposed. However, such platforms only provide support for partial compositionality, since they are based on orchestration [1], choreography [2], [3] or hierarchical orchestration [4], [5], [6]. Partial compositionality [7] requires software developers to design individual workflows for the invocation of service operations, leading to combinatorial explosion and, therefore, increasing the complexity of SOA system development.

Total compositionality [7] means that two or more services can be composed into a new (composite) service of the same type, that preserves all the operations provided by the composed services. It implies a hierarchical composition structure but not the other way round. Total compositionality is crucial for the scalability of SOA systems since it only requires the design of one workflow for the invocation of any operation in any composed service.

In this paper, we present DX-MAN, a platform for total compositionality based on the hierarchical model we presented in [7], where services and exogenous connectors are first-class entities. Exogenous connectors are architectural elements that mediate the interaction between services. They originate control and coordinate the execution of an SOA system by passing only control; to this end, they encapsulate a network communication mechanism in general and control in particular.

The rest of the paper is organized as follows. Section II presents an overview of the proposed platform. Section III discusses the strengths of the proposed platform and presents the concluding remarks.

II. Platform Overview

DX-MAN is a platform that delivers the necessary programming abstractions and the runtime environment to design, deploy and execute SOA systems. DX-MAN relies on the notion of service template and service instance. A service template provides the skeleton of a service design, whereas a service instance is the result of a service template deployment.

In this section, we describe the main concepts of DX-MAN with the help of a case study based on the popular MusicCorp [8]. The objective of this case study is the creation of new customers which get a record in a loyalty points bank and receive a welcome pack/email. Fig. 1 shows the service composition and the data flow of our case study. For further details about the model and the case study, please refer to [7].

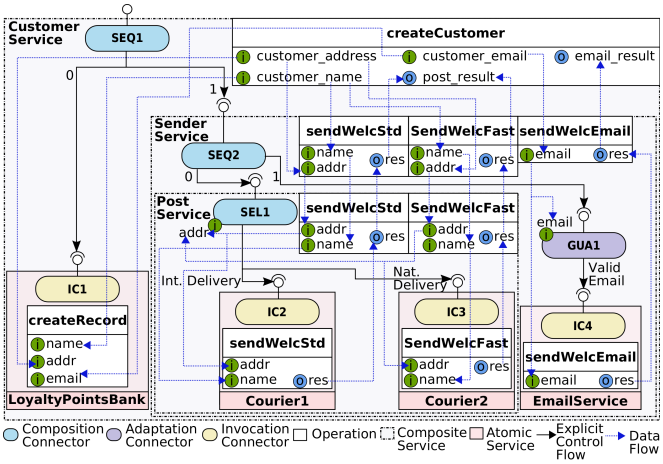


Fig. 1. Service composition and data flow of our case study.

A. Platform Architecture

We implemented DX-MAN in Java due to the popularity of this programming language. A central service repository was also implemented to publish and retrieve service templates so as to support reuse. Data is managed by MozartSpaces 2.3 [9],[1] a popular data space that offers extensive support. Figure 2 illustrates the architecture of DX-MAN.

[1]The central service repository and the data space can reside at any network address.

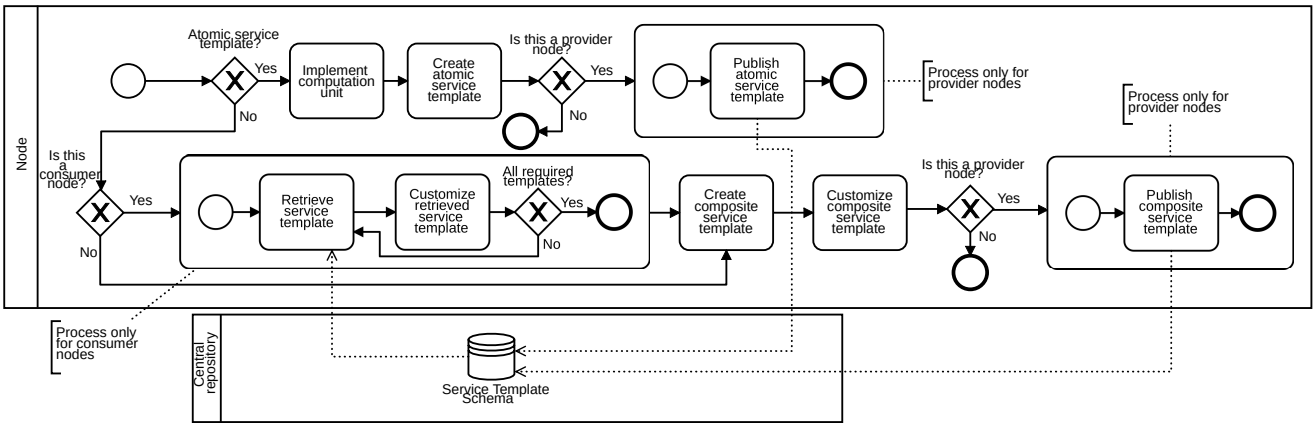


Fig. 3. Process for service design and reuse in DX-MAN.

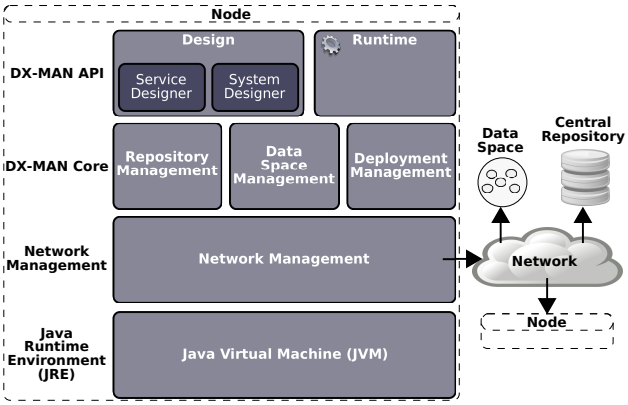


Fig. 2. DX-MAN platform.

DX-MAN API hides the complexity of the platform and offers the constructs to design and deploy services, and execute SOA systems. *DX-MAN Core* is divided into three modules: (a) *Repository Management* provides the functionality to publish and retrieve services from the central service repository; (b) *Data Space Management* provides the functionality to perform operations in the data space such as reading and writing; and (c) *Deployment Management* offers the functionality to deploy services. *Network Management* contains the communication mechanisms to perform operations on the network such as passing control between connectors and connecting to the central repository.

A node is a logical entity within a network that uses DX-MAN. It can host any number of service instances in its Java Virtual Machine (JVM). DX-MAN requires every node to have support for *Java Runtime Environment (JRE) 1.8*. A node can play the role of provider, consumer, or both. On the one hand, a provider node publishes service templates in the central repository for further reuse. On the other hand, a consumer node retrieves templates from the central repository, in order to design composite service templates.

Provider nodes are required to set a deployment directive in service templates. A *downloadable* directive indicates that the service template must be deployed in the Java Virtual Machine (JVM) of consumer nodes. A *non-downloadable* directive states that the service template is always deployed in the JVM of the respective service provider.

A complete life cycle for SOA development should consist of two life cycles: a *service* life cycle and a *system* life cycle. The service life cycle comprises two phases: (1) design and (2) deployment. During the phase (1), a node designs service templates. For the phase (2), deployment directives drive the deployment of service templates in the JVM of the respective nodes.

The system life-cycle consists of three phases: (1) design, (2) deployment and (3) execution. During the phase (1), a node designs a system template (which has the form of a composite service template). System templates are deployed in the phase (2) by using a bottom-up approach: atomic services are deployed first and the top-level composite is deployed at the end. Finally, systems are executed in the phase (3).

Figure 3 shows a BPMN diagram that depicts the overall process for service design and reuse in DX-MAN. Designing an atomic service template comprises the following steps: (1) implementation of the computation unit, (2) creation of the atomic service template, and (3) publication of the atomic service template in the central repository. The step (3) is carried out only if the node is a provider node.

Service composition requires (1) the retrieval of service templates from the central repository for the composed services; (2) the customization of the retrieved service templates; (3) the creation of the composite service template; (4) the customization of operations and data flow for the composite service; and (5) the publication of the composite service template in the central repository. It is important to mention that composite service templates can be designed without reusing templates from the central repository. The step (1) is carried out only if the node is a consumer node and the step (5) is performed only if the node is a provider node. Steps (2) and

(4) are optional.

Next, we describe how DX-MAN maps definitions of our model to Java language primitives. In particular, we follow a programmer's point of view to show how the case study is implemented using DX-MAN API constructs.

B. Atomic Services

An atomic service is formed by connecting an invocation connector with a computation unit. A computation unit encapsulates the implementation of some behaviour and is not allowed to call other computation units. An invocation connector provides access to the operations implemented in the computation unit. A computation unit has the form of a Java class (Fig. 4). Computation unit operations are defined as class methods, annotated with *@Operation*. Operation parameters must be annotated with *@ParameterInfo*, and they must specify a property (of String type) for the parameter *name* and a property (of Class type) for the parameter *type*. The *DXManAtomicParameterIn* class is a wrapper for an input parameter, while the *DXManAtomicParameterOut* class is a wrapper for an output parameter. *DXManAtomicParameterIn* and *DXManAtomicParameterOut* provide methods to get and set data values, respectively. A computation unit is unaware of how data is handled internally by DX-MAN.

```
1 public class EmailServiceCU {
2     ...
3     @Operation
4     public void sendWelcEmail(
5         @ParameterInfo(name="email", type=String.class)
              ↪ DXManAtomicParameterIn customerEmail,
6         @ParameterInfo(name="res", type=String.class)
              ↪ DXManAtomicParameterOut msgResult) {
7         ...
8     }
9 }
```

Fig. 4. Example of a computation unit definition.

The constructor of an atomic service template requires the name of the service, the class of the computation unit and the deployment directive. When an atomic service template is created, atomic service operations are automatically extracted from the methods annotated in the computation unit; then, the invocation connector is automatically created and connected to the respective computation unit.

Provider nodes publish atomic service templates in the central repository, using the *publish*(ServiceTemplate) method of the *ServiceDesigner* class. For instance, the template for *EmailService* could be created and published with a *non-downloadable* directive as follows:

```
serviceDesigner.publish(serviceDesigner.createAtomicServiceTemplate(
      ↪ "EmailService", EmailServiceCU.class, NON_DOWNLOADABLE));
```

C. Composite Services

A composite service consists of a set of (atomic and/or composite) services composed by a composition connector. A composition connector defines explicit control flow and coordinates the execution of $n > 1$ (atomic and/or composite) services. Thus, services do not have any code for invoking other services. Composition connectors can be defined for the usual control structures in SOA for sequencing, branching, and parallelism. A parallel connector executes all the composed services in parallel, whose constructor only requires the templates for the composed services.

A sequencer connector executes composed services in sequential order. Its constructor receives the set of composed service templates, whose argument order matches the execution order.

A selector connector uses predefined conditions to choose the composed services to be executed. Its constructor receives a set of instances of the *ConditionMapping* class which associates a condition with a service template. Conditions are specified in the *matches*(*ConnectorDataSpace*) method of a Java class implementing the *ConnectorCondition* interface (Fig. 5). The *ConnectorDataSpace* class provides methods to match the value of a connector's input parameter with any value specified by the designer. For instance, the *matchesRegex*() method requires two arguments: the name of the connector's input and the regular expression to match with. Designers do not know how data is handled internally by connectors.

```
1 public class ConditionEmailGuard implements ConnectorCondition {
2     @Override
3     public boolean matches(ConnectorDataSpace cds) {
4         return cds.matchesRegex("email", getEmailPattern());
5     }
6     ...
7 }
```

Fig. 5. Example of a connector's condition definition.

Adaptation connectors provide complementary control structures in SOA such as looping and guarding. They do not compose services as they only operate, if a predefined condition is true, over an individual service. Any number of adaptation connectors can be connected to any composed service. For instance, our case study requires a guard adapter to deny the invocation of *EmailService*, if the customer email is invalid. Fig. 5 shows the definition of the condition for this adapter.

Figure 6 shows an example of the design of a composite service template. The *retrieveFromRemoteRepository*(*int*) method, provided by the *ServiceDesigner* class, is used by consumer nodes to retrieve service templates from the central repository (lines 1-2). This method only requires the id of the service template to be retrieved. Retrieved service templates can be customized, e.g., by changing the service name (line 3), selecting the operations to be used or both.

The constructor of a composite service template requires the service name, the template for the composition connector, the deployment directive, and the set of composed services (line 9). When a composite service template is created, a composite service interface is automatically constructed from the interfaces of the composed services. Hence, a composite has available all the operations of the composed services.

We use data channels to define data flow which is orthogonal to control flow. A data channel connects two endpoints: an

```
1  CompositeServiceTemplate postService = (CompositeServiceTemplate)
       ↪ serviceDesigner.retrieveFromRemoteRepository(4);
2  AtomicServiceTemplate emailService = (AtomicServiceTemplate)
       ↪ serviceDesigner.retrieveFromRemoteRepository(3);
3  emailService.getInfo().setServiceName("EmailService");
4
5  GuardAdapterTemplate gua1 = new GuardAdapterTemplate(ConditionEmailGuard.
       ↪ class);
6  gua1.addInput(new DXManParameterIn("email", String.class, 0));
7  emailService.addAdapter(0, gua1);
8
9  CompositeServiceTemplate senderService = serviceDesigner.
       ↪ createCompositeServiceTemplate("SenderService", new
       ↪ SequencerConnectorTemplate(postServiceTemplate,
       ↪ emailServiceTemplate), DOWNLOADABLE, postServiceTemplate,
       ↪ emailServiceTemplate);
10
11 serviceDesigner.createDataChannel(senderService, sendWelcomeEmail,
       ↪ senderServiceTemplate, "sendWelcomeEmail", "email",
       ↪ emailServiceTemplate, gua1, "email");
12
13 serviceDesigner.publish(senderService);
```

Fig. 6. Example of a design process for a composite service template.

origin parameter $from$ with a destination parameter to. Data channels are automatically created when a composite service template is created. After composition, composite service operations can be customized to add new data channels or remove the existing ones (line 11).

Like atomic service templates, composite service templates are published in the central repository using the *publish*(*ServiceTemplate*) method of the *ServiceDesigner* class (line 13).

D. System Design, Deployment and Execution

Our approach for service composition enables hierarchical construction of SOA systems. Therefore, there is a service at every level of the hierarchy and there is always one connector at the top-level that initiates the execution. The top-level composite represents a system *per se*.

The *SystemDesigner* class provides the means to create and deploy system templates. A system template does not require a deployment directive since it is always deployed in the JVM of the provider node. The deployment of a system template results in a system instance available to final users. In our case study, *CustomerService* is created and deployed as follows:

```
systemDesigner.deploySystem(systemDesigner.createSystemTemplate(
    ↪ "CustomerService", new SequencerConnectorTemplate(
    ↪ loyaltyPointsBankTemplate, senderServiceTemplate),
    ↪ loyaltyPointsBankTemplate, senderServiceTemplate));
```

The *RemoteSystem* class allows final users to interact with the system, e.g., by invoking operations or reading output values.

III. Discussion and Concluding Remarks

In this paper, we presented a platform that supports total compositionality in SOA. Current platforms for service composition are only focused on partial composition, where the designer needs to create multiple workflows for the invocation of service operations, leading to combinatorial explosion. In contrast, in DX-MAN, designers only need to design one workflow for the invocation of services (not for the invocation of individual operations). We described the main concepts of DX-MAN with the help of a case study based on the popular MusicCorp.

DX-MAN separates data, control and computation, in order to encourage the maintenance, reuse and evolution of SOA. In particular, such a separation of concerns makes it easy to reason about data flow, control flow and behaviour separately.

DX-MAN is based on exogenous connectors which coordinate services from outside, so services do not have code to interact one another directly. Thus, DX-MAN allows the development of encapsulated services. This helps to avoid *rigidity* so if the designer changes a service, other services are not changed.

Moreover, services do not know the location of other services. This is important for SOA as service instances can be anywhere and their locations can even dynamically change.

An important advantage of DX-MAN is its hierarchical nature to construct systems, resulting in well-structured code for the final system, which is easy to understand and therefore maintain. Services can be as simple as possible and their size can be small (e.g., a microservice) or big (e.g., a composite service composing plenty of services). A bottom-up approach should make services more tractable and, hence, practicable to reason about services and their composition separately.

Model-Driven Engineering (MDE) is gaining popularity in software system development. For this reason, we are currently working on MDE techniques for DX-MAN. Additionally, we would like to migrate our platform to the Cloud and evaluate it in a real-world application. In fact, we are currently in discussion with an industrial partner on this matter.

Acknowledgment

The first author would like to thank CONACyT for the financial support to carry out his research.

References

[1] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, Oct. 2014.

[2] N. Taušan, J. Markkula, P. Kuvaja, and M. Oivo, "Choreography in the embedded systems domain: A systematic literature review," *Information and Software Technology*, Jun. 2017.

[3] S. Keller, M. Tivoli, M. Autili, and C. Thomas, "CHOReVOLUTION: Dynamic and Secure Choreographies of Services," CHOReOS, White paper, Mar. 2017.

[4] W. Jaradat, A. Dearle, and A. Barker, "Towards an autonomous decentralized orchestration system," *Concurrency Computat.: Pract. Exper.*, vol. 28, no. 11, pp. 3164–3179, Aug. 2016.

[5] G. Chafle, S. Chandra, and V. Mann, "Decentralized Orchestration of Composite Web Services," in *Proceedings of the 13th International WWW Conference*, 2004, pp. 134–143.

[6] W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. ter Hofstede, "Design and Implementation of the YAWL System," in *Advanced Information Systems Engineering*. Springer, Berlin, Heidelberg, Jun. 2004, pp. 142–159.

[7] D. Arellanes and K.-K. Lau, "Exogenous Connectors for Hierarchical Service Composition," in *Proceedings of the 10th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2017)*. Kanazawa, Japan: IEEE Computer Society, 2017.

[8] S. Newman, *Building Microservices*, 1st ed. Beijing Sebastopol, CA: O'Reilly Media, Feb. 2015.

[9] E. Kuehn, "MozartSpaces," http://www.mozartspaces.org, 2017.

4.3 Algebraic Service Composition for User-Centric IoT Applications

Damian Arellanes and Kung-Kiu Lau

In **Internet of Things – ICIOT 2018**.

Published by Springer,
Volume 10972 of Lecture Notes in Computer Science,
pages 56-69,
ISBN 978-3-319-94370-1,
2018.

Impact Indicators:
• Impact factor (2018): 1.71,
• Second most cited article amongst 14 publications in the proceedings (as of 2019),
• **Best Paper Award** by Springer and the Services Conference Federation.

The final authenticated version [AL18a] is available online at `https://doi.org/10.1007/978-3-319-94370-1_5`.

Summary: This paper introduces *workflow variability* semantics into the DX-MAN model by defining composition operators as variation points. It particularly presents the definition of abstract workflow tree and concrete workflow tree. The evaluation is done using a case study in the domain of IoT end-user applications.

Comments on authorship: I proposed the main idea of the paper, extended the model semantics, validated the new semantics, conducted a qualitative evaluation, analysed results, investigated related work, provided and edited all graphics, participated in the entire writing process and addressed the reviewer's comments. My supervisor, Kung-Kiu Lau, also contributed to the idea, proofread the paper and approved the model extension. He also guided the whole research process.

Key contributions: Contribution 4.1 (see Section 1.5).

Algebraic Service Composition for User-Centric IoT Applications

Damian Arellanes and Kung-Kiu Lau

School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

Abstract. The Internet of Things (IoT) requires a shift in our way of building applications, as it is aimed at providing many services to society in general. Non-developer people require increasingly complex IoT applications and support for their ever changing run-time requirements. Although service composition allows the combination of functionality into more complex behaviours, current approaches provide support for dealing with one IoT scenario at a time, as they allow the definition of only one workflow. In this paper, we present DX-MAN, an algebraic model for static service composition that allows the definition of composite services that encompass multiple workflows for run-time scenarios. We evaluate our proposal on an example in the domain of smart homes.

Keywords: IoT applications, algebraic service composition, scalability, exogenous connectors, end-user development, DX-MAN

1 Introduction

The Internet of Things (IoT) promises a new era in which every physical world object and all living entities will be interconnected through innovative distributed services. Thus, the scale of IoT applications will go beyond human mind expectations.

IoT applications are mainly aimed at providing value to society in general. People with no development expertise are able to control, manage and customize their own applications [6, 11]. For this reason, IoT requires a shift in our way of building applications: a developer must be able to create a generic application that encompasses multiple scenarios, in order to accommodate as much as possible the run-time user requirements. Thus, users will be able to autonomously choose a behaviour among the alternative ones.

Although some scenarios are simple, many others require the combination of a huge number of services. Hence, service composition is crucial for building complex IoT applications. However, designing a generic composite that accommodates multiple IoT scenarios is not trivial, since user requirements may vary from one scenario to another. Moreover, the dynamism of IoT applications causes an increase in the number of possible scenarios as the number of services grows.

Current composition approaches do not fulfill the demands of IoT applications that require user-centric compositions of a huge number of services. This is because their semantics allows the definition of only one workflow at a time. Thus, tackling a new scenario would require an entirely new workflow or the modification of an existing one. This paper proposes DX-MAN, an algebraic model for static IoT service composition, which enables the development of composite services that encompass many workflows so as to accommodate multiple run-time scenarios.

The rest of the paper is structured as follows. Sec. 2 presents the related work. Sec. 3 presents a motivating example. Sec. 4 describes our model for algebraic IoT service composition. Sec. 5 presents examples to show the feasibility of our model. Sec. 6 presents a discussion of our results as well as challenges related to this research. Finally, Sec. 7 presents the conclusions and the future work.

2 Related Work

Current composition approaches include orchestration, nested orchestrations, choreography, data flows and nested data flows.

Orchestration [13, 22] and *choreography* [8, 26, 27] have been used for many years in Service Oriented Architecture (SOA) and are now gaining attention for IoT applications. Orchestration defines a central coordinator for the invocation of operations in services. In order to eliminate the performance bottleneck caused by the central coordinator or to support multiple administrative domains, a number of sub-workflows can be defined in *nested orchestrations* [7, 16]. On the other hand, a choreography realizes a workflow through the collaborative and decentralized exchange of messages between the services involved. Regardless of the underlying mechanics, (nested) orchestration and choreography allow the definition of only one workflow at a time.

A data-driven workflow, or *data flow*, allows the combination of data streams from different IoT sources. It is basically a graph where nodes represent computation and edges represent data paths: a node receives data, then performs some computation and finally passes data on. Although data flows are increasingly popular for IoT applications thanks to the emergence of *mashups* [5, 14, 24], they allow the creation of one workflow at a time; and this is also true in *nested data flows* [12].

Like orchestration, data flows are considered as exogenous composition mechanisms because a workflow is defined with no knowledge of the services involved [18]. Reo [19, 23] is a declarative language for data flows, which also has the notion of exogenous connectors. Unlike DX-MAN, the composition of two Reo connectors yields a more complex connector, but not a service. Of course, a Reo connector can be transformed into a service, but this would require an extra step as it is not part of Reo semantics. More importantly, Reo allows the creation of one workflow at a time, like other data flow approaches.

Automatic service composition [1, 9, 10, 22] consists of discovering, selecting and combining services at run-time, in order to construct a workflow that fulfills

a given specification [17, 25, 28]. It does not provide new composition semantics, but it is built on top of existing ones: data flows [9], orchestration [22], choreography [1], or any combination thereof [10]. Therefore, automatic service composition also allows the definition of only one workflow at a time.

Other approaches [11, 15, 29] do not provide any composition constructs, because they are only *frameworks* or *software tools* for end-user development. Some of them provide support to define only one straightforward workflow at a time, typically a sequential one.

3 Motivating Example

To motivate our approach, this section introduces a running example. The example is in the domain of smart homes and is based on the case study presented in [22]. It consists of two independent services shown in Fig. 1: (i) a *Windows* service for opening and closing windows and (ii) a *Climate* service to turn dehumidifiers on and off. For simplicity, we only show two operations per service. The distribution of services over IoT nodes is out of the scope of this paper.

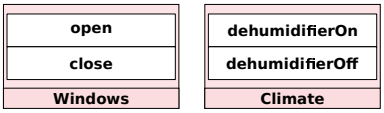


Fig. 1. Services involved in our motivating example.

Imagine a user requires different workflows at run-time depending on climatic conditions. Automatic service composition is the best approach currently available to generate workflows on the fly. However, it allows the definition of only one workflow at a time, since it is built on top of existing composition semantics.

For example, on a sunny day the user may want to open the windows and turn the dehumidifier off. The workflow depicted in Fig. 2 is generated by an automatic composition mechanism so as to accommodate this user requirement. Suppose it suddenly starts raining so the user decides to close the windows and turn the dehumidifier on. Thus, the automatic composition mechanism would need the generation of the entirely new workflow shown in Fig. 3.

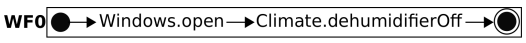


Fig. 2. Workflow for a sunny day.

Of course, the user can express all his needs in a single step. The workflow generated by the automatic composition mechanism for this scenario is shown in

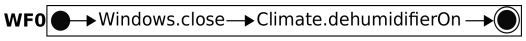


Fig. 3. Workflow for a rainy day.

Fig. 4, and it includes the scenarios depicted in Figs. 2 and 3. If the user changes his mind again, a new workflow would be needed.

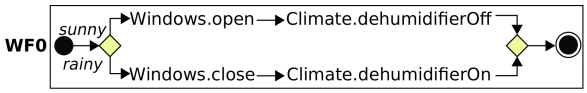


Fig. 4. Workflow for a sunny and rainy day.

It might seem that nested workflows are an alternative solution to this problem, as shown in Fig. 5. However, their composition semantics also allow the definition of only one workflow at a time. Thus, individual nested workflows are required whenever user requirements change.

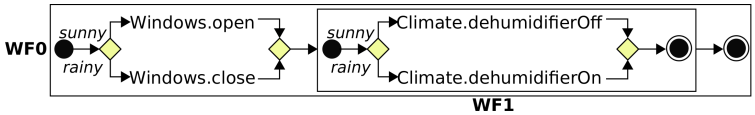


Fig. 5. Nested workflows for a sunny and rainy day.

4 DX-MAN

We propose DX-MAN (Distributed X-MAN) [3] to mitigate the impact of change of run-time user requirements. It is a multi-level service composition model [4] inspired by algebra and the X-MAN component model [20, 21], where services and exogenous connectors are first-class entities. Fig. 6 illustrates the DX-MAN constructs which we further describe in this section.

A DX-MAN service is a distributed software unit that exposes a set of operations through a well-defined interface. It can be deployed in any IoT node such as a Cloud, an edge device or a sensor. Distribution semantics are out of the scope of this paper, but we refer the reader to another paper on that matter [3].

An atomic service is the most primitive kind of DX-MAN service. It is formed by connecting an invocation connector with a computation unit (see Fig. 6). The invocation connector provides access to the operations implemented in the computation unit, and the computation unit is not allowed to call other computation units. The atomic service interface has all the operations implemented in

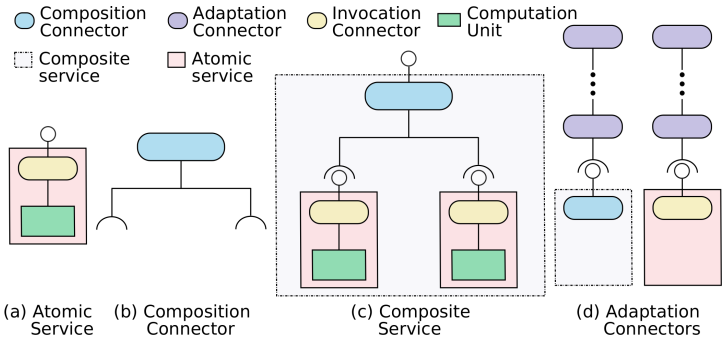


Fig. 6. DX-MAN constructs.

the computation unit. Formally, an atomic service $AS \in \mathbb{S}$, where $\mathbb{S}$ is the type of services, is a set of operations defined as follows:

$$AS = \{op_i \mid i \in \mathbb{N}\} \tag{1}$$

Exogenous connectors are architectural elements that define explicit control flow and encapsulate a network communication mechanism, in order to coordinate the execution of an IoT application from outside services. So, services are unaware they are part of a larger piece of behaviour.

Our notion of algebraic composition is inspired by algebra where functions are composed hierarchically into a new function of the same type, using the operator $\circ$. The resulting function can be further composed with other functions, yielding a more complex one.

Algebraic service composition means that a composition connector is used as an operator $(\circ)$ *to hierarchically compose* > 1 *services, atomic or composite, into a (composite) service. As it is constructed from sub-service interfaces, the composite interface has all the sub-service operations. Like an algebraic function, a composite service is a generalization of a particular problem because it implicitly contains multiple workflows whose formation is constrained by the composition connector being used.* Formally, a composite service $CS \in \mathbb{S}$, where $\mathbb{S}$ is the type of services, is a set of services defined as follows:

$$CS = \{S_i \mid i \in \mathbb{N} \wedge S \in \mathbb{S}\} \tag{2}$$

DX-MAN provides composition connectors for sequencing, branching and parallelism. A sequencer connector (SEQ) allows the invocation of sub-service operations in a user-defined order. A sub-service operation can be associated with ≥ 0 orders. Sub-service operations with no given order are never invoked, and when no sub-service operation has an order assigned, an empty workflow is thrown at run-time. Any sub-service operation can be invoked any number of times within a workflow. Thus, a sequencer connector defines a composite service that contains an infinite number of sequential workflows. Fig. 7 shows an example of a composite service constrained by a sequencer connector.

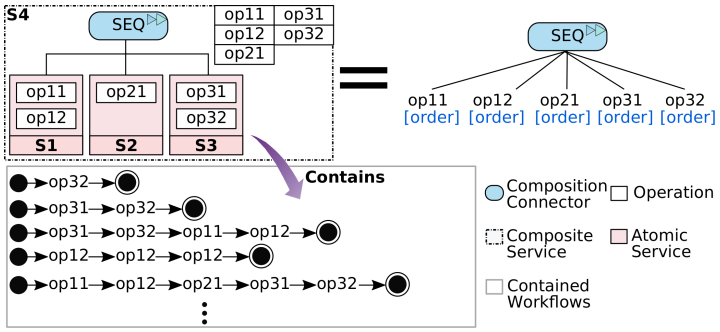


Fig. 7. Sequencer connector.

A selector (*SEL*) connector chooses the sub-service operations to be invoked, according to user-defined conditions which are evaluated concurrently. A sub-service operation can be associated with exactly zero or one condition. Sub-service operations with no condition associated are never invoked. When no sub-service operation has a condition associated or all conditions hold false, an empty workflow is thrown at run-time. A selector connector defines a composite service that contains $2^{|\bigcup_{i=1}^{|CS|} S_i|}$ workflows. For example, Fig. 8 shows a composite service that contains 32 possible branching workflows as there are five sub-service operations. We do not show all possible workflows because of space constraints.

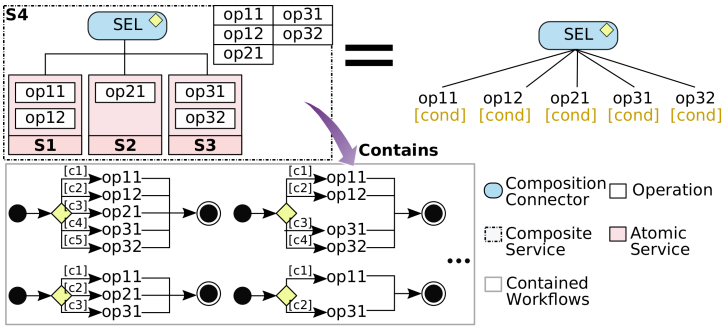


Fig. 8. Selector connector.

A parallel connector (*PAR*) allows the parallel invocation of sub-service operations. A sub-service operation can be invoked multiple times in parallel within a workflow; to do so, the user needs to specify the number of jobs for each sub-service operation. When no sub-service operation has jobs assigned, an empty workflow is thrown at run-time. A parallel connector defines a composite ser-

vice that contains infinite parallel workflows. Fig. 9 shows a composite service constrained by a parallel connector.

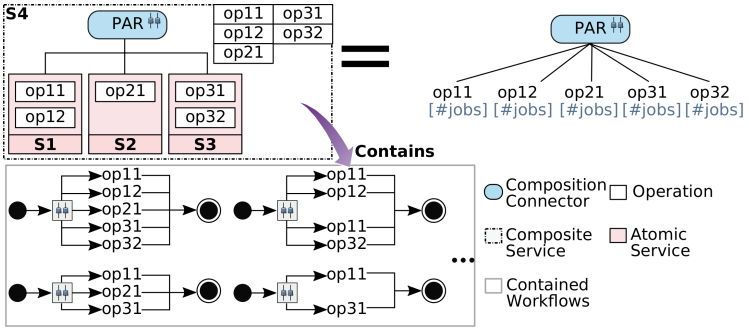


Fig. 9. Parallel connector.

Although they do not compose services, adapters can also constrain workflows by applying additional control structures over an individual service. A looping adapter can be used to iterate a number of times over a sub-workflow, while a user-defined condition holds true. A guard adapter invokes a sub-workflow only if a user-defined condition is true.

Selection trees are abstract templates that allow the selection of workflows at run-time. They are implicitly created from a composite service during design-time. Figs. 7, 8 and 9 show examples of selection trees for a sequencer connector, selector connector and parallel connector, respectively. In the next section, we present examples that show how to choose workflows using selection trees.

5 Examples

This section presents two examples of using DX-MAN for user-centric IoT applications. The first example describes how a one-level composite service accommodates the run-time user requirements described in our motivating example (see Sec. 3). The second example describes how a two-level composite service enables more complex workflows by hierarchically composing services. For both examples, we distinguish between developers and users. Developers design, deploy and execute DX-MAN services, while users choose the workflow they need at run-time. To do so, we developed a platform prototype [2].[1] Composite services and selection tree instances are defined using JavaScript Object Notation (JSON) documents. Due to space constraints and clarity, we omit the JSON documents used for the examples. Instead, we show a graphical representation of composite services and selection trees.

[1] `https://gitlab.cs.man.ac.uk/mbaxrda2/DX-MAN`

5.1 One-level Composition

At design-time, the developer uses a sequencer connector *SEQ0* to compose the services *Windows* and *Climate* into a composite service *C0* which contains infinite sequential workflows (see Fig. 10). At run-time, the user only chooses the workflow he needs from the composite *C0*.

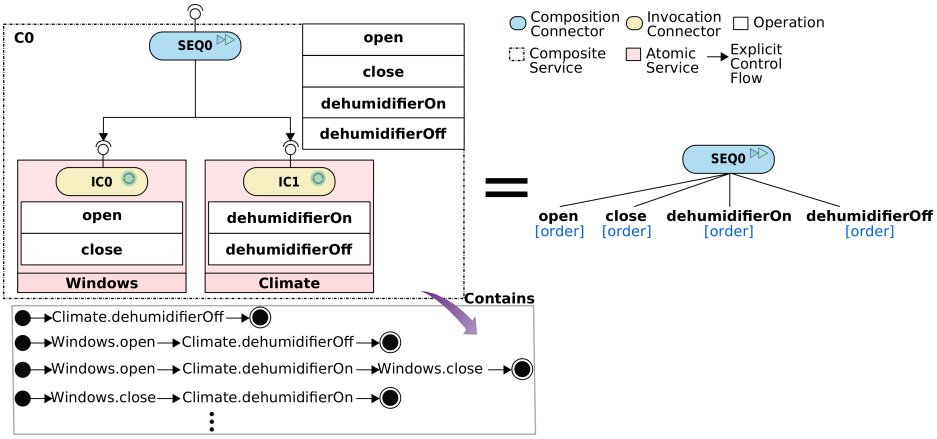


Fig. 10. DX-MAN architecture for the scenarios of our motivating example.

For example, on a sunny day the user chooses the workflow depicted in Fig. 2 in Sec. 3, by assigning execution order as shown in Fig. 11. Suddenly, it starts raining so the user chooses the workflow illustrated in Fig. 3 in Sec. 3, by assigning execution order as shown in Fig. 12. Thus, there is clearly no need of creating an individual workflow or a new composite service whenever user requirements change, but only defining an instance of the respective selection tree.

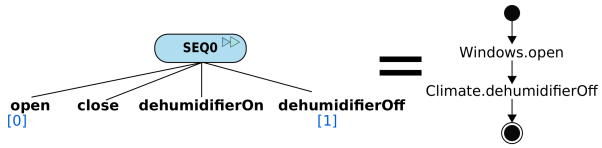


Fig. 11. Choosing a workflow for a sunny day.

As another example, on a cold day the user may want to only close the windows. To do so, the user assigns the execution order shown in Fig. 13. Again, without the need of creating an individual workflow or a new composite service.

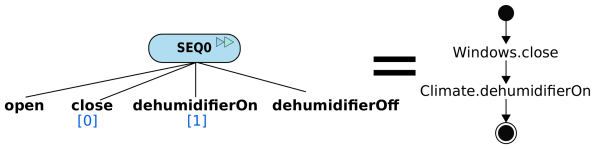


Fig. 12. Choosing a workflow for a rainy day.

Fig. 13. Choosing a workflow for a cold day.

5.2 Two-level Composition

In the previous subsection, we presented a one-level composition as a solution for our motivating example. Nevertheless, DX-MAN allows more complex workflows by hierarchically composing services into multi-level structures.

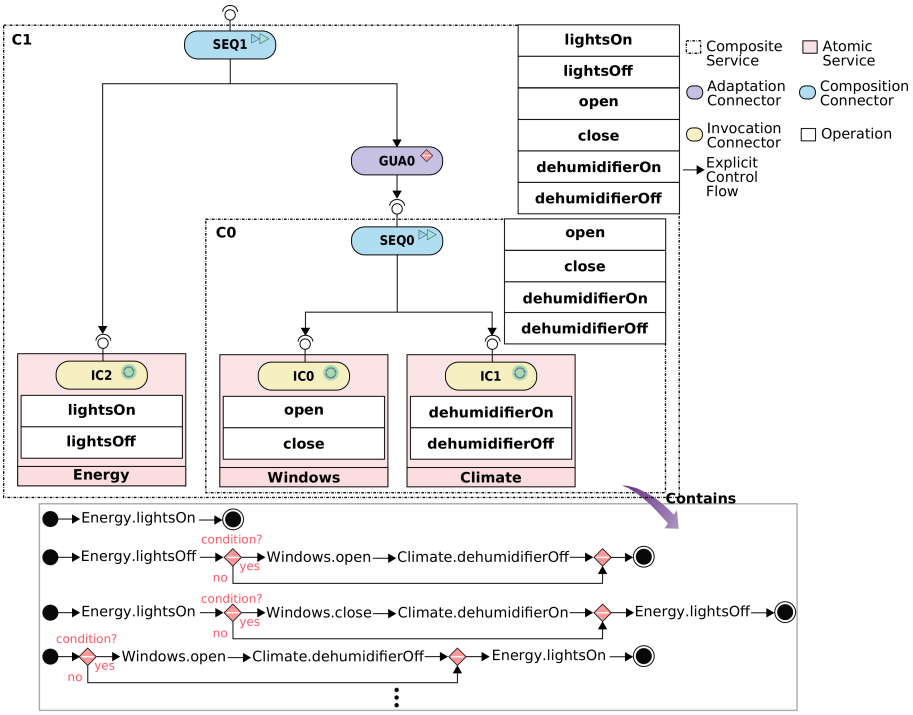


Fig. 14. Two-level DX-MAN architecture.

Suppose there is an atomic service *energy* for turning lights on and off. The developer uses a sequencer *SEQ1* to compose the existing composite *C0* and the atomic service *energy* into a new composite *C1*. He also adds a guard adapter to invoke *C0* if a user-defined condition holds true. Fig. 14 shows the resulting two-level DX-MAN composition, and Fig. 15 shows the respective selection tree.

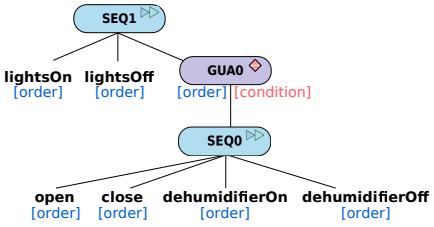


Fig. 15. Resulting tree from the two-level DX-MAN architecture.

Unlike nested workflows, a DX-MAN composite service enables an entirely new world of alternative workflows as shown in Fig. 14. For example, the user may want the following workflow before sleeping: turn the lights off and, if it all the lights were successfully turned off, close the windows and turn the dehumidifier off. To choose that workflow from *C1*, the user assigns the execution order shown in Fig. 16. A condition is represented as a JSON document and specifies the name of the parameter, the operator (only "==" and "!=" are supported at this stage) and the value to compare with. For example, the condition for *GUA0* would be { *"parameterName":"lightsStatus", "operator":"==", "value":"off"*}.

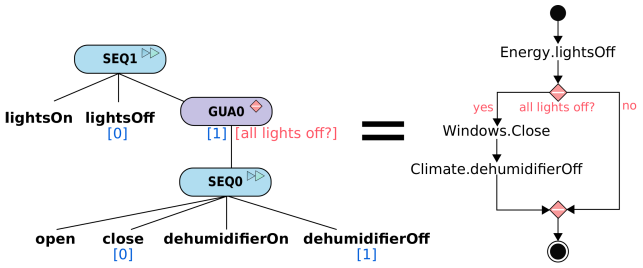


Fig. 16. Choosing a workflow before sleeping.

6 Discussion

We presented a preliminary version of DX-MAN in another paper [3]. In this paper, we present additional semantics that allows the selection of workflows at

run-time. We also present a comparison between DX-MAN and current composition approaches in the context of user-centric IoT applications.

Developers can use current composition semantics (e.g., orchestration or choreography) to define a workflow that accommodates as many run-time scenarios as possible. However, it is impossible for them to predict all possibilities during the design-phase and, even if they try, the resulting workflow would potentially require a lot of computing resources, because it becomes larger, more complex and cumbersome as the number of possible scenarios increases. This is in fact highly likely in IoT applications where the number of available services is always growing.

Although automatic service composition mechanisms could mitigate the ever changing run-time user requirements, their overhead increases exponentially as the number of available services grows [9]. Thus, they are only suitable for a small number of services and straightforward workflows. A large number of services would require a user to wait hours (or even days) before getting a responsive application. For that reason, current automatic composition mechanisms are not yet ready to tackle the imminent scale of user-centric IoT applications.

Even though it is focused on static composition, DX-MAN provides semantics to enable multiple workflows at run-time. In some cases, it may be necessary to change a DX-MAN composition at run-time so as to support even more scenarios. This can be done using automatic composition or dynamic reconfiguration techniques on top of DX-MAN semantics.

In contrast to other composition approaches, DX-MAN does not entail much composition overhead, since there is no need to deploy individual workflows, but only a composite service from which a workflow is chosen (not created) at run-time. In fact, IFTTT or any similar tool can be used on top of DX-MAN to choose a workflow, according to a set of user-defined rules.

At this point, the reader may notice that there are clearly many challenges for future work. We discuss some of them below.

Automatic service composition. We believe that our work opens new opportunities for automatic service composition, as this technique can be applied on top of DX-MAN semantics. Services (with all their implicit workflows) can be composed to find more possible workflows at run-time, rather than attempting to construct only one workflow at a time. We are particularly interested in decentralized approaches for automatic service composition, since decentralization is crucial to unleash the full potential of IoT.

Self-adaptive behaviour. Self-adaptive mechanisms can built on top of DX-MAN to autonomically choose a workflow out of the alternative ones, e.g., based on QoS requirements. A DX-MAN composite service can mutate so as to accommodate changes in the context. However, changing a composition at run-time is not trivial, specially when the response time is critical for the user.

Workflow validation at run-time. As a sequencer connector currently allows the invocation of any operation in any order, there is a need for avoiding invalid

sequences (e.g., opening a window three consecutive times). At this stage, it is up to the user to decide which workflows are valid.

Concurrency. DX-MAN only provides support for basic concurrency in parallel invocations. However, many IoT scenarios require active services that can be operating on their own (e.g., using a scheduler). Extending DX-MAN with concurrent capabilities requires further investigation.

Data flows at run-time. In DX-MAN, data flow is orthogonal to control flow. Current DX-MAN semantics only allow one data flow for every possible workflow within a composite service. For that reason, at this stage DX-MAN can only be used in scenarios where data flow is unimportant, e.g., actuator triggering. In more complex IoT scenarios, different data flows per workflow will be required. Nevertheless, determining data flows at run-time according to user requirements is a challenging task.

7 Conclusions and Future Work

Users may want to customize their own IoT applications. However, current composition approaches allow the definition of only one workflow at a time. This is not desirable for IoT applications where run-time user requirements are always changing. Although automatic composition is a promising technique to tackle this problem, it is still based on existing composition semantics, thus allowing the definition of only one workflow at a time. For that reason, we need to accommodate run-time user requirements as much as possible during the design phase. In this paper, we presented DX-MAN as a solution for this issue.

The algebraic nature of DX-MAN is suitable to mitigate the impact of change in run-time user requirements. We showed with a small example how DX-MAN allows the definition of (general) composite services that contain multiple workflows. Users only choose the workflow they need out of the alternative ones, rather than resort to the cumbersome and inefficient task of creating individual workflows at run-time.

In the short term, we plan to extend the DX-MAN semantics, in order to enhance the flexibility of composite services. Additionally, as workflows are chosen using JSON documents at this stage, we would like to allow the selection of workflows in a more interactive way (e.g., using a visual tool or voice commands). We are in fact currently working on a visual Web editor to fill this gap.

We believe that DX-MAN opens new research directions to tackle the challenges that user-centric IoT applications pose. Given the novelty of DX-MAN, in what creative ways can you define composite services during the design-phase, in order to accommodate as much as possible run-time user requirements?

References

1. Ahmed, T., Tripathi, A., Srivastava, A.: Rain4Service: An Approach towards Decentralized Web Service Composition. In: IEEE International Conference on Services Computing (SCC '14). pp. 267–274 (2014)
2. Arellanes, D., Lau, K.K.: D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures. In: 7th IEEE International Symposium on Cloud and Service Computing (SC2 '17). pp. 283–286 (2017)
3. Arellanes, D., Lau, K.K.: Exogenous Connectors for Hierarchical Service Composition. In: 10th IEEE International Conference on Service Oriented Computing and Applications (SOCA '17). pp. 125–132 (2017)
4. Arellanes, D., Lau, K.K.: Analysis and Classification of Service Interactions for the Scalability of the Internet of Things. In: IEEE International Congress on Internet of Things (IEEE ICIOT 2018) (2018)
5. Blackstock, M., Lea, R.: WoTKit: A lightweight toolkit for the web of things. In: 3rd International Workshop on the Web of Things (WoT '12). pp. 1–6 (2012)
6. Brambilla, M., Umuhoza, E., Acerbis, R.: Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. Journal of Internet Services and Applications 8(1), 14 (2017)
7. Chafle, G., Chandra, S., Mann, V.: Decentralized Orchestration of Composite Web Services. In: 13th International World Wide Web Conference (WWW '04). pp. 134–143 (2004)
8. Cherrier, S., Ghamri-Doudane, Y., Lohier, S., Roussel, G.: D-LITe : Distributed Logic for Internet of Things sErvices. In: International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing (ITHINGSCPSCOM '11). pp. 16–24 (2011)
9. Ciortea, A., Boissier, O., Zimmermann, A., Florea, A.M.: Responsive Decentralized Composition of Service Mashups for the Internet of Things. In: 6th International Conference on the Internet of Things (IoT '16). pp. 53–61 (2016)
10. Dar, K., Taherkordi, A., Vitenberg, R., Rouvoy, R., Eliassen, F.: Adaptable service composition for very-large-scale Internet of Things systems. In: 11th Middleware Doctoral Symposium (MDS '11). pp. 1–2 (2011)
11. Ghiani, G., Manca, M., Paternò, F., Santoro, C.: Personalization of Context-Dependent Applications Through Trigger-Action Rules. Transactions on Computer-Human Interaction (TOCHI) 24(2), 14:1–14:33 (2017)
12. Giang, N.K., Blackstock, M., Lea, R., Leung, V.C.M.: Developing IoT applications in the Fog: A Distributed Dataflow approach. In: 5th International Conference on the Internet of Things (IOT '15). pp. 155–162 (2015)
13. Glombitza, N., Ebers, S., Pfisterer, D., Fischer, S.: Using BPEL to Realize Business Processes for an Internet of Things. In: 3rd International Conference on Ad-Hoc Networks and Wireless (ADHOCNETS '11). pp. 294–307 (2011)
14. Guinard, D., Trifa, V., Wilde, E.: A resource oriented architecture for the Web of Things. In: Internet of Things (IOT '10). pp. 1–8 (2010)
15. IFTTT: IFTTT. https://ifttt.com/ (2018)
16. Jaradat, W., Dearle, A., Barker, A.: Towards an autonomous decentralized orchestration system. Concurrency Computat.: Pract. Exper. 28(11), 3164–3179 (2016)
17. Jatoth, C., Gangadharan, G.R., Buyya, R.: Computational Intelligence Based QoS-Aware Web Service Composition: A Systematic Literature Review. IEEE Transactions on Services Computing 10(3), 475–492 (2017)

18. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in Dataflow Programming Languages. ACM Comput. Surv. 36(1), 1–34 (2004)
19. Jongmans, S.S., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Orchestrating web services using Reo: From circuits and behaviors to automatically generated code. Service Oriented Computing and Applications 8(4), 277–297 (2014)
20. Lau, K.K., Tran, C.M.: X-MAN: An MDE Tool for Component-Based System Development. In: 2012 38th Euromicro Conference on Software Engineering and Advanced Applications. pp. 158–165 (2012)
21. Lau, K.K., Velasco Elizondo, P., Wang, Z.: Exogenous Connectors for Software Components. In: 8th Int. Conference on Component-Based Software Engineering. pp. 90–106 (2005)
22. Lee, C., Wang, C., Kim, E., Helal, S.: Blueprint Flow: A Declarative Service Composition Framework for Cloud Applications. IEEE Access 5, 17634–17643 (2017)
23. Palomar, E., Chen, X., Liu, Z., Maharjan, S., Bowen, J.: Component-Based Modelling for Scalable Smart City Systems Interoperability: A Case Study on Integrating Energy Demand Response Systems. Sensors 16(11), 1810 (2016)
24. Persson, P., Angelsmark, O.: Calvin – Merging Cloud and IoT. Procedia Computer Science 52, 210–217 (2015)
25. Sheng, Q., Qiao, X., Vasilakos, A., Szabo, C., Bourne, S., Xu, X.: Web services composition: A decade's overview. Information Sciences 280, 218–238 (2014)
26. Taušan, N., Markkula, J., Kuvaja, P., Oivo, M.: Choreography in the embedded systems domain: A systematic literature review. Information and Software Technology 91, 82–101 (2017)
27. Thuluva, A., Bröring, A., P. Medagoda Hettige Don, G., Anicic, D., Seeger, J.: Recipes for IoT Applications. In: 7th International Conference on the Internet of Things (IOT '17) (2017)
28. Wang, S., Zhou, A., Yang, M., Sun, L., Hsu, C.H., yang, f.: Service Composition in Cyber-Physical-Social Systems. IEEE Transactions on Emerging Topics in Computing (2017)
29. Zapier: Zapier — The easiest way to automate your work. https://zapier.com/ (2018)

4.4 Workflow Variability for Autonomic IoT Systems

Damian Arellanes and Kung-Kiu Lau

In proceedings of the **16th International Conference on Autonomic Computing (ICAC 2019)**.

Published by IEEE,
pages 24-30,
ISBN 978-1-7281-2411-7,
2019.

Impact Indicators:
- Conference Core ranking (2018): B,
- Acceptance rate (2019): 30.00%,
- Impact factor (2018): 3.18.

The final authenticated version [AL19c] is available online at `https://doi.org/10.1109/ICAC.2019.00014`.

Summary: This paper advances the theory of *workflow variability* by extending the semantics of DX-MAN with both autonomic capabilities and the notion of *workflow spaces*. It outlines a comparison with related work and presents a qualitative evaluation using a case study in the domain of autonomous smart homes.

Comments on authorship: I proposed the main idea of the paper, extended the model semantics, validated the new semantics, conducted a qualitative evaluation, analysed results, investigated related work, provided and edited all graphics, participated in the entire writing process and addressed the reviewer's comments. My supervisor, Kung-Kiu Lau, also contributed to the idea, proofread the paper and approved the model extension. He also guided the whole research process.

Key contributions: Contribution 4.1 (see Section 1.5).

Workflow Variability for Autonomic IoT Systems

Damian Arellanes and Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

***Abstract*—Autonomic IoT systems require variable behaviour at runtime to adapt to different system contexts. Building suitable models that span both design-time and runtime is thus essential for such systems. However, existing approaches separate the variability model from the behavioural model, leading to synchronization issues such as the need for dynamic reconfiguration and dependency management. Some approaches define a fixed number of behaviour variants and are therefore unsuitable for highly variable contexts. This paper extends the semantics of the DX-MAN service model so as to combine variability with behaviour. The model allows the design of composite services that define an infinite number of workflow variants which can be chosen at runtime without any reconfiguration mechanism. We describe the autonomic capabilities of our model by using a case study in the domain of smart homes.**

***Index Terms*—Internet of Things, autonomic systems, DX-MAN, exogenous connectors, algebraic service composition, workflow variability, models@runtime, smart homes, self-adaptive**

I. Introduction

The Internet of Things is an emerging paradigm that envisions the interconnection of everything through novel distributed services which are combined into complex workflows using service composition mechanisms. Workflows represent IoT systems composed of billions of services with an overwhelming number of interactions. Thus, it becomes infeasible to manually manage such systems as the scale and complexity increases.

Autonomicity is a crucial desideratum for the management of complex large-scale IoT systems operating in highly dynamic environments. It is a property that allows adapting behaviour at runtime to different contexts with minimal or no human intervention. Autonomicity thus requires workflow variability for the definition of alternative system behaviours.

Although relatively trivial in static IoT systems, changing behaviour at runtime in highly variable environments is a complex and challenging task. For that reason, variability-based autonomicity has been an active research topic for software engineering in the last decade [1], [2]. Although there are many proposals for managing variability, they fail at incorporating variability in behavioural elements (i.e., in the solution space) while avoiding the cumbersome time-consuming task of dynamic reconfiguration [1], [3].

This paper extends the semantics of the DX-MAN service model [4], [5], [6] with autonomicity capabilities for IoT systems. The semantics allows adapting workflows at runtime to different contexts without requiring any dynamic reconfiguration mechanism. Our contribution is thus two-fold: (i) *a model* that combines variability with behaviour in the solution space, while providing an infinite number of workflow variants for composite IoT services; and (ii) *an approach* that avoids dynamic reconfiguration (by using *non-deployable* and *executable only* workflows).

The rest of the paper is structured as follows. Sect. II describes the main constructs of the DX-MAN model. Sect. III presents the mechanism to realize workflow variability. Sect. IV describes the autonomicity dimension of the model. Sect. V presents a case study to show autonomicity in a case study. Sect. VI describes the related work. Finally, Sect. VII presents the conclusions and the future work.

II. DX-MAN Model

DX-MAN is an algebraic model for IoT systems where services and exogenous connectors are first-class entities. An exogenous connector is a deployable entity that executes multiple workflows with explicit control flow. A service S is a stateless distributed software unit with a well defined interface, which can be either atomic (A) or composite (C):

$$S := A|C \tag{1}$$

A service defines a workflow space W which is a non-empty (finite or infinite) set, where each $w \in W$ is a workflow variant that represents an alternative service behaviour. The workflow space constitutes the service interface, and is semantically equivalent to a service S:

$$S \equiv W = \{w_1, w_2, \ldots\} \tag{2}$$

A. Atomic Services

An atomic service A is a tuple $\langle IC, O\rangle$ consisting of an invocation connector IC and a non-empty finite set O of j primitive operations (Fig. 1). It is formed by connecting an invocation connector with a computation unit.

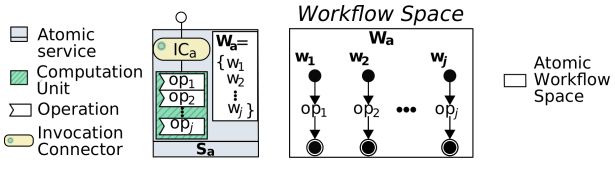


Fig. 1. A DX-MAN atomic service defines j workflows: $|W| = j$.

A computation unit is not allowed to call other computation units, and is the place where j service operations are implemented using well-known technologies such as REST. To satisfy

an external request, an invocation connector is responsible for executing a workflow in W.

Fig. 1 shows that an atomic service $S_a \in A$ defines an atomic workflow space W_a s.t. $|W_a| = j$ and each $w_{i\in[1,j]} \in W_a$ is a workflow invoking an operation $op_{i\in[1,j]} \in O$. The atomic workflow space W_a is the interface of S_a.

B. Algebraic Composition

Our notion of algebraic service composition is inspired by algebra where functions are hierarchically composed into a new function of the same type. The resulting function can be further composed with other functions, yielding a more complex one. Algebraic service composition is then the operation by which a composition connector composes k services into a more complex service. The result is a (hierarchical) composite service whose interface is constructed from the sub-service interfaces. Formally, a composite service is a tuple $\langle CC, \mathcal{W} \rangle$ consisting of:

- a composition connector CC that invokes multiple workflows defined by the composite service, and
- a non-empty finite $\mathcal{W}$ set which is a family of non-empty (finite of infinite) sets of sub-workflow spaces s.t. each $W_i \in \mathcal{W}, i = 1, \ldots, k$ is a workflow space of either an atomic sub-service or a composite sub-service.

A composite service is a variation point which defines a new non-empty (finite or infinite) workflow space W using the sub-workflow spaces $\mathcal{W}$ via algebraic references (Fig. 2). W serves as the composite service interface, and is available to more complex composites.

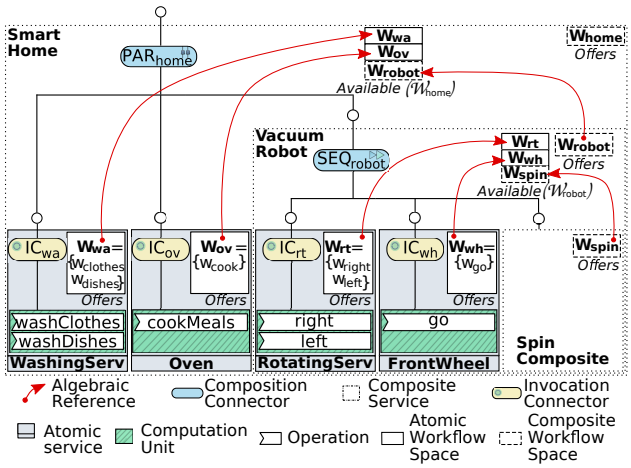


Fig. 2. Algebraic Composition for a Smart Home.

Fig. 2 depicts a two-level DX-MAN composition for a smart home with four atomic services (i.e., *WashingServ*, *Oven*, *RotatingServ* and *FrontWheel*) and three composite services (i.e., *SpinComposite*, *VacuumRobot* and *SmartHome*). The services are described in Sect. III. For the sake of clarity, we omit the internal structure of *SpinComposite*, but we show its interface: the composite workflow space W_{spin}. The interfaces of *WashingServ*, *Oven*, *RotatingServ* and *FrontWheel* are the atomic workflow spaces $W_{wa} = \{w_{clothes}, w_{dishes}\}$, $W_{ov} = \{w_{cook}\}$, $W_{rt} = \{w_{right}, w_{left}\}$ and $W_{wh} = \{w_{go}\}$, respectively. The services *RotatingServ*, *FrontWheel* and *SpinComposite* are composed into *VacuumRobot* (using the composition connector SEQ_{robot}, see Fig. 3). Thus, the interfaces W_{rt}, W_{wh} and W_{spin} are available in *VacuumRobot* which, in turn, defines the composite workflow space W_{robot}. Then, *WashingServ*, *Oven* and *VacuumRobot* are composed into the top-level composite *SmartHome* (using the composition connector PAR_{home}, see Fig. 6). So, *SmartHome* has available the interfaces W_{wa}, W_{ov} and W_{robot}, and yields the composite workflow space W_{home}.

C. Workflow Selection

A composition connector CC is a variability operator that defines the alternative behaviours of a composite service. It is a function that defines a workflow space W, given a family of sub-workflow spaces $\mathcal{W}$:

$$CC : \mathcal{W} \mapsto W \tag{3}$$

A composition connector has access to atomic sub-workflow spaces, but not to composite sub-workflow spaces. This is because a composite sub-service is a black box whose behaviour is unknown. Hence, a composition connector operates on n elements to define sequential, branching or parallel workflows for a composite $c \in C$. The total number of elements n is the sum of the cardinality of atomic sub-workflow spaces and the number of composite sub-services:

$$n = \sum_{i=1}^{|\mathcal{W}_c|} \begin{cases} |W_c^i| & s_c^i \in A \\ 1 & s_c^i \in C \end{cases} \tag{4}$$

where $\mathcal{W}_c \in \mathcal{W}$ is the set of sub-workflow spaces of the composite c, $n \geq |\mathcal{W}_c|$ and $W_c^i \in \mathcal{W}_c$ is the workflow space of a sub-service S_c^i.

At design-time, an *abstract workflow tree* is automatically created for a composite service, as a result of composition. It represents the hierarchical control flow structure of a composite service, where n leaves are atomic workflows, composite workflow spaces or any combination thereof (e.g., Fig. 3). The leaves are also referred to as the *elements* of a workflow tree. The edges represent customizable control flow parameters (e.g., execution order or conditions) which are determined by the composition connector being used. In our current implementation, abstract workflow trees are JSON objects.

A *concrete workflow tree* enables the selection of a workflow variant at runtime. It particularly sets specific values for the customizable control flow parameters of an abstract workflow tree, in order to select the elements (i.e., atomic workflows or composite workflow spaces) to include in a workflow out of n possibilities (e.g., Fig. 4). In our current implementation, concrete workflow trees are also JSON objects.

III. Composition Connectors as Variability Operators

This section describes some of the composition connectors currently supported by DX-MAN, namely sequencer, parallelizer and exclusive selector. Although the inclusive selector is also supported, we do not describe it due to space constraints.

A. Sequencer

A *sequencer* connector SEQ uses the Kleene star operation to allow the repetition of n elements, resulting in infinite sequences. It then defines an infinite workflow space for a composite service s.t. each $w_i \in W, i = 1, \ldots, \infty$ is a sequential workflow. A sequencer is a function defined as:

$$SEQ : \mathcal{W} \mapsto W \tag{5}$$

where $|W| = \infty$.

1) Example: Consider a vacuum robot that cleans a room in a smart home using a composite service *VacuumRobot*. It relies on two atomic services and one composite service to navigate efficiently, as shown by Fig. 3. The atomic service *RotatingServ* provides two operations for turning the robot to the *left* and *right*, respectively. The atomic service *FrontWheel* offers the operation *go* to move the robot one unit forward. There is also a *SpinComposite* service that enables the robot to spin 360°, in order to clean the dirtiest areas of the room. For clarity, we do not show the internal structure of *SpinComposite*.

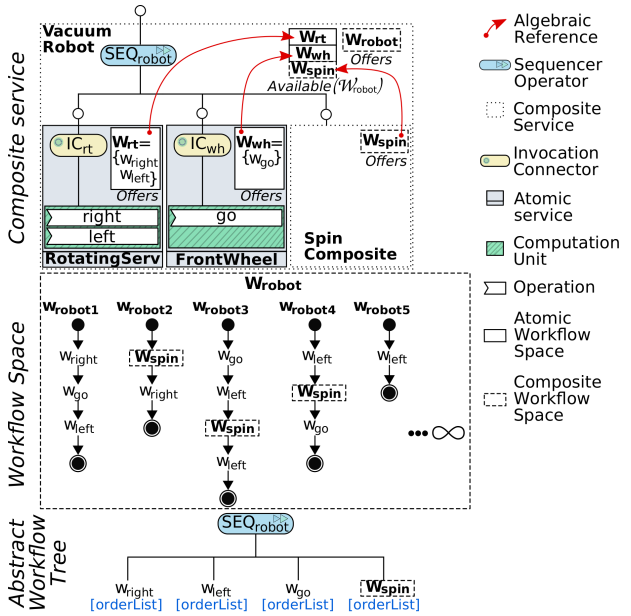


Fig. 3. A sequencer defines ∞ workflows for a composite service: $|W| = \infty$. In this example, there are ∞ sequential workflows for *Vacuum Robot*.

The sequencer connector SEQ_{robot} composes the services *RotatingService*, *FrontWheel* and *SpinComposite* into *VacuumRobot*, resulting in the infinite workflow space W_{robot}. Fig. 3 illustrates a few workflow variants for *VacuumRobot*. For instance, the variant w_{robot4} indicates that the atomic workflow w_{left} is executed before the composite workflow space W_{spin} which, in turn, is executed before the atomic workflow w_{go}. Note that W_{spin} cannot be accessed by the *VacuumRobot* since the *SpinComposite* sub-service is a black box entity which can take any possible behaviour. Instead, only atomic workflow spaces (i.e., W_{rt} and W_{wh}) can be accessed.

2) Workflow Selection: An abstract workflow tree of a sequencer requires the specification of the execution order for n elements. An execution order is a non-negative integer that reflects the position of an element in a workflow. As a sequencer allows repetition, an element requires an order list $[order_1, order_2, \ldots]$, as shown by Figs. 4 and 5. Elements with no order lists are not included in a workflow and, to ensure consistent sequences, an order cannot appear in multiple lists.

Fig. 4 shows an example of a concrete workflow tree for choosing the sequential workflow w_{robot3} for the composite *VacuumRobot*. The element w_{right} is left out as it does not have any order list. Fig. 5 illustrates another example for the selection of the sequential workflow w_{robot1} which now excludes the composite workflow space W_{spin}.

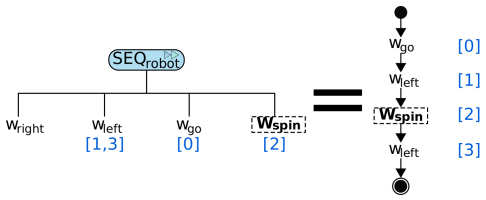


Fig. 4. Concrete workflow tree for choosing the sequential workflow w_{robot3} for the $VacuumRobot$ composite.

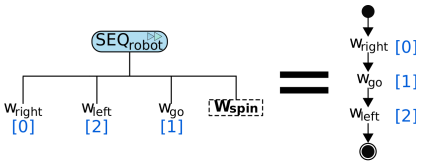


Fig. 5. Concrete workflow tree for choosing the sequential workflow w_{robot1} for the $VacuumRobot$ composite.

B. Parallelizer

A *parallelizer* connector PAR allows the execution of multiple elements in parallel. As it supports element repetition, it defines ∞ parallel workflows for a composite service s.t. each $w_i \in W, i = 1, \ldots, \infty$ is a workflow executing all the elements in parallel. Formally, a parallelizer is a function defined as:

$$PAR : \mathcal{W} \mapsto W \tag{6}$$

where $|W| = \infty$.

1) Example: Consider the composition depicted in Fig. 6 where *SmartHome* is the top-level composite which is able to do the chores for a user. The atomic service *WashingServ* provides the operations *washClothes* and *washDishes* for washing clothes and washing dishes, respectively. The atomic service *Oven* offers the operation *cookMeals* for cooking breakfast, lunch and dinner in a specific day. The composite service *VacuumRobot*, previously presented in Fig. 3, is also available for the smart home. For clarity concerns, we omit the internal structure of *VacuumRobot* and we only show the respective interface.

A parallelizer connector PAR_{home} composes *WashingServ*, *Oven* and *VacuumRobot* into *SmartHome*, resulting in the

workflow space W_{home} of infinite parallel workflows. Some workflow variants are displayed in Fig. 6. For instance, the variant w_{home2} executes the atomic workflows $w_{clothes}$ and w_{cook} in parallel. w_{home4} is another variant that leverages the support for repetition so as to execute the atomic workflow w_{cook} in three different tasks. This is useful for cooking three meals for three different people simultaneously.

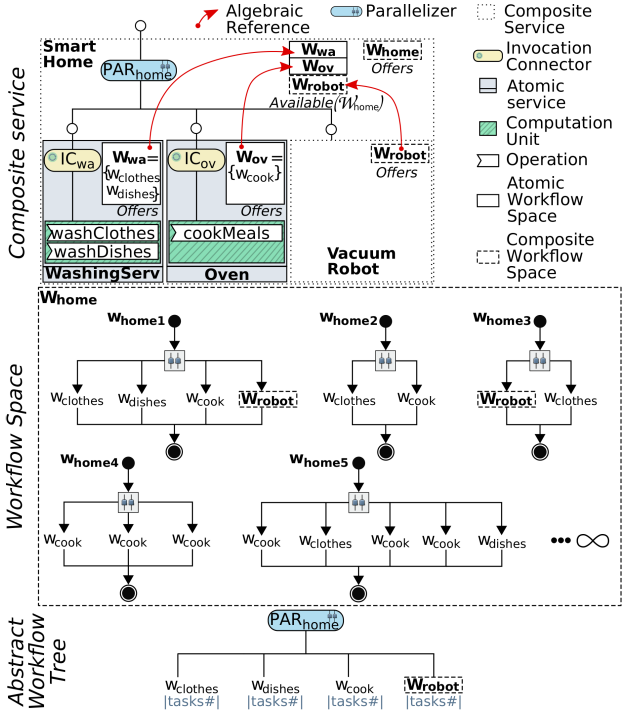


Fig. 6. A parallelizer defines ∞ workflows for a composite service: $|W| = \infty$. In this example, there are ∞ parallel workflows for *SmartHome*.

2) Workflow Selection: The abstract workflow tree of a parallelizer allows the selection of elements to include in a parallel workflow, and there are n elements that can be selected with repetition allowed. Each element requires the specification of a natural number that represents the number of tasks for that particular element, and elements with no tasks are excluded from the workflow being constructed. A task basically represents the number of times an element is repeated in a parallel workflow. So, at runtime it is an invocation thread.

Fig. 7 shows a concrete workflow tree for choosing the variant w_{home5}. It defines three tasks for the atomic workflow w_{cook}, one task for the atomic workflow $w_{clothes}$ and another one for the atomic workflow w_{dishes}. This means that the smart home washes dishes, prepares three meals and washes clothes at the same time. The composite workflow space W_{robot} is excluded from w_{home5}. Fig. 8 shows another concrete workflow tree for choosing w_{home3} which only includes the composite workflow space W_{robot} and the atomic workflow $w_{clothes}$.

C. Exclusive Selector

An *exclusive selector* $XSEL$ defines a workflow space with $2^n - 1$ exclusive branching workflows for a composite service.

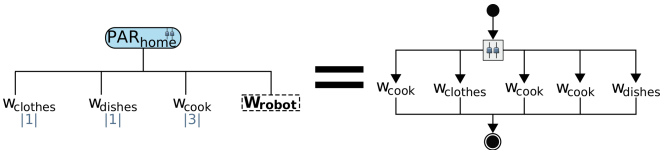


Fig. 7. Concrete workflow tree for choosing the parallel workflow w_{home5} for the $SmartHome$ composite.

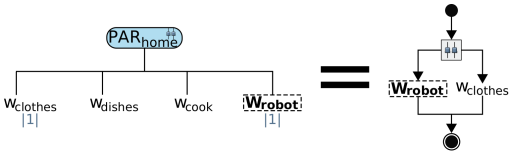


Fig. 8. Concrete workflow tree for choosing the parallel workflow w_{home3} for the $SmartHome$ composite.

Each workflow $w_i \in W, i = 1, \ldots, (2^n - 1)$ contains at least one element out of n possibilities, and chooses a single element to be executed. An exclusive selector is a function defined as:

$$XSEL : \mathcal{W} \mapsto W \tag{7}$$

where $|W| = 2^n - 1$.

1) Example: Consider a speaker controlled by a composite service *Player* for playing audio in a room. It has an atomic service *Music* that provides two operations for playing Jazz and playing pop music, respectively. There is also an atomic service *News* for reading the most recent news, and a composite service *WeatherReport* for listening to the weather forecast. For clarity, we omit the internal structure of *WeatherReport*.

Fig. 9 shows that the exclusive selector $XSEL_{play}$ composes the services *Music*, *News* and *WeatherReport* into *Player*. The composition process results in the workflow space W_{play} of $2^4 - 1 = 15$ exclusive branching workflows, as there are four elements available: the atomic workflows w_{jazz}, w_{pop} and w_{news}, and the composite workflow space $W_{weather}$. Fig. 9 illustrates some workflow variants for the composite *Player*. For instance, the workflow w_{play15} may execute w_{jazz}, w_{pop} or $W_{weather}$. Another variant is w_{play6} which chooses to play either jazz or pop.

2) Workflow Selection: The abstract workflow tree of an exclusive selector chooses the elements to include in a workflow out of n possibilities. To do so, a binary tag must be specified for each element, so elements tagged with *One* are included, whilst elements tagged with *Zero* are not included. A single condition must be specified for the entire branch because an exclusive selector applies 1 condition to multiple elements, thereby choosing only one element at a time. Thus, the maximum number of possible executions is the same number of elements included in the workflow, plus an empty execution. The empty execution means that no element is executed when the condition holds false at runtime. In our current implementation, we use Java interfaces for defining conditions.

Fig. 10 shows a concrete workflow tree for choosing the variant w_{play15} which excludes the atomic workflow w_{news}. It applies a single condition to w_{jazz}, w_{pop} and $W_{weather}$ for

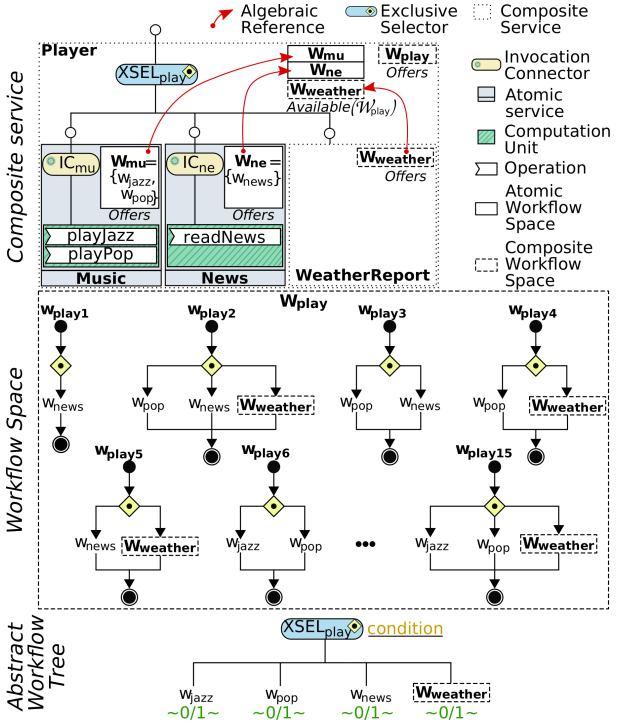


Fig. 9. An exclusive selector defines $2^n - 1$ workflows for a composite service: $|W| = 2^n - 1$. In this example, there are $2^4 - 1 = 15$ exclusive branching workflows for *Player*.

playing Jazz at nights, pop music on afternoons or the *weather forecast* in the morning. w_{play15} has four possible executions.

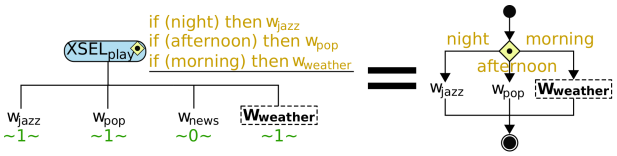


Fig. 10. Concrete workflow tree for choosing the exclusive branching workflow w_{play15} for the *Player* composite.

Fig. 11 illustrates another concrete workflow tree for choosing the workflow variant w_{play3}. It has a condition for playing pop music if there are multiple users present, or listening to the news when there is only one user. As it uses an *if-else* condition, w_{play3} enables only two possible executions.

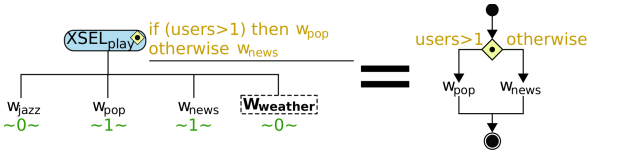


Fig. 11. Concrete workflow tree for choosing the exclusive branching workflow w_{play3} for the *Player* composite.

IV. Emergent Behaviour of DX-MAN Compositions using Feedback Control Loops

This section describes the mechanism that enables an autonomous selection of workflow variants at runtime in composite services.

In DX-MAN, workflow spaces represent the adaptation space of a composite service, since they provide a wide range of workflow variants, each representing a different behaviour. Unlike existing approaches, DX-MAN does not require to link the variability model with the behavioural model, as those dimensions are mixed in the semantics of a composite service.

The selection of workflow variants (i.e., changing behaviour) takes place at runtime whenever the context changes. This is done by building the concrete workflow tree that best adapts to the current context. For this, we use Monitoring, Analysis, Planning, Execution and Knowledge (MAPE-K) [7] which endow composite services with autonomicity. MAPE-K is a feedback control loop consisting of multiple sensors, a monitor, an analyzer, a planner, an executor, an effector and a knowledge base. Fig. 12 shows that a MAPE-K loop manages a composite service and collects information from the external context (e.g., the surrounding environment or user preferences). Remarkably, autonomicity is an orthogonal dimension to control, data and computation in the DX-MAN model.

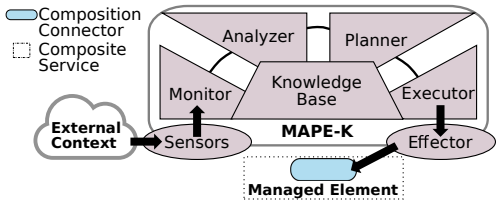


Fig. 12. MAPE-K for DX-MAN.

The MAPE-K components are able to read and update the *knowledge base* which stores relevant information for realizing autonomic behaviour. By default, the knowledge base stores the abstract workflow tree for the managed composite service.

The *monitor* uses sensor data to build a context model for the external environment, which is used by the analyzer to decide if a new behaviour is required. If so, the planner determines the best workflow variant for the current context state, resulting in a plan that is passed to the *executor* which transforms it into a concrete workflow tree matching the structure of the abstract workflow tree. Finally, the executor uses the effector to change the behaviour of the managed composite service, by executing the chosen concrete workflow tree. In our current implementation, the context model, the context state, plans and workflow trees are JSON documents. We do not show the source code due to space constraints, but JSON samples are available at https://gitlab.cs.man.ac.uk/mbaxrda2/dxman.

At runtime, control blocks when it reaches a composition connector. Once a MAPE-K determines the "best" workflow for a managed composite service, the executor resumes the workflow execution by passing a concrete workflow tree to the connector of the managed composite.

As every composite service is managed by a different MAPE-K loop, any composite at any level in the hierarchy is able to change its behaviour at runtime independently. This inevitably requires ensuring consistency for the current workflow execution. Fortunately, dynamic workflow deployment is not required since DX-MAN workflows are executable only. Whenever a new workflow is required, the effector kills the thread of the current workflow execution, thereby instantly stopping the sub-workflows being executed by the managed composite. A new thread is then created for the execution of the new workflow.

Workflow selection may potentially happen simultaneously at multiple levels in the hierarchy. So, continuously changing sub-workflows leads to an emergent behaviour of the whole system. MAPE-K loops are continuously operating, even though control flow has not yet reached the managed composition connector. However, they can only change the composite service behaviour, by executing a concrete workflow tree, when control flow has passed through or is blocked in the managed connector.

A running IoT system is practically a complex workflow consisting of sub-workflows s.t. each sub-workflow represents a composite service behaviour. This is precisely due to the hierarchical structure of a DX-MAN composition. By contrast, MAPE-K loops are not structured hierarchically as they never interact. Instead, they only select a workflow for the managed composite service (at any level in the hierarchy) and they execute new workflows (when control is blocked in the managed composition connector) or replace an existing workflow with a "better one" (when control has already passed through).

V. Case Study: Smart Home

This section presents a case study in the domain of end-user smart homes where the external context (e.g., user presence) is always changing and users are always willing a quick workflow selection. So, existing approaches for variability-based autonomicity (see Sec. VI) are not suitable for smart homes. This is because those approaches require time for changing behaviour due to dynamic reconfiguration and/or provide a limited number of variants which may not be suitable for some contexts. We leverage the capabilities of DX-MAN to avoid dynamic reconfiguration and provide a wide range of workflow variants. The DX-MAN composition for our case study is basically the composite service *SmartHome* described in Sect. II and depicted in Fig. 2. Although we endow every composite service with its own MAPE-K loop, this section just focuses on the autonomicity of *VacuumRobot* and *SmartHome*.

A. Autonomic Vacuum Robot Composite

The goal of the *VacuumRobot* composite (Fig. 3) is to clean a room as efficiently as possible by continuously changing the robot trajectory. As it operates on a dynamic environment where people is always moving, the robot changes trajectory whenever an obstacle is detected. For that, a MAPE-K loop chooses the most efficient trajectory (i.e., the best sequential workflow) that cleans every accessible areas of the room while avoiding collisions.

The MAPE-K is equipped with three range sensors that perceive the external environment of the vacuum robot. The infrared proximity sensor is used for detecting obstacles while the robot moves around. A cliff sensor is important to avoid driving over cliffs (e.g., stairwells or ledges) and a dirt sensor detects the dirtiness level on the current position of the robot.

The MAPE-K *knowledge* contains information about the surrounding map, in addition to the abstract workflow selection tree of *VacuumRobot*. The map contains information about obstacles and dirtiness levels in the room which are updated by the *monitor* to improve future navigation, and is queried when a new trajectory is required. We assume that the dirtiness levels are determined by any existing approach (e.g., Poisson processes [8]). We also assume that the map is bidimensional where each position is a disk shape fitting the robot size, as shown in [9]. In particular, a disk can be either an obstacle or a free space with a (high or normal) dirtiness level.

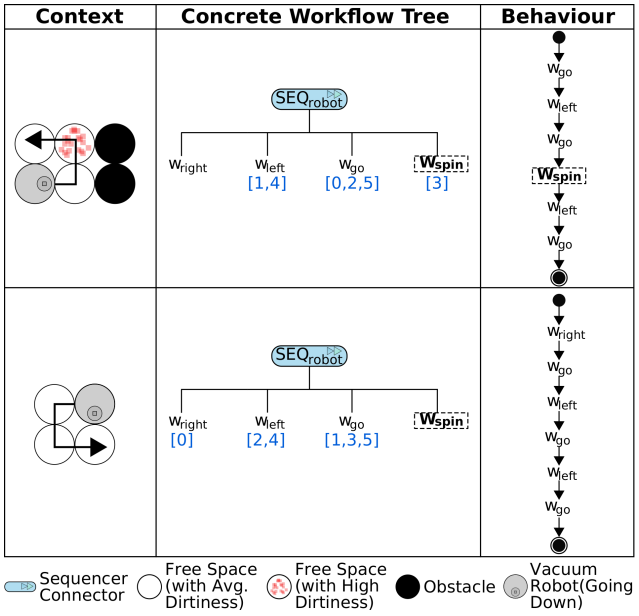


Fig. 13. Possible behaviours for the *VacuumRobot* composite.

The *analyzer* determines if there is an obstacle in the current robot position, and discovers new areas to cover. The *planner* is notified when the analyzer detects an obstacle, and uses *scan matching online cell decomposition* [10] for finding the best trajectory. To ensure a harder cleaning, we modified such an approach so as to enable trajectories where the robot spins on the dirtiest areas of the room. This mechanism is out of the scope of this paper.

The *executor* transforms the best trajectory into a sequence of actions (i.e., the best sequential workflow variant) the robot needs to carry out for the current context. For this, it uses the abstract workflow tree to build a concrete workflow tree, and then triggers the effector. Finally, the effector executes the variant by passing the concrete workflow tree to the sequencer SEQ_{robot}. The current execution (if any) is overridden (i.e., stopped and replaced) by the new workflow variant. Fig. 13

shows two possible behaviours for the *VacuumRobot* composite in two different contexts. Due to space constraints, the contexts are fragments of the map presented in [9].

B. Autonomic Manager for the Smart Home Composite

The *SmartHome* composite does chores in parallel for a user, while minimizing energy consumption and maximizing tidiness. Its behaviour changes once a day and depends on user preferences, changes in the external environment, and non-functional properties of *SmartHome* elements. Table I shows the annotated non-functional properties for $w_{clothes}$, w_{dishes}, w_{cook} and W_{robot}. The *userPresence* property takes a binary value to indicate whether the element should be executed when the user is at home (i.e., *One*) or away (i.e., *Zero*). The *energy* property defines the average discrete amount of energy (in Watts per hour) required for the execution of an element. The *tidiness* property determines the discrete level of tidiness resulting from the execution of a specific element. The sum of all *tidiness* values must be equal to *One*. It is also important to note that the non-functional properties we assume can be much more complex in other case studies.

Element	UserPresence(u)	Energy(e)	Tidiness(t)
$W_{clothes}$	0	500.0	0.25
W_{dishes}	0	350.0	0.25
W_{cook}	1	1300.0	0.10
W_{robot}	0	150.0	0.40

TABLE I
NON-FUNCTIONAL PROPERTIES FOR THE ELEMENTS OF *SmartHome*.

The *userPresence* values depend on user-defined rules which indicate to hoover and wash when the user is away, in order to avoid accidents and noise disturbances. Thus, only w_{cook} has a *userpresence* of 1.

A workflow variant $w_i \in W_{home}$ includes v elements s.t. $v \leq n$, and its properties are computed using Equations 8, 9 and 10. The *userPresence* $u(w_i)$ is an average s.t. each $u_i^x, x = 1, \ldots, v$ is the *userPresence* value of an element x of w_i. The *energy* consumption $e(w_i)$ is a sum s.t. each $e_i^x, x = 1, \ldots, v$ is the *energy* consumption of an element x of w_i. Similarly, the level of *tidiness* $t(w_i)$ is a sum s.t. each $t_i^x, x = 1, \ldots, v$ is the *tidiness* value of an element x of w_i. Thus, the workflow variant w_i with all the elements of *SmartHome* (i.e., $v = n$), provides the highest tidiness and the highest energy consumption.

$$u(w_i) = \frac{\sum_{x=1}^{v} u_i^x}{v} \tag{8}$$

$$e(w_i) = \sum_{x=1}^{v} e_i^x \tag{9}$$

$$t(w_i) = \sum_{x=1}^{v} t_i^x \tag{10}$$

The external context ϕ changes daily and is modeled by setting the user presence $u(\phi)$, the current energy cost $c(\phi)$ (in dollars per Watt-hour) and a threshold $\tau(\phi)$ which defines the maximum amount (in dollars) the user is willing to spend for energy (in a given day). We particularly define utility functions to express the quantitative level of satisfaction of workflow variants for the current context [11]. Overall, the objective is to minimize energy cost and maximize tidiness. The utility functions range from [0,1] where 0 reflects the worst satisfability and 1 means the opposite.

Equation 11 is the utility function f_1 that computes the suitability of a workflow variant $w_i \in W_{home}$ for the user presence. Equation 11 describes a piecewise utility function f_2 that determines how well w_i minimizes energy costs. Finally, Equation 13 is the utility function f_3 that computes the contribution to tidiness of w_i.

$$f_1(w_i, \phi) = 1 - \mid u(\phi) - u(w_i) \mid \tag{11}$$

$$f_2(w_i, \phi) = \begin{cases} 1 - \frac{e(w_i) \cdot c(\phi)}{\tau(\phi)} & e(w_i) \cdot c(\phi) < \tau(\phi) \\ 0 & e(w_i) \cdot c(\phi) \geq \tau(\phi) \end{cases} \tag{12}$$

$$f_3(w_i) = t(w_i) \tag{13}$$

Equation 14 computes the overall utility $U(w_i, \phi)$ of a workflow variant $w_i \in W_{home}$ for the current context ϕ. The weights ω_1, ω_2 and ω_3 define the preference of taking into account user presence, the priority of considering the energy cost and the preference of having a tidy environment, respectively. They are continuous values in the range $[0, 1]$ s.t. a higher value indicates a higher preference. For our experiments, $\omega_1 = \omega_2 = \omega_3 = 1$.

$$U(w_i, \phi) = \frac{\omega_1 \cdot f_1(w_i, \phi) + \omega_2 \cdot f_2(w_i, \phi) + \omega_3 \cdot f_3(w_i)}{\omega_1 + \omega_2 + \omega_3} \tag{14}$$

The behaviour of the *SmartHome* composite is controlled by a MAPE-K loop which has three sensors collecting information from the external context ϕ, namely user presence, current energy costs (from the energy supplier) and a threshold value (continuously changed by the user). In addition to the abstract workflow tree of *SmartHome*, the *knowledge base* includes the aforementioned utility functions, as well as context values and selected workflows from previous days. It also contains the values of the non-functional properties presented in Table I.

The *monitor* is executed once a day, and builds a relationship between context properties and sensor values. Some examples of context models are presented in Table II. The *analyzer* receives a context model as an event, and triggers an Event-Condition-Action (ECA) rule. The rule decides a new plan is required if the current context values are different from the previous day; otherwise, it executes the plan from the previous day and no planning phase is performed.

As the size of W_{home} is infinite (Fig. 6), evaluating all workflow variants is infeasible. For that reason, we propose a *planner* using a metaheuristic approach which finds the most suitable workflow for a specific context. For clarity, we reduce the space search by omitting element repetition for every $w_i \in W_{home}$. So, elements of selected workflow variants have

Day (ϕ)	UserPresence(u_ϕ)	EnergyCost(e_ϕ)	Threshold(τ_ϕ)
1	0	0.00014	0.2
2	1	0.00007	0.6
3	1	0.00012	0.3
4	0	0.00013	0.5

TABLE II
POSSIBLE CONTEXT MODELS.

only one task. As *SmartHome* has four elements (i.e., $w_{clothes}$, w_{dishes}, w_{cook} and W_{robot}), there would be $2^4 - 1 = 15$ workflow variants in W_{home}. Although $|W_{home}|$ is relatively small, we use a genetic algorithm to show what a planner would do for larger workflow spaces.

A chromosome represents a workflow variant with four boolean genes.[1] Fig. 14 shows that the order of genes is mandatory as each gene represents an element of the *SmartHome* composite, where a gene $Zero$ means that the element is not selected, whilst a gene One entails that the element has one task. For instance, the chromosome *0101* represents a workflow variant for executing w_{dishes} and W_{robot} in parallel. A population is thus a set of workflow variants representing possible solutions for the current context ϕ. Each variant is evaluated by the utility function presented in Equation 14.

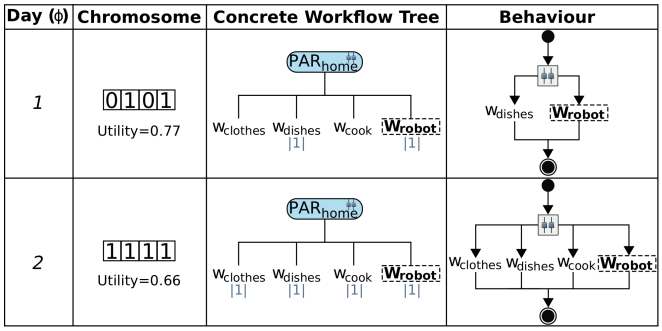


Fig. 14. Possible behaviours for the *SmartHome* composite.

After two workflow variants are selected in a generation, a one-point crossover operator is used. The crossover point is randomly selected and replaces the gene of one variant with the gene of another one. The result is two children representing two new workflow variants for the next generation. To increase diversity, we introduce mutation by randomly selecting a gene and flipping it from zero to one, or viceversa. For our implementation, we use the NSGA-II algorithm and the MOEA framework. Our source code is available at https://gitlab.cs.man.ac.uk/mbaxrda2/dxman. As this is a relatively small problem, the parameters of the genetic algorithm are as follows: population size is 8, crossover probability is 0.5, mutation probability is 0.2 and number of iterations is 20.

The result of the planner is a chromosome representing the optimal parallel workflow for the current context. The *executor* then creates a concrete workflow tree that fits the plan. Fig. 14 shows the behaviours of *SmartHome* for adapting to the context of days 1 and 2 (described in Table II). We only show two behaviours due to space constraints. To change the behaviour of the *SmartHome* composite, the *effector* passes the respective concrete workflow tree to the parallelizer PAR_{home} at runtime.

[1]For infinite workflow spaces, we could consider a chromosome where each gene is a non-negative integer in $[0, \infty]$.

VI. RELATED WORK

The related work is classified into two categories concerning workflow variability: *solution space variability* and *Models@Runtime*. We omit approaches using variability at the planning-level (e.g., [12]) as they do not propose any model constructs for supporting workflow variability, but they are built on top of existing component models with reconfiguration capabilities (e.g., Fractal [13]).

A. Solution Space Variability

The solution space captures variability at the level of composition constructs of either component models or process languages. In particular, components models define variation points using parametric variability or enumerative variability. Approaches using parametric variability [14], [15], [16] manually define a fixed number of behaviour variants at the implementation-level during design-time. Hence, there is only one workflow with multiple branching structures. Furthermore, dynamic reconfiguration is needed to change the composition structure at runtime.

Only FX-MAN [17] enumerates all possible variants in the solution space at design-time. However, it does not support service composition, requires variation generators on top of compositions, and does not addresses variability of control flow (i.e., workflow variability) and workflow selection at runtime.

Approaches extending Process Modeling Languages allow the definition of control flow constructs (e.g., activities or gateways) as variation points whose variants are realized via model transformations [2]. Most of the approaches [18], [19], [20], [21] support control flow variability only at conceptual level as they operate on non-executable models. Only few approaches [22], [23] support control flow variability via executable models (e.g., YAWL or BPEL). The main drawback is that they operate on a single flat workflow which is customized by adding, removing or replacing business process fragments via reconfiguration rules. At runtime, workflows are customized using process flexibility (i.e., dynamic reconfiguration) [24].

Other approaches [25] extend business processes with support for parametric variability. However, they also require dynamic binding at runtime and the number of variants are limited as they are manually fixed at design-time.

B. Models@Runtime

Traditional Software Product Lines (SPL) [26] enable the modeling of families of related products (i.e., workflows). As variability is separated from the behavioural model, SPL requires linking a non-executable variability model with an executable software architecture. To do so, a developer needs to implement the product in such a way that the software

architecture matches the selected features. So, SPL naturally lacks mechanisms for changing behaviour at runtime.

Dynamic Software Product Lines (DSPL) [27] change behaviour at runtime whenever the context changes, by using models@runtime [28] to causally connect a variability model (typically a feature model [29] or an orthogonal variability model [30]) with a behavioural model (typically architectural units). To change behaviour, they bind variation points at runtime by selecting (i.e., activating or deactivating) features that best adapt to the current context. Thus, a set of features represents a behaviour variant, which is transformed into a software architecture using a transformation mechanism [29], [31]. Undoubtedly, such a mechanism increases the overhead for changing behaviour at runtime. Furthermore, DSPL requires dynamic reconfiguration of the running composition, as they also separate variability from behaviour.

Dynamic reconfiguration includes code substitution (e.g., parametrization or pre-processor directives) [32], [33], dynamic aspect weaving [34], [29], [35], [36], [1], [36], enabling/disabling services and connectors [37], [3], and component substitution [38], [39].

C. Discussion

Parametric variability is only suitable when all variants can be defined and implemented in advance. However, IoT systems require plenty of different alternative behaviours for adapting to the ever changing context, even though they operate under closed environments. For that reason, parametric variability is inconvenient for highly dynamic IoT environments.

Remarkably, DX-MAN does not require the manual definition of alternative behaviours since an infinite number of workflow variants simultaneously exist at the conceptual level of a composite service. As it is infeasible to implement and deploy infinite workflow variants, workflows are non-deployable and executable only. Exogenous connectors are the actual deployable entities (cf., [4]) which coordinate the execution of multiple workflow variants. Thus, our approach does not operate on a single flat workflow, but on a multi-level composite where there is a workflow space (with multiple workflows) at every level of the hierarchy.

Constraints are important to filter out the workflows that a designer considers invalid under a closed environment. Hence, DX-MAN supports the definition of constraints in a similar fashion to feature models, with the difference that constraints are directly applicable to system's behaviour. DX-MAN currently supports topological sorting (for sequencers) and logical constraints (for parallelizers). We do not explain them due to space constraints.

Models@runtime separate variability and behaviour to allow an independent reasoning of these concerns. However, as scale increases and dependencies become overwhelming, the relationship between features and architectural artefacts becomes unmanageable. Hence, models@runtime face several problems when coping with dependencies. Moreover, the separation between variability and behavior requires dynamic reconfiguration to maintain a causal relationship between both dimensions. Dynamic reconfiguration is undesirable for highly dynamic IoT environments, since it takes time to decide the actions to be done, performing those actions, ensuring state consistency, checking safeness and redeploying the running composition. Remarkably, DX-MAN does not require any means to connect variability with behaviour as those dimensions are mixed in the definition of composite services, thereby avoiding the need of dynamic reconfiguration.

We previously presented a preliminary version of DX-MAN (cf. [5]). In this paper we described new semantics for supporting variability using workflow spaces. We also presented detailed examples to explain autonomicity, and a new composition connector called *exclusive selector*. Furthermore, we extended DX-MAN with capabilities for changing behaviour at runtime using MAPE-K loops.

A MAPE-K loop controls the behaviour of a composite service and is defined according to the expected goal of the managed composite. We particularly focus on the executor component which do not perform dynamic reconfiguration, but only execute a concrete workflow tree (i.e., a workflow variant) for adapting to different contexts.

Although our examples show autonomicity only in the context of IoT, DX-MAN can be used for other domains such as robotics, unmanned space or e-commerce. It is important to mention that we emphasize on the semantics of our model, rather than focusing on a particular implementation. Nevertheless, an implementation of DX-MAN is available at https://gitlab.cs.man.ac.uk/mbaxrda2/dxman.

VII. Conclusions and Future Work

This paper extended the semantics of the DX-MAN model by mixing variability with behaviour in composite services. In particular, composition connectors are variability operators that define composite workflow spaces containing an infinite number of workflow variants which represent alternative composite service behaviours. Thus, composite services define an infinite number of Turing machines at once in the design phase.

A MAPE-K manages a composite service behaviour and selects the workflow variant that best adapts to the current context. As workflows are non-deployable and executable only, the executor changes a composite service behaviour by executing the selected variant instead of dynamically reconfiguring the whole workflow. The variant is a concrete workflow tree built at runtime from an abstract workflow tree (defined at design-time). Composition connectors are the actual deployable entities which coordinate the execution of multiple workflows, thereby reusing the same deployment configuration for multiple executions.

We demonstrated the autonomic capabilities of DX-MAN using a case study in the domain of smart homes. Our results indicate that DX-MAN is a promising model for autonomic IoT systems. Nevertheless, there are some open issues.

DX-MAN currently enables control flow variability, making it suitable for actuating operations that do not require any data, e.g., switching the lights on. We plan to investigate novel

ways of incorporating data flow variability by leveraging the separation of autonomicity, control, data and computation.

DX-MAN is suitable for closed environments only where the designer understands the context in which the system is deployed. We are currently investigating novel ways to dynamically evolve a DX-MAN composition, so as to enable the emergence of new workflow spaces at runtime. Evolution is indeed another important characteristic of autonomic IoT systems, in addition to workflow variability.

REFERENCES

[1] G. H. Alférez and V. Pelechano, "Achieving autonomic Web service compositions with models at runtime," *Computers & Electrical Engineering*, vol. 63, pp. 332–352, Oct. 2017.

[2] M. L. Rosa *et al.*, "Business Process Variability Modeling: A Survey," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 2:1–2:45, Mar. 2017.

[3] H. Gomaa and M. Hussein, "Dynamic Software Reconfiguration in Software Product Families," in *Software Product-Family Engineering*, ser. Lecture Notes in Computer Science, F. J. van der Linden, Ed. Springer Berlin Heidelberg, 2004, pp. 435–444.

[4] D. Arellanes and K.-K. Lau, "Exogenous Connectors for Hierarchical Service Composition," in *IEEE SOCA*, 2017, pp. 125–132.

[5] D. Arellanes and K.-K. Lau, "Algebraic Service Composition for User-Centric IoT Applications," in *ICIOT 2018*, ser. Lect. Notes Comp. Sci. Springer Int. Pub., 2018, pp. 56–69.

[6] D. Arellanes and K.-K. Lau, "D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures," in *IEEE SC2*, 2017, pp. 283–286.

[7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[8] J. Hess *et al.*, "Poisson-driven dirt maps for efficient robot cleaning," in *2013 IEEE International Conference on Robotics and Automation*, May 2013, pp. 2245–2250.

[9] M. A. Yakoubi and M. T. Laskri, "The path planning of cleaner robot for coverage region using Genetic Algorithms," *Journal of Innovation in Digital Ecosystems*, vol. 3, no. 1, pp. 37–43, Jun. 2016.

[10] B. Dugarjav *et al.*, "Scan matching online cell decomposition for coverage path planning in an unknown environment," *Int. J. Precis. Eng. Manuf.*, vol. 14, no. 9, pp. 1551–1558, Sep. 2013.

[11] K. Kakousis *et al.*, "Optimizing the Utility Function-Based Self-adaptive Behavior of Context-Aware Systems Using User Feedback," in *On the Move to Meaningful Internet Systems: OTM 2008*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds. Springer Berlin Heidelberg, 2008, pp. 657–674.

[12] R. R. Filho and B. Porter, "Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning," *ACM Trans. Auton. Adapt. Syst.*, vol. 12, no. 3, pp. 16:1–16:25, Sep. 2017.

[13] E. Bruneton *et al.*, "The FRACTAL component model and its support in Java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[14] A. Haber *et al.*, "Hierarchical Variability Modeling for Software Architectures," in *2011 15th International Software Product Line Conference*, Aug. 2011, pp. 150–159.

[15] R. v. Ommering *et al.*, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

[16] E. M. Dashofy *et al.*, "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 2, pp. 199–245, Apr. 2005.

[17] C. Qian and K. Lau, "Enumerative Variability in Software Product Families," in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec. 2017, pp. 957–962.

[18] M. La Rosa *et al.*, "Configurable multi-perspective business process models," *Information Systems*, vol. 36, no. 2, pp. 313–340, Apr. 2011.

[19] I. Reinhartz-Berger *et al.*, "Extending the Adaptability of Reference Models," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 40, no. 5, pp. 1045–1056, Sep. 2010.

[20] A. Hallerbach *et al.*, "Capturing Variability in Business Process Models: The Provop Approach," *J. Softw. Maint. Evol.*, vol. 22, no. 6-7, pp. 519–546, Oct. 2010.

[21] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, R. Glück and M. Lowry, Eds. Springer Berlin Heidelberg, 2005, pp. 422–437.

[22] F. Gottschalk *et al.*, "Configurable workflow models," *Int. J. Coop. Info. Syst.*, vol. 17, no. 02, pp. 177–221, Jun. 2008.

[23] A. Kumar and W. Yao, "Design and management of flexible process variants using templates and rules," *Computers in Industry*, vol. 63, no. 2, pp. 112–130, Feb. 2012.

[24] R. Cognini *et al.*, "Business process flexibility - a systematic literature review with a software systems perspective," *Inf Syst Front*, vol. 20, no. 2, pp. 343–371, Apr. 2018.

[25] M. Koning *et al.*, "VxBPEL: Supporting variability for Web services in BPEL," *Information and Software Technology*, vol. 51, no. 2, pp. 258–269, Feb. 2009.

[26] K. C. Kang and a. P. Donohoe, "Feature-oriented product line engineering," *IEEE Software*, vol. 19, no. 4, pp. 58–65, Jul. 2002.

[27] S. Hallsteinsen *et al.*, "Dynamic Software Product Lines," *Computer*, vol. 41, no. 4, pp. 93–95, Apr. 2008.

[28] G. Blair *et al.*, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[29] B. Morin *et al.*, "Models@ Run.time to Support Dynamic Adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, Oct. 2009.

[30] N. Bencomo *et al.*, "Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 811–814, event-place: Leipzig, Germany.

[31] I. Schaefer *et al.*, "Delta-Oriented Programming of Software Product Lines," in *Software Product Lines: Going Beyond*, ser. Lecture Notes in Computer Science, J. Bosch and J. Lee, Eds. Springer Berlin Heidelberg, 2010, pp. 77–91.

[32] B. Morin *et al.*, "Taming Dynamically Adaptive Systems using models and aspects," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 122–132.

[33] C. Parra *et al.*, "Context Awareness for Dynamic Service-oriented Product Lines," in *Proceedings of the 13th International Software Product Line Conference*, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 131–140, event-place: San Francisco, California, USA.

[34] G. H. Alférez *et al.*, "Dynamic adaptation of service compositions with variability models," *Journal of Systems and Software*, vol. 91, pp. 24–47, May 2014.

[35] L. Baresi *et al.*, "Service-Oriented Dynamic Software Product Lines," *Computer*, vol. 45, no. 10, pp. 42–48, Oct. 2012.

[36] F. Fleurey *et al.*, "A Generic Approach for Automatic Model Composition," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, H. Giese, Ed. Springer Berlin Heidelberg, 2008, pp. 7–15.

[37] C. Cetina *et al.*, "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes," *Computer*, vol. 42, no. 10, pp. 37–43, Oct. 2009.

[38] J. Floch *et al.*, "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, pp. 62–70, Mar. 2006.

[39] J. White *et al.*, "Creating self-healing service compositions with feature models and microrebooting," *International Journal of Business Process Integration and Management*, vol. 4, no. 1, p. 35, 2009.

4.5 Decentralized Data Flows in Algebraic Service Compositions for the Scalability of IoT Systems

Damian Arellanes and Kung-Kiu Lau

In proceedings of the **5th World Forum on Internet of Things (WF-IoT 2019)**.

Published by IEEE,
pages 668-673,
ISBN 978-1-5386-4980-0,
2019.

Impact Indicators:
• Conference Qualis ranking (2016): B3,
• Impact factor (2017): 3.07.

The final authenticated version [AL19a] is available online at `https://doi.org/10.1109/WF-IoT.2019.8767238`.

Summary: This paper describes the data flow dimension of DX-MAN and proposes an approach that leverages the *separation of control and data* to realise *decentralised data flows*. The approach is validated on top of the DX-MAN platform using the Blockchain as the underlying data space, and an empirical evaluation is done by comparing decentralised data flows versus distributed data flows.

Comments on authorship: I proposed the main idea of the paper, developed and validated the main approach, conducted an empirical evaluation, analysed results, investigated related work, provided and edited all graphics, participated in the entire writing process and addressed the reviewer's comments. My supervisor, Kung-Kiu Lau, proofread the paper and approved the results. He also guided the whole research process.

Key contributions: Contribution 4.2 (see Section 1.5).

Decentralized Data Flows in Algebraic Service Compositions for the Scalability of IoT Systems

Damian Arellanes and Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

***Abstract*—With the advent of the Internet of Things (IoT), scalability becomes a significant concern due to the huge amounts of data generated in IoT systems. A centralized data exchange is not desirable as it leads to a single performance bottleneck. Although a distributed data exchange removes the central bottleneck, it has network performance issues as data passes among multiple coordinators. A decentralized approach is the only solution that fully enables the realization of efficient IoT systems, since there is no single performance bottleneck and network overhead is minimized. In this paper, we present an approach that leverages the semantics of DX-MAN for realizing decentralized data flows in IoT systems. The algebraic semantics of such a model allows a well-defined structure of data flows which is easily analyzed by an algorithm that forms a direct relationship between data consumers and data producers. For the analysis, the algorithm takes advantage of the fact that DX-MAN separates control flow and data flow. Thus, our approach prevents passing data alongside control among multiple coordinators, so data is only read and written on a decentralized data space. We validate our approach using smart contracts on the Blockchain, and conducted experiments to quantitatively evaluate scalability. The results show that our approach scales well with the size of IoT systems.**

***Index Terms*—Internet of Things, decentralized data flows, Blockchain, DX-MAN, exogenous connectors, scalability, separation between control and data, algebraic service composition**

I. Introduction

The Internet of Things (IoT) envisions a world where everything will be interconnected through distributed services. As new challenges are forthcoming, this paradigm requires a shift in our way of building software systems. With the rapid advancement in hardware, the number of connected *things* is increasing considerably, to the extent that scalability becomes a significant concern due to the huge amounts of data involved in IoT systems. Thus, IoT services must exchange data over the Internet efficiently.

Although a centralized data exchange approach has been successful in enterprise systems, it will easily cause a bottleneck in IoT systems which potentially generate a huge amount of data continuously. To avoid the bottleneck, a distributed approach can be used to distribute the load of data over multiple coordinators. However, this would introduce unnecessary network overhead as data is passed among many coordinators.

A decentralized data exchange approach is the most efficient solution to tackle the imminent scale of IoT systems, as it achieves better response time and throughput by minimizing network hops [1], [2], [3], [4], [5], [6], [7]. However, exchanging data among loosely-coupled IoT services is challenging, especially in resource-constrained environments where *things* have poor network connection and low disk space.

Moreover, constructing data dependency graphs is not trivial when control flow and data flow are tightly coupled. The separation between control and data is necessary because it allows a separate reasoning, monitoring, maintenance and evolution of these concerns [8]. Consequently, an efficient data exchange approach can be done without considering control flow, thereby reducing the messages sent over the Internet.

This paper proposes an approach that leverages the semantics of DX-MAN [9], [10] for the realization of decentralized data flows in IoT systems. The algebraic semantics of such a model allows a well-defined structure of data flows which is easily analyzed by an algorithm that forms a direct relationship between data consumers and data producers. For the analysis, the algorithm particularly takes advantage of the fact that DX-MAN separates control flow and data flow.

The rest of the paper is organized as follows. Sect. II introduces the semantics of the DX-MAN model. Sect. III describes DX-MAN data flows and Sect. IV presents the algorithm that analyzes such data flows. Sect. V presents the implementation of our approach. Sect. VI outlines a quantitative evaluation of our approach. Finally, we present the related work in Sect. VII and conclusions in Sect. VIII.

II. DX-MAN Model

DX-MAN is an algebraic model for IoT systems where services and exogenous connectors are first-class entities. An *exogenous connector* is a variability operator that defines multiple workflows with explicit control flow, while a DX-MAN *service* is a distributed software unit that exposes a set of operations through a well-defined interface.

An *atomic service* provides a set of operations and it is formed by connecting an *invocation connector* with a *computation unit*. A computation unit represents an actual service implementation (e.g., a RESTful Microservice or a WS-* service) and it is not allowed to call other computation units. As a consequence of the algebraic semantics, the interface of an atomic service has all the operations in the computation unit, as shown by the red arrows in Fig. 1(a). An invocation

connector defines the most primitive workflow which is the invocation of one operation in the computation unit.

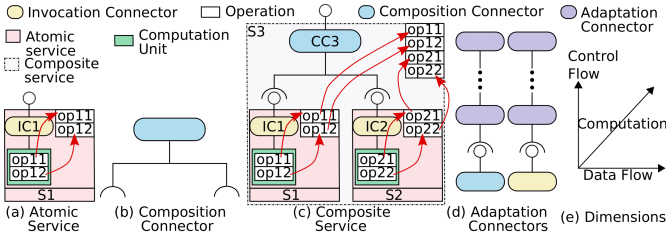


Fig. 1. DX-MAN Model.

Our notion of algebraic service composition is inspired by algebra where functions are hierarchically composed into a new function of the same type. The resulting function can be further composed with other functions so as to yield a more complex one. *Algebraic service composition* is the operation by which a composition connector is used as an operator to compose multiple services, resulting in a (hierarchical) *composite service* whose interface has all the sub-service operations. Thus, a top-level composite will always contain the operations of all the atomic services (Fig. 1(c)). In particular, there are composition connectors for sequencing, branching and parallelism. A *sequencer connector* enables ∞ workflows for the sequential invocation of sub-service operations. A *selector connector* defines $2^n - 1$ branching workflows and chooses the sub-service operations to invoke, such that n is the number of operations in the composite service interface. A *parallel connector* defines ∞ parallel workflows and executes sub-service operations in parallel according to user-defined tasks.

Fig. 1(d) shows that an adapter can be connected with only one exogenous connector. A looping adapter iterates over a sub-workflow while a condition holds true, and a guard adapter invokes a sub-workflow whenever a condition holds true. There are also adapters for sequencing, branching and parallelism which act over the operations of an individual atomic service.

Fig. 1(e) shows that data, control and computation are orthogonal dimensions in DX-MAN. Exogenous connectors enable the separation between control flow and computation, since they decouple service implementations from the (hierarchical) composition structure. Unlike existing composition approaches, data flow never follows control flow as exogenous connectors only pass control to coordinate workflow executions. For further details about the control flow dimension, we refer the reader to our previous papers [9], [10].

III. Data Connectors

A DX-MAN operation is a set of input parameters and output parameters. An input parameter defines the required data to perform a computation, while an output parameter is the data resulting from a specific computation. Although exogenous connectors do not provide any operation (because they do not perform any computation), some of them require data. In particular, selector connectors, selector adapters, looping adapters and guard adapters require input values to evaluate boolean conditions. Exogenous connectors do not have any parameters by default, but designers manually define parameters for a chosen workflow. Workflow selection is out of the scope of this paper, but we refer the reader to our previous paper on workflow variability [10].

In addition to the operations created on algebraic composition, custom operations can be defined in composite services. This is particularly useful when designers want to create a unified composite service interface, in order to hide operations created during algebraic composition.

A *data connector* defines an explicit data flow by connecting a source parameter with a destination parameter. Fig. 2 shows that an *algebraic data connector* is automatically created during composition and is available for all the workflows defined by a composite service. In particular, an algebraic data connector connects two parameters vertically, i.e., bottom-up for outputs and top-down for inputs. Fig. 3 shows the data connection rules for our approach, where we can see that algebraic data connectors can be defined in four different ways only.

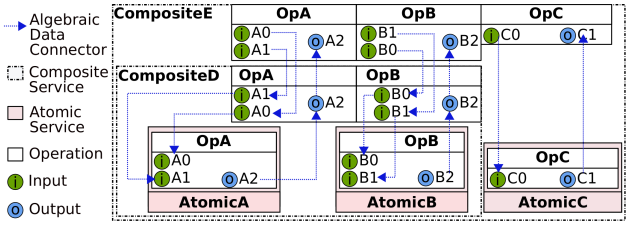


Fig. 2. Algebraic data channels.

Fig. 2 shows that composite services encapsulate data flows to ensure reusability. So, composite services are (black boxes) unaware of data flows of other composites. Hence, there are no data connections between parameters in different composite services, but only data connectors within a composite.

A *custom data connector* is manually created for only one workflow, which connects two parameters either vertically or horizontally. Fig. 3 shows that designers can define custom data connectors in 16 different ways.

Currently, DX-MAN supports custom data connectors for the most common data patterns, namely sequencing and map-reduce. For the sequencing pattern, the parameters of two different operations are horizontally connected. Fig. 4 shows an example of this pattern, where operation *OpB* requires data from operation *OpA*. In particular, a custom data connector links the output *A0* with the input *B0*, while another custom data connector connects the output *A1* with the input *B1*. To improve readability, we omit algebraic data connectors.

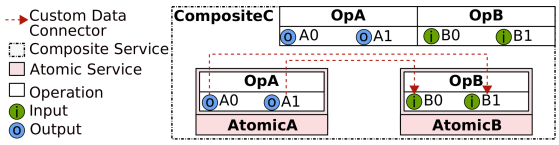


Fig. 4. An example of two sequential data flows.

A *data processor* is particularly useful when data pre-processing needs to be done before executing an operation. It waits until all input values have been received, then performs some computation and returns transformed data in the form of outputs. A *mapper* executes a user-defined function on each

	from\to	Operation in sub-atomic		Operation in sub-composite		Composition connector		Adapter		Operation in the composite		Data processor	
		in	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
Operation in sub-atomic	*in*	✕	✕	✕	✕	✕	*N/A*	✕	*N/A*	✕	✕	✕	✕
	out	**Cust.**	✕	**Cust.**	✕	✕	*N/A*	**Cust.**	*N/A*	✕	**Alge.**	**Cust.**	✕
Operation in sub-composite	*in*	✕	✕	✕	✕	✕	*N/A*	✕	*N/A*	✕	✕	✕	✕
	out	**Cust.**	✕	**Cust.**	✕	✕	*N/A*	**Cust.**	*N/A*	✕	**Alge.**	**Cust.**	✕
Composition connector	*in*	✕	✕	✕	✕	✕	*N/A*	✕	*N/A*	✕	✕	✕	✕
	out	*N/A*											
Adapter	*in*	✕	✕	✕	✕	✕	*N/A*	✕	*N/A*	✕	✕	✕	✕
	out	*N/A*											
Operation in the composite	*in*	**Alge.**	✕	**Alge.**	✕	**Cust.**	*N/A*	**Cust.**	*N/A*	✕	✕	**Cust.**	✕
	out	✕	✕	✕	✕	✕	✕	✕	*N/A*	✕	✕	✕	✕
Data processor	*in*	✕	✕	✕	✕	✕	*N/A*	✕	*N/A*	✕	✕	✕	✕
	out	**Cust.**	✕	**Cust.**	✕	✕	*N/A*	**Cust.**	*N/A*	✕	**Cust.**	**Cust.**	✕

Alge. Algebraic Data Connection

Cust. Custom Data Connection

✕ No Data Connection

N/A No Applicable

Fig. 3. Data connection rules.

input value received and, similarly, a *reducer* takes the result from a mapper and executes a user-defined reduce function on inputs. A *reducer* can also be used in isolation to perform straightforward computation such as combining data into a list. Fig. 5 shows an example of the map-reduce pattern, where operation *opB* requires the pre-processing of data generated by operation *opA*. In particular, two custom data connectors link the input *A0* and the output *A1* with the inputs of the mapper. The output of the mapper is connected to the input of the reducer and, similarly, the output of the reducer is connected to the input *B0*. Note that *A0* can only be connected from the composite service operation, according to Fig. 3.

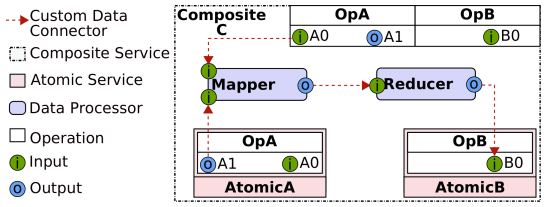


Fig. 5. An example of the map-reduce pattern.

In some workflows, algebraic data connectors may not be needed. For that reason, such connectors can be removed manually at the discretion of the designer. For example, in Fig. 5 all algebraic connectors were removed because data is only needed for the realization of the map-reduce pattern.

IV. Analysis of Data Connectors

Algebraic service composition and the separation of concerns are key enablers of decentralized data flows. In particular, exogenous connectors provide a hierarchical control flow structure that is completely separated from the data flow structure enabled by data connectors. Thus, the data connections in a composite service form a well-structured data dependency graph that is easily analyzed at deployment-time by means of Algorithm 1. To understand this algorithm, some formal definitions are necessary.

Let $\mathbb{D}$ be the data type, $\mathbb{PD}$ the processor parameter type, $\mathbb{OD}$ the operation parameter type and $\mathbb{CD}$ the type of exogenous connector inputs, such that $\mathbb{PD}, \mathbb{OD}, \mathbb{CD} \subseteq \mathbb{D}$. A data connector is then a tuple of type $\mathbb{DC} : \mathbb{D} \times \mathbb{D}$ that connects a $source \in \mathbb{D}$ parameter with a $destination \in \mathbb{D}$ parameter.

Reader parameters are the entities that directly consume data produced by *writer parameters*. I_r is the set of inputs that read data during a workflow execution, namely the inputs of atomic service operations, the inputs of exogenous connectors and the inputs of data processors. O_r is the set of operation outputs in the top-level composite, useful for reading data resulting from a workflow execution. The set I_w represents the required data for a workflow execution, which are the inputs of operations in the top-level composite. O_w is the set of outputs that write data during a workflow execution, namely the outputs of atomic service operations and the outputs of data processors.

Algorithm 1 Algorithm for the analysis of data connectors

1: **procedure** ANALYZE(dc, R, W) ▷ The types of R and W are defined in the text below, and $dc \in \mathbb{DC}$
2: $X_w \leftarrow \emptyset$ ▷ $X_w = \{x \mid x \in \mathbb{D}\}$
3: $Y_r \leftarrow \emptyset$ ▷ $Y_r = \{y \mid y \in \mathbb{D}\}$
4: **if** $\Pi_1(dc) \notin \mathbb{PD} \wedge \Pi_1(dc) \in dom(R)$ **then**
5: $X_w \leftarrow R(\Pi_1(dc))$
6: **else**
7: $X_w \leftarrow \{\Pi_1(dc)\})$
8: **if** $\Pi_2(dc) \notin \mathbb{PD} \wedge \Pi_2(dc) \in dom(W)$ **then**
9: $Y_r \leftarrow W(\Pi_2(dc))$
10: **for each** $y \in Y_r$ **do**
11: $R \oplus \{y \mapsto R(y) - \{\Pi_2(dc)\} \cup X_w\}$
12: **else**
13: $Y_r \leftarrow \{\Pi_2(dc)\})$
14: $R \oplus \{\Pi_2(dc) \mapsto R(\Pi_2(dc)) \cup X_w\}$
15: **for each** $x \in X_w$ **do**
16: $W \oplus \{x \mapsto W(x) \cup Y_r\}$

Basically, Algorithm 1 analyzes data connectors for all composite services, using a bottom-up approach. The goal of this algorithm is to create a relationship between reader parameters and writer parameters, while ignoring those parameters that do not need to manipulate data. To do so, Algorithm 1 receives a data connector $dc \in \mathbb{DC}$ as an input, and uses $R \in ((I_r \cup O_r) \mapsto \{w \mid w \subset I_w \cup O_w\})$ for mapping a reader parameter to a set of writer parameters and $W \in ((I_w \cup O_w) \mapsto \{r \mid r \subset I_r \cup O_r\})$ for mapping a writer parameter to a set of reader parameters.

Algorithm 1 creates two empty sets X_w and Y_r for analyzing the endpoints of a data connector dc. X_w is the set of parameters connected to the *source parameter* $\Pi_1(dc)$ iff $\Pi_1(dc)$ is not a data processor parameter and has incoming data connectors; otherwise, X_w only contains $\Pi_1(dc)$. Similarly, if the *destination parameter* $\Pi_2(dc)$ is not a data processor parameter and $\Pi_2(dc)$ has outgoing data connectors, then Y_r is

the set of parameters connected from $\Pi_2(dc)$ and X_w (without $\Pi_2(dc)$) is added into the writers of each element $y \in Y_r$; otherwise, Y_r only contains $\Pi_2(dc)$ and X_w is added into the writers of $\Pi_2(dc)$. Finally, the set Y_r is added into the readers of each element $x \in X_w$. The result of the algorithm is thus a mapping of reader parameters to writer parameters.

V. Implementation

We implemented our approach on top of the DX-MAN Platform [11], and we used the Blockchain as the underlying data space for persisting parameter values, and for leveraging the capabilities provided by these decentralized platforms, such as performance, security and auditability. Furthermore, using the Blockchain ensures that every service is the owner of its own data, while data provenance is provided to discover data flows (i.e., how data is moved between services) or how parameters change over time. In particular, we defined three smart contracts using *Hyperledger Composer 0.20.0* for executing transactions on *Hyperledger Fabric 1.2*. We do not show the source code due to space constraints, but it is available at https://gitlab.cs.man.ac.uk/mbaxrda2/dxman/tree/development.

The DX-MAN platform provides an API to support the three phases of a DX-MAN system lifecycle: design-time, deployment-time and run-time. Composite service templates only contain algebraic data connectors, as they represent a general design with multiple workflows. Using API constructs, a designer chooses a workflow and defines custom data connectors (and perhaps data processors) for each composite service involved. Similarly, data processor functions are defined by designers using API constructs.

At deployment-time, Algorithm 1 analyzes data connectors (defined at design-time), in order to construct a Java HashMap for readers where the keys are reader parameter UUIDs and the values are lists of writer parameter UUIDs. After getting the map for a given workflow, reader parameters (with their respective list of writers) are stored as assets in the Blockchain by means of the transaction *CreateParameters*.

At run-time, exogenous connectors coordinate a workflow execution by passing control via CoAP messages. When control reaches an invocation connector, the five steps illustrated in Fig. 6 are performed. Although the rest of exogenous connectors behave similarly, they only perform the first two steps. First, the invocation connector uses the transaction *readParameters* to read input values from the Blockchain. For each input, the Blockchain reads values directly from the writers list. As there might be multiple writer parameters, this transaction returns a list of the most recent values that were updated during the workflow execution. Hence, a timestamp is set whenever a parameter value is updated. Output values are written onto the data space as soon as they are available, even before control reaches data consumers. Thus, having concurrent connectors (e.g., a parallel connector) may lead to synchronization issues during workflow execution. To solve this, control flow blocks in the invocation connector until all input values are read.

Once all inputs are ready, the invocation connector executes the implementation of an operation by passing the respective input values. Then, the operation performs some computation and returns a result in the form of outputs. Finally, the invocation connector writes the output values onto the Blockchain using the transaction *updateParameters*.

Fig. 6. Steps for the invocation of an operation implementation.

An *UpdateParameterEvent* is published whenever a parameter value is updated. At deployment-time, the DX-MAN platform subscribes data processor instances to the events produced by the respective writer parameters. Thus, a data processor instance waits until it receives all events, before performing its respective designer-defined computation. Although our current implementation supports only *mappers* and *reducers*, further data processors can be introduced using the semantics presented in Sect. III, e.g., a *shuffler* can be added to sort data by key.

Our approach enables transparent data exchange as data routing is embodied in the Blockchain. Thus, reader parameters are not aware where the data comes from, and writer parameters do not know who reads the data they produce. Furthermore, the map generated by Algorithm 1 avoids the inefficient approach of passing values through data connectors during workflow execution. Thus, exogenous connectors and data processors read data directly from parameters which only write values onto the Blockchain. Undoubtedly, this enables a transparent decentralized data exchange.

VI. Evaluation

In this section, we present a comparative evaluation between distributed data flows and decentralized data flows for a DX-MAN composition. In the former approach, data is passed over the network through data connectors, whereas the second approach is our proposal. Our evaluation intends to answer two major research questions: (A) Does the approach scale with the number of data connectors? and (B) Under which conditions is decentralized data exchange beneficial?

As a DX-MAN composition has a multi-level hierarchical structure, an algebraic data connector passes a data value vertically in a bottom-up way (for inputs) or in a top-down fashion (for outputs) while a custom data connector passes values horizontally or vertically. For our evaluation, we only consider vertical routing through algebraic data connectors.

$M_p = \{\lambda_j | \lambda_j \in \mathbb{R}\}$ is the set of network message costs for vertically routing the value of a parameter p, where λ_j is the cost of passing a value for p through an algebraic data connector j. Likewise, Γ_p and ω_p are the costs of reading and writing a value for p on the data space, respectively.

Equations 1 and 2 calculate the total message cost of routing a value p using a distributed approach s.t. α_p is for input values and β_p is for output values. Remarkably, as the decentralized approach does not pass values through data connectors, its total message cost of routing the value of p is Γ_p for inputs, and ω_p for outputs.

$$\alpha_p(\Gamma_p, M_p) = \Gamma_p + \sum_{j=0}^{|M_p|-1} (\lambda_j \in M_p) \quad (1)$$

$$\beta_p(\omega_p, M_p) = \omega_p + \sum_{j=0}^{|M_p|-1} (\lambda_j \in M_p) \quad (2)$$

Fig. 7 depicts the DX-MAN composition that we consider for our evaluation, which has three levels, three atomic services and two composite services. The composites *ServiceD* and *ServiceE* have three and five data connectors, respectively. Fig. 7 shows that a data connector has a cost $\lambda_{j\in[0,7]}$ of passing a value over the network. Then, the vertical routing sets for the parameters are $M_{A0} = \{\lambda_3\}$, $M_{A1} = \{\lambda_4\}$, $M_{B0} = \{\lambda_0, \lambda_5\}$, $M_{B1} = \{\lambda_1, \lambda_6\}$ and $M_{C0} = \{\lambda_2, \lambda_7\}$.

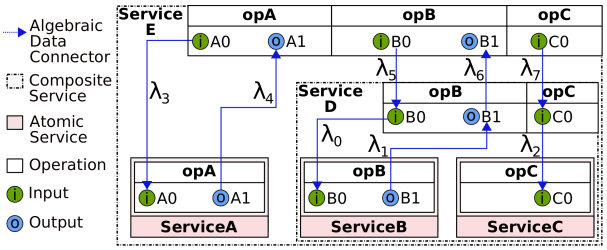


Fig. 7. DX-MAN composition for the evaluation of our approach.

For clarity, we assume that the DX-MAN composition interacts with an external application via a shared data space. So, we can ignore the cost of passing data between the application and the composition. The costs of reading the inputs *A0*, *B0* and *C0* are Γ_{A0}, Γ_{B0} and Γ_{C0}, respectively, and the costs of writing the outputs *A1* and *B1* are ω_{A1} and ω_{B1}, respectively.

Suppose that a specific workflow requires the invocation of the operations *opA* and *opC*. Using a distributed exchange would require passing and reading values for two inputs, and returning and writing one output value. Therefore, the total message cost would be $\alpha_{A0} + \beta_{A1} + \alpha_{C0} = \lambda_3 + \lambda_4 + \lambda_2 + \lambda_7 + \Gamma_{A0} + \omega_{A1} + \Gamma_{C0}$. Remarkably, the total message cost using the decentralized exchange would be $\Gamma_{A0} + \omega_{A1} + \Gamma_{C0}$.

A. RQ1: Does the approach scale with the number of data connectors?

We conducted an experiment that dynamically increases the number of data connectors of the DX-MAN composition depicted in Fig. 7. The experiment is carried out in 100000 steps with $\Gamma_{A0} = \omega_{A1} = \Gamma_{B0} = \omega_{B1} = \Gamma_{C0} = 1$.

For each step of the experiment, we add a new parameter in a random atomic operation. As a consequence of algebraic composition, another parameter is added in the respective composite operation and a data connector links these parameters.

In this experiment, we compare the cost of the distributed exchange vs. the cost of the decentralized exchange. Rather than computing the costs for the invocation of specific operations, we compute the total costs for the DX-MAN composition using $\Gamma_{A0} + \omega_{A1} + \Gamma_{B0} + \omega_{B1} + \Gamma_{C0} + \sum_{j=0}^{7} \lambda_j$. Fig. 8 shows that the costs grow linearly with the number of data connectors, and that the decentralized approach outperforms its counterpart by reducing costs by a factor of 2.67 in average.

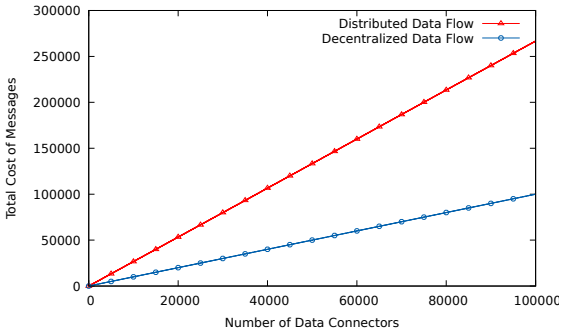


Fig. 8. Impact of increasing the number of data connectors in a DX-MAN composition.

B. RQ2: Under which conditions is decentralized data exchange beneficial?

We conducted an experiment of 100000 steps to see the benefit of the decentralized approach as the number of levels of the composition increases. We particularly consider the total costs for the input *A0* and we assume that $\Gamma_{A0} = 1$. At each step, the number of levels is increased by 1 and $\sum_{j=0}^{|M_{A0}|-1} \lambda_j$ by 0.0004. Thus, increasing the sum of vertical costs means that $\frac{\sum_{j=0}^{|M_{A0}|-1} \lambda_j}{|M_{A0}|} = 1$ and increasing the number of levels by 1 means that $|M_{A0}|$ is also increased by 1. The improvement rate of the decentralized data exchange is $1 - \frac{\Gamma_{A0}}{(\Gamma_{A0} + \sum_{j=0}^{|M_{A0}|-1} \lambda_j)}$.

Fig. 9 shows the results of this experiment, where it is clear that the benefit of the decentralized approach becomes more evident as the number of levels of the composition increases. This is because the number of data connectors increases with the number of levels and so the cost of the distributed approach. The only way a distributed approach would outperform the decentralized one is when the cost of performing operations on the data space is more expensive than the total cost of passing values vertically. In particular, for our experiment the DX-MAN composition would get a benefit only if $\Gamma_{A0} < \sum_{j=0}^{|M_{A0}|-1} \lambda_j$.

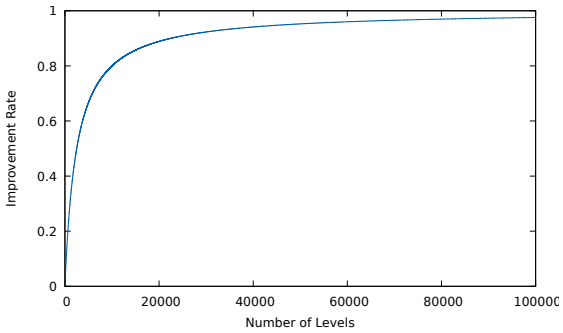


Fig. 9. Impact of increasing the number of levels in a DX-MAN composition.

VII. Related Work

This section presents the related work on decentralized data flows in service-oriented systems. We classified our findings into three categories, depending on the composition semantics the approaches are built on: orchestration (with central control flows and decentralized data flows), decentralized orchestration, data flows and choreographies.

Orchestration approaches [12], [1] partially separate data from control so as to enable P2P data exchanges. To do so, an orchestrator coordinates the exchanges by passing data references alongside control. Thus, extra network traffic is introduced as data references (and acknowledge messages) are transferred over the network. These approaches are typically based on proxies that keep data, thus causing an issue for *things* with low disk space. By contrast, DX-MAN does not require any coordinator for data exchange, and exogenous connectors do not store data. Besides, exogenous connectors do not exchange references, thanks to the separation of concerns.

Only few approaches discuss data decentralization using the semantics of decentralized orchestration [5], [7]. [13] stores data and control in distributed tuple spaces which may become a bottleneck in IoT environments continuously generating huge amounts of data. [3] stores references instead of values, but references are still needed because data is mixed with control. Moreover, [3] requires the maintenance of tuple spaces (for passing references), and databases (for storing data). In DX-MAN, references and values coexist in the same data space.

Although exogenous data flows [14] allocate flows over different *things*, there is a master engine that coordinates data exchange for slave engines. Hence, this approach introduces extra network hops as data is passed among multiple engines. Service Invocation Triggers [2] use endogenous data flows to exchange data directly, but they rely on workflows that do not contain loops and conditionals. This limitation arises from the fact that it is not trivial to analyze data dependencies when control is mixed with data. In general, endogenous data flows [15], [16] do not support explicit control flow which is a crucial requirement for the scalability of IoT systems [8].

A choreography describes interactions among participants using decentralized message exchanges (a.k.a. conversations). Workflow participants [17] pass data among multiple engines leading to network degradation. Although services may exchange data directly by message passing, they are not reusable because data and control are mixed [8]. [4] uses peers to separate control from computation; however, peers pass data alongside control according to predefined conversations, leading to the issues discussed in [6]. Although [18] proposes the separation of control and data for choreographies, it uses a middleware which may potentially become a central bottleneck.

VIII. Conclusions

In this paper, we presented an approach on top of DX-MAN for decentralized data flows in IoT systems. At design-time, the algebraic semantics of DX-MAN enables a well-defined structure of data connections. As data connections are not mixed with control flow structures, an algorithm easily analyzes data connections at deployment-time. The result is a mapping between reader parameters and writer parameters, which prevents passing values through data connectors. In our current implementation, the Blockchain embodies this mapping to manage data values at run-time.

DX-MAN is a service model that separates data flow, control flow and computation, for a separate reasoning, monitoring, maintenance and evolution of such concerns. Separating data and control particularly prevents exogenous connectors from passing data alongside control, and allows the use of different technologies to handle data flows and control flows separately.

Our experiments confirm that our approach scales well with the number of data connectors and with the number of levels of a DX-MAN composition. They also suggest that our approach provides the best performance when the cost of performing operations on the data space is less than the total cost of passing data over the network. Thus, our approach is extremely beneficial for IoT systems consisting of plenty of services.

Acknowledgment

This research is sponsored by the National Council of Science and Technology (CONACyT).

References

[1] A. Barker *et al.*, "Reducing Data Transfer in Service-Oriented Architectures: The Circulate Approach," *IEEE Trans. Serv. Comput.*, vol. 5, no. 3, pp. 437–449, 2012.

[2] W. Binder *et al.*, "Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration," *Int. J. High Perf. Comp. and Net.*, vol. 6, no. 1, pp. 81–90, 2009.

[3] M. Sonntag *et al.*, "Process space-based scientific workflow enactment," *Int. J. Business Proc. Integr. and Man.*, vol. 5, no. 1, pp. 32–44, 2010.

[4] A. Barker *et al.*, "Choreographing Web Services," *IEEE Trans. Serv. Comput.*, vol. 2, no. 2, pp. 152–166, 2009.

[5] M. Pantazoglou *et al.*, "Decentralized Enactment of BPEL Processes," *IEEE Trans. Serv. Comput.*, vol. 7, no. 2, pp. 184–197, 2014.

[6] M. Hahn *et al.*, "Data-Aware Service Choreographies Through Transparent Data Exchange," in *Web Eng.*, ser. Lect. Notes Comp. Sci. Springer Int. Pub., 2016, pp. 357–364.

[7] W. Jaradat *et al.*, "Towards an autonomous decentralized orchestration system," *Concurr. Comp. Pract E.*, vol. 28, no. 11, pp. 3164–3179, 2016.

[8] D. Arellanes and K.-K. Lau, "Analysis and Classification of Service Interactions for the Scalability of the Internet of Things," in *IEEE ICIOT*, 2018, pp. 80–87.

[9] D. Arellanes and K.-K. Lau, "Exogenous Connectors for Hierarchical Service Composition," in *IEEE SOCA*, 2017, pp. 125–132.

[10] D. Arellanes and K.-K. Lau, "Algebraic Service Composition for User-Centric IoT Applications," in *ICIOT 2018*, ser. Lect. Notes Comp. Sci. Springer Int. Pub., 2018, pp. 56–69.

[11] D. Arellanes and K.-K. Lau, "D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures," in *IEEE SC2*, 2017, pp. 283–286.

[12] D. Liu, "Data-flow Distribution in FICAS Service Composition Infrastructure," 2002.

[13] D. Wutke *et al.*, "Model and Infrastructure for Decentralized Workflow Enactment," in *Proc. Symp. on Appl. Comp.* ACM, 2008, pp. 90–94.

[14] N. K. Giang *et al.*, "Developing IoT applications in the Fog: A Distributed Dataflow approach," in *IOT*, 2015, pp. 155–162.

[15] J. Seeger *et al.*, "Running Distributed and Dynamic IoT Choreographies," in *IEEE GIoTS*, 2018, pp. 1–6.

[16] S. Cherrier *et al.*, "D-LITe: Distributed Logic for Internet of Things Services," in *IEEE iThings/CPSCom*, 2011, pp. 16–24.

[17] G. Decker *et al.*, "BPEL4chor: Extending BPEL for Modeling Choreographies," in *IEEE ICWS*, 2007, pp. 296–303.

[18] M. Hahn *et al.*, "TraDE - A Transparent Data Exchange Middleware for Service Choreographies," in *On the Move to Meaningful Internet Syst.*, ser. Lect. Notes Comp. Sci. Springer Int. Pub., 2017, pp. 252–270.

4.6 Analysis and Classification of Service Interactions for the Scalability of the Internet of Things

Damian Arellanes and Kung-Kiu Lau

In proceedings of the **2nd International Congress on Internet of Things (ICIOT 2018)**.

Published by IEEE,
pages 80-87,
ISBN 978-1-5386-7244-0,
2018.

Impact Indicators:
• Conference Core rank (2018): B,
• Acceptance rate (2019): 18.60%,
• Impact factor (2018): 1.71,
• Third most cited article amongst 28 publications in ICIOT 2018 (as of 2019),
• Runner up for the **Carole Goble medal** for outstanding doctoral paper in the Department of Computer Science, University of Manchester.

The final authenticated version [AL18b] is available online at `https://doi.org/10.1109/ICIOT.2018.00018`.

Summary: This paper classifies and analyses IoT service interactions into four schemas which are described using a case study in the domain of smart cities and evaluated against *explicit control flow*, *separation of control and computation*, *distributed workflows* and *location transparency*.

Comments on authorship: I proposed the main idea of the paper, investigated service interactions, conducted a qualitative comparison, analysed results, provided and edited all graphics, participated in the entire writing process and addressed the reviewer's comments. My supervisor, Kung-Kiu Lau, also contributed to the idea, proofread the paper and approved the results. He also guided the whole research process.

Key contributions: Contribution 2 and Contribution 3 (see Section 1.5).

Analysis and Classification of Service Interactions for the Scalability of the Internet of Things

Damian Arellanes and Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

***Abstract*—Scalability is an important concern for Internet of Things (IoT) applications since the amount of service interactions may become overwhelming, due to the huge number of interconnected nodes. In this paper, we present an IoT scenario for real-time Electrocardiogram (ECG) monitoring, in order to analyze how well different kinds of service interactions can fulfill the scalability requirements of IoT applications.**

***Index Terms*—Internet of Things (IoT), service interactions, scalability, Internet of Services (IoS)**

I. Introduction

The Internet of Things (IoT) promises a new era in which not only people interact through Internet, but so do things. Currently, the number of connected devices worldwide is about 17 billion, and it is estimated that this number will grow by a factor of 1.82 in the next three years [1]. For this reason, scalability in terms of the size of IoT applications, rather than vertical or horizontal scalability [2], is an important concern.

For this kind of scalability, four crucial desiderata has been identified: explicit control flow [3], separation between control and computation [4], decentralization [5] and location transparency [6]. In this paper, we analyze how well different kinds of service interactions can fulfill these scalability requirements.

Service interactions play a central role in the Internet of Services (IoS) [7] which will be a key enabler of the IoT goals. IoT services interact in different ways to achieve a common goal in a specific application. Despite an increasing number of proposed network protocols for IoT, there is a lack of understanding about which service interactions best fulfill the scalability requirements of IoT applications.

The rest of the paper is structured as follows. Sect. II presents an IoT scenario for real-time Electrocardiogram (ECG) monitoring. Sect. III describes our classification of service interactions. Sect. IV presents the results of our analysis. Sect. V presents a discussion of our results. Finally, Sect. VI presents the conclusions and the future work.

II. IoT Scenario: Electrocardiogram Monitoring Network (EMoNet)

This section introduces a running example for the rest of the paper. The example is an IoT scenario for real-time Electrocardiogram (ECG) monitoring: Electrocardiogram Monitoring Network (EMoNet). EMoNet is a network deployed in a smart city, consisting of patients with cardiac diseases, plenty of ambulances moving around the city, patients' smartwatches and wearable ECG sensors. Fig. 1 depicts the workflow of EMoNet which corresponds to a timing task triggered every 3 minutes for a particular patient. It basically consists of pulling and analyzing ECG data, and requesting the nearest ambulance if the patient has heart attack signs.

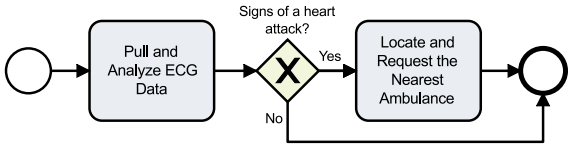


Fig. 1. EMoNet workflow.

The EMoNet workflow involves four independent IoT nodes shown in Fig. 2: a wearable ECG sensor installed on a patient's chest, the patient's smartwatch, a healthcare cloud and an ambulance.

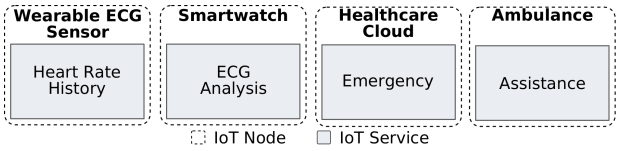


Fig. 2. Nodes and services involved in the EMoNet workflow.

The wearable ECG sensor provides the *Heart Rate History* service as an interface for the records of the electrical activity of a patient's heart. The smartwatch provides the *ECG Analysis* service that determines if a patient is showing signs of a heart attack. A healthcare cloud provides the *Emergency* service to find the nearest ambulance and request it immediately. Ambulances provide the *Assistance* service to attend to those patients in need on-site. For simplicity, we assume that these services dispatch many requests concurrently. In the next section, we will describe different ways to realize the EMoNet workflow using these services.

III. Service Interaction Schemas

IoT services provide low-level functionality implemented in nodes [8]. Resource-constrained nodes (e.g., a pulse sensor) provide fine-grained services for basic functionality (e.g., fetching sensor data). Non resource-constrained nodes (e.g., a smart TV) may offer coarse-grained services in addition.

IoT services interact via a network in order to realize complex functionality. Services can interact by message passing, event exchanges, or any combination thereof. In order to determine what kind of interactions best fulfills the scalability requirements of IoT, we have classified service interactions into four schemas: (i) direct service interactions, (ii) indirect service interactions, (iii) exogenous service interactions and (iv) event-driven service interactions.

Schemas (i), (ii) and (iii) are based on message-passing, where there are two roles: service sender and service receiver. A service sender accesses functionality offered by a service receiver, by passing a message (expressing control) via the network. Schema (iv) is based on events so a service registers itself with events that will be produced by another service(s). In this section, we describe these four schemas in more detail.

Microservice Architecture [9] has gained considerable attention in the last few years, and is becoming increasingly important and popular for the development of IoT applications [10]. Every Microservice Architecture is a Service-Oriented Architecture (SOA), but not the other way round [11]. Hence, the service interaction schemas presented in this section can be used interchangeably in both Microservices and traditional SOA services.

A. Direct Service Interactions

The direct service interaction schema consist of sending a message (e.g., a XML-based document or a JSON-based document) from a sender to a receiver with no mediator between them [12], [13]. The sender interacts with the receiver using Remote Procedure Calls (RPC) [14] or REST API calls over HTTP [15]. RPC is akin to method invocations in traditional Object-Oriented programming languages, the main difference being that the invoked procedures may reside at different network addresses. REST does not require to know procedure names in advance, but only the location of external resources that can be manipulated using HTTP methods. Direct interactions are typically done using the request-response pattern [16].

Fig. 3 illustrates direct interactions for the EMoNet workflow. *ECG Analysis* triggers the control flow periodically by passing control to *Heart Rate History* so as to get the last sensor reading. Then, *Heart Rate History* returns the control to *ECG Analysis*. If there are signs of a heart attack, *ECG Analysis* passes control to the *Emergency* service which forwards control to the *Assistance* service of the nearest ambulance. Control is returned to *ECG Analysis*, after passing through the *Emergency* service. Fig. 3 shows that data flow follows control flow, and control and data are always originated in service computation.

Although they look old-fashioned, direct interactions are being used in emerging technologies (including IoT). For example, a Microservice choreography [17] describes direct interactions which are typically done using RESTful APIs [18]. REST has also been fostered by the Web of Things [8] for direct interactions among IoT services via the Web. Moreover, recent server-less programming frameworks for IoT [19] enable Java RPC for direct service interactions.

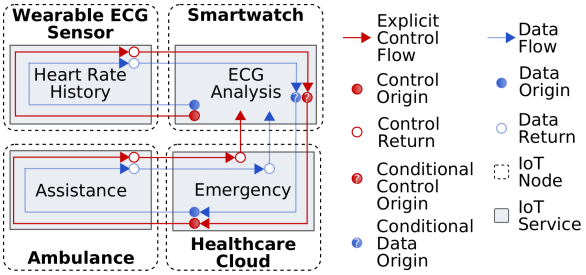


Fig. 3. Direct service interactions for the EMoNet workflow.

B. Indirect Service Interactions

The indirect interaction schema consists of using a service bus to broker sender requests, locate an appropriate receiver, transmit requests, and send responses back to senders. Since it passes messages between senders and receivers, a service bus can be thought of as a universal connector that provides a level of indirection between services [20], [21].

Fig. 4 illustrates indirect interactions for the EMoNet workflow, where *ECG Analysis* triggers control flow periodically. EMoNet services register their interfaces with a service bus that forwards control (and data) originated by *ECG Analysis* and *Emergency*, and sends back control (and data) from *Heart Rate History*, *Assistance* and *Emergency*, respectively. A glance at Fig. 4, reveals that even though a service bus provides indirection between senders and receivers, control and data are originated in service computation, and data follows control.

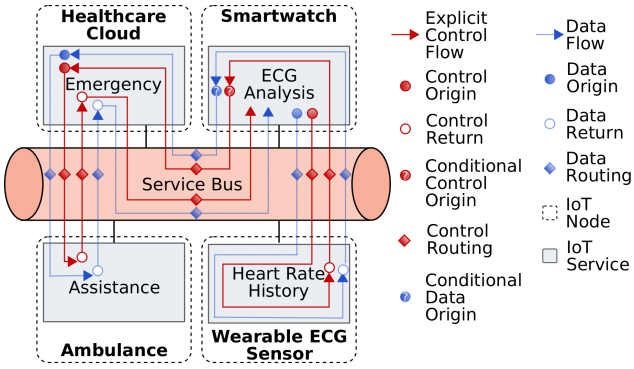


Fig. 4. Indirect service interactions for the EMoNet workflow.

Although the Enterprise Service Bus (ESB) [20] has been used for over a decade for enterprise SOA applications, the Microservice Architecture community has recently expressed their interest of using a lighter bus (known as Gateway) for indirect Microservice interactions [22], [23]. An IoT application can use a Gateway, an ESB or both [24].

C. Event-Driven Service Interactions

The event-driven interaction schema is based on the publish-subscribe pattern [16] so there are two roles: producer (i.e., service sender) and consumer (i.e., service receiver). Producers trigger events (perhaps carrying data) which are then stored in a queue. Consumers can subscribe to the events they are interested in, retrieve those events from the queue and react

accordingly. As events are dequeued in FIFO mode, there is no guarantee that responses from consumers are delivered to producers, so event-driven interactions usually follow the principle *fire and forget* [25], [26], [27].

Event-driven interactions can be done with or without a service bus. *P2P event-driven interactions* enable every service to be responsible of its own queue, so events are exchanged with no mediator. ZeroMQ is the most popular library to realize this interaction schema.[1] Fig. 5(a) shows P2P event-driven interactions for the EMoNet workflow.

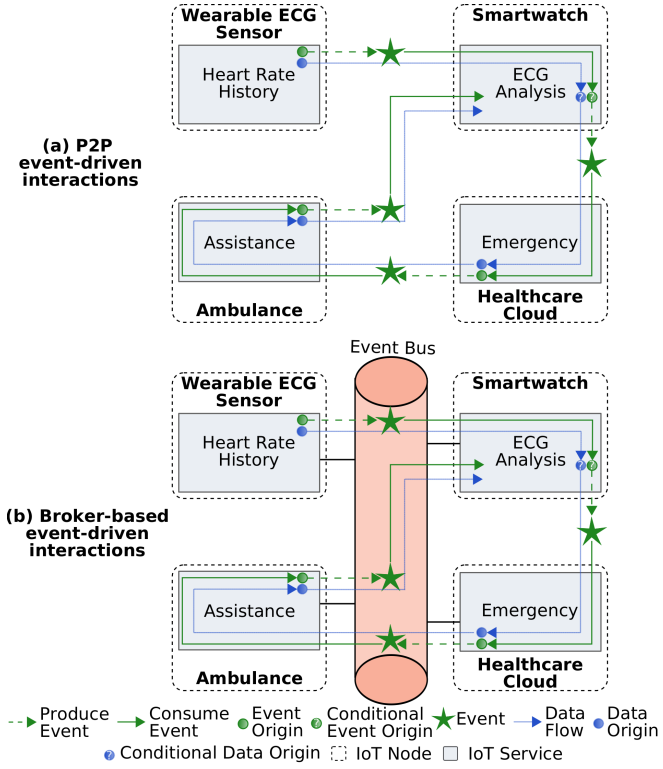


Fig. 5. Event-driven service interactions for the EMoNet workflow.

ECG Analysis periodically gets the last sensor readings by consuming events produced by *Heart Rate History*. If it detects a heart attack, *ECG Analysis* announces an emergency situation by producing an event for *Emergency*. After determining the nearest ambulance, *Emergency* produces an event that is consumed only by the *Assistance* service of that ambulance. Finally, *Assistance* produces an event for *ECG Analysis* to indicate the status of the current emergency.

Broker-based event-driven interactions use an event bus to manage event queues for a particular IoT application. An event bus is generally implemented using a messaging protocol such as the Advanced Message Queuing Protocol (AMQP) or the Message Queue Telemetry Transport (MQTT). RabbitMQ is the most popular implementation of the AMQP protocol.[2] The EMoNet services shown in Fig. 5(b) interact in the same way as the ones shown in Fig. 5(a), with the fundamental difference that events are now stored in the queue of an event bus.

[1]http://zeromq.org/
[2]https://www.rabbitmq.com/

Event-driven interactions are preferred to direct interactions for implementing Microservice choreographies [9], [23], [11]. Microservices use the strategy *smart endpoints and dumb pipes* [9] to define event-driven interactions in endpoints.

There is an increasing trend to use event-driven interactions for the exchange of data between IoT applications [25], [26]. In fact, the author in [28] found that a vast majority of current IoT platforms provide support for the event-driven interaction schema. In particular, broker-based event driven interactions are gaining considerable attention since MQTT was particularly designed for resource-constrained nodes [29], [30].

D. Exogenous Service Interactions

The exogenous service interaction schema enables a coordinator to define interactions (in the form of a workflow) over mutually anonymous services or other coordinators. Thus, control is always originated in coordinators and services do not interact with each other [31], [32].

Exogenous interactions can be done in one or multiple levels. One-level exogenous interactions are realized by orchestration [33], [34], where the coordinator is a workflow engine running in a specialized server. Fig. 6(a) shows one-level interactions for the EMoNet workflow.

EMoNet Workflow Engine is the coordinator for all the involved services. It passes control to *Heart Rate History* and *ECG Analysis* sequentially, in order to pull and analyze the last sensor readings. Then, according to the results of the analysis, the coordinator decides if there are signs of a heart attack. If so, the coordinator passes control to *Emergency* and *Assistance*, in that order. A glance at Fig. 6(a), reveals that control is always originated in the coordinator, and services are only concerned with returning control and data after performing some computation.

Multi-level exogenous interactions are done by hierarchical orchestration [35], [36] or exogenous connectors [37], [38], [39]. In this schema, multiple coordinators create a hierarchy of service interactions. Unlike, one-level exogenous interactions, in this schema control flows over multiple distributed coordinators.

Hierarchical orchestration [36] has multiple workflow engines, each of them responsible for the interaction of services or other workflow engines. In other words, it allows nesting a workflow within another workflow. Fig. 6(b) depicts a two-level hierarchical orchestration for the EMoNet services. *EMoNet Workflow Engine* coordinates the execution of coordinators *Monitoring Workflow Engine* and *Decision-Making Workflow Engine*. First, *EMoNet Workflow Engine* passes control to *Monitoring Workflow Engine* which is responsible for the interactions of the services *Heart Rate History* and *ECG Analysis*. Once control is returned from *Monitoring Workflow Engine* to *EMoNet Workflow Engine*, the latter passes control to *Decision-Making Workflow Engine*. If *Decision-Making Workflow Engine* determines that there are signs of a heart attack, it passes control to *Emergency* and *Assistance* sequentially. Finally, the control flow ends when the *Decision Making Workflow Engine* returns control to *EMoNet Workflow Engine*. Fig. 6(b) shows that a

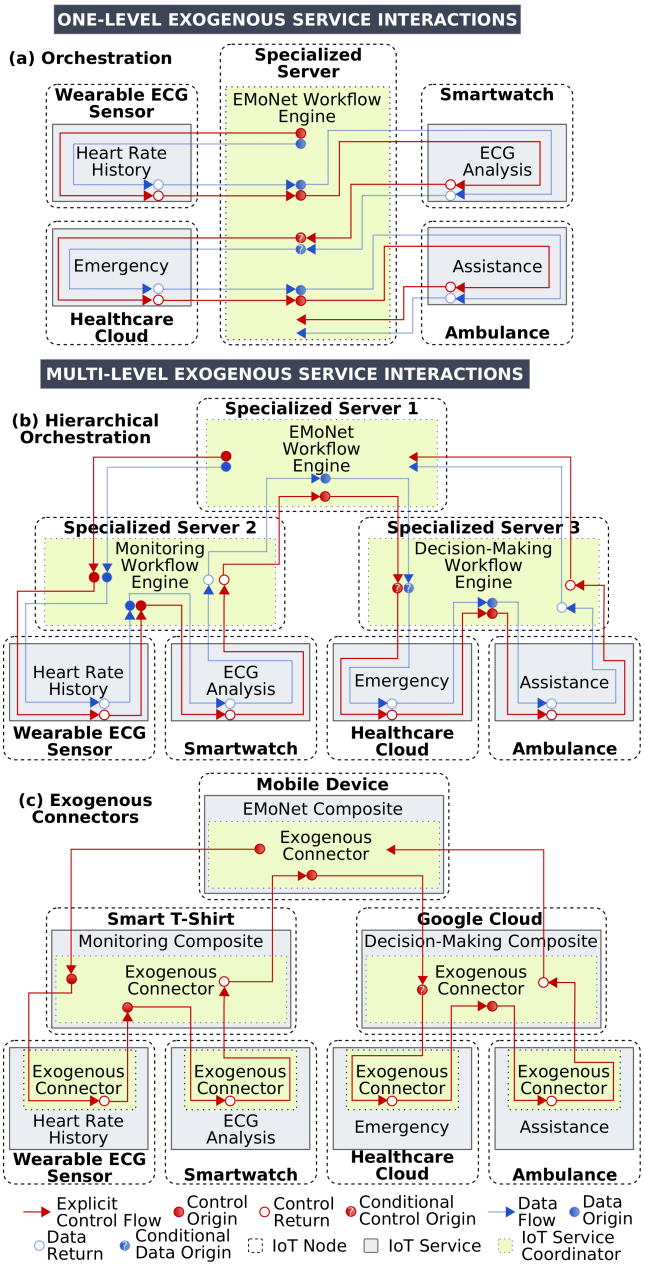


Fig. 6. Exogenous service interactions for the EMoNet workflow.

coordinator is able to pass and receive control and data to and from other coordinators.

Exogenous connectors are lightweight distributed coordinators that define micro-workflows. Fig. 6(c) illustrates how exogenous connectors coordinate service interactions for the EMoNet workflow. The control flow is the same as the one depicted in Fig. 6(b) for hierarchical orchestration. Unlike hierarchical orchestration, where control can be passed from a coordinator to a service, in exogenous connectors control is only passed between cooordinators (as data is an orthogonal dimension). Furthermore, the composition of two services results in a composite service (not a workflow) that preserves all the operations from the sub-services. Another difference is that coordinators do not need any specialized server as they can run in any IoT node (including resource-constrained nodes). For the EMoNet workflow, *Heart Rate History* and *ECG Analysis* are composed into *Monitoring Composite* which is deployed in a smart t-shirt; similarly, *Emergency* and *Assistance* are composed into *Decision-Making Composite* which is deployed in the *Google Cloud*. Exogenous connectors allow composite services to be further composed into even bigger services. For example, the *Monitoring Composite* and the *Decision-Making Composite* are composed into the *EMoNet composite* which is deployed in the patient's mobile device.

Due to the popularity of one-level exogenous interactions in SOA, in the last years we have seen the emergence of software platforms to support such a schema, e.g., Intel IoT SOL (Service Orchestration Layer) [40]. To the best of our knowledge, there are currently no IoT platforms for multi-level exogenous interactions.

IV. Evaluation and Results

This section presents the results of a qualitative evaluation of our service interaction schemas. A tick mark indicates that a specific interaction schema fulfills the requirement being analyzed, while a cross mark means the opposite. *NA* means that the analysis is not applicable for a particular interaction schema. In order to determine which schema best fulfills the scalability requirements of IoT applications, we specifically investigate the following research questions:

- **RQ1:** Which schemas allow the visualization of control flow?
- **RQ2:** Which schemas allow a separate reasoning between control and computation?
- **RQ3:** Which schemas allow decentralized interactions?
- **RQ4:** Which schemas enable services that are unaware of the location of other services?

A. RQ1: Explicit Control Flow vs Implicit Control Flow

Control flow can be *explicit* or *implicit*. Explicit control flow is visible as an entity defines the order in which individual services are executed. Conversely, implicit control flow is opaque since it is not defined anywhere, but it is implicit in the interactions of the participant services. Table I shows that event-driven interactions do not support visible control flow as it is implicit in the collaborative exchange of events (see Fig. 5) [17], [27]. In both direct interactions and indirect interactions, services are the entities who control the application flow, e.g., *ECG Analysis* defines a guard to execute *Emergency* when a heart attack is detected (see Figs. 3 and 4). In exogenous interactions, coordinators define control, e.g., *EMoNet Workflow Engine* defines control structures to realize one-level exogenous interactions for the EMoNet workflow (see Fig. 6(a)).

The amount of service interactions in IoT applications may become overwhelming due to the huge number of nodes involved. Since it is not visible, implicit control flow limits the scalability of IoT applications as the number of services grows and the complexity of service interactions increases.

TABLE I
EXPLICIT CONTROL FLOW IN SERVICE INTERACTION SCHEMAS.

		Explicit control flow
Direct interactions		✓
Indirect interactions		✓
Event-driven interactions	P2P	✗
	Broker-based	✗
Exogenous interactions	One-level	✓
	Multi-level	✓

Implicit control flow has been an issue for software companies over many years and it is undoubtedly a barrier for IoT. For instance, Netflix has recently expressed that implicit control flow limits the scalability of distributed applications, as they found that process flows are spread across multiple applications and it is difficult to monitor workflow processes. As an attempt to visualize control flow, Netflix recently moved from event-driven interactions to exogenous interactions [3].

Visualizing control flow (e.g., to find execution paths) in event-driven interactions is challenging because it is necessary to look at logs to understand the correlation between events [41]. This evidently makes it hard to monitor workflow execution, debug code and modify application workflow. For instance, in the event-based EMoNet workflow it is hard to know which is the most popular ambulance, since there are many ambulances involved. Explicit control flow helps to mitigate this problem so a graphical user interface [3], [42] can be used to display a visual representation of the blueprint with the paths the control has taken during the execution of EMoNet.

In general, explicit control flow is crucial to facilitate the monitoring, maintenance and evolution of IoT applications [23], [3], [27].

B. RQ2: Separation between control and computation

IoT is characterized by heterogeneity in several forms, e.g., different vendors, different hardware and a wide variety of programming languages. For this reason, control and computation should be orthogonal dimensions in every IoT application, in order to enable a flexible integration of services in a heterogeneous environment [43], [44], [45], [4].

Computation is the low-level functionality of an IoT node (provided by a service), and control defines the logic to realize service interactions. The separate reasoning of these concerns enables application developers to focus on the IoT application logic, whilst IoT service developers can focus on the development of efficient service functionality. This separation not only results in reduced time to market, but also reduced software production and maintenance costs.

In both direct interactions and indirect interactions, a sender and a receiver are tightly coupled in terms of control, since control is always originated in the sender's computation. For example, in the EMoNet workflow done by either indirect interactions or direct interactions, *ECG Analysis* is responsible for the conditional control structure that passes control to *Emergency* when a heart attack is detected (see Figs. 3 and 4).

Services mixing control with computation are not reusable, as control flow may vary from one application to another. Suppose we want EMoNet to execute a planning phase after the analysis phase, in order to predict heart diseases and determine heart attacks in real-time. To do so, the *HPC Computing* node, providing the *ECG Planning* service, is introduced. In the EMoNet workflow done by either direct interactions or indirect interactions, both *ECG Analysis* and *ECG Planning* must be changed to accommodate the new requirement. In particular, the conditional control structure is removed from *ECG Analysis* and added into *ECG Planning* which is now responsible for passing control to *Emergency* (when a heart attack is detected). For that reason, *ECG Analysis* is not reused in the new application.

Our analysis of the separation between control and computation is not applicable for event-driven interactions, since control flow is implicit in this schema. Nevertheless, in event-driven interactions, events are originated in service computation (see Fig. 5). For example, *Emergency* and *ECG Planning* would require changes in their computation so as to accommodate the planning phase. In particular, *ECG Planning* needs to consume the events produced by *ECG Analysis*, while *Emergency* needs to consume the events produced by *ECG Planning*. For that reason, *Emergency* is not reused in the new application.

Table II shows that only exogenous interactions separate control from computation, as control is always originated in the coordinator(s) (see Fig. 6). In contrast to the rest of the schemas, exogenous interactions do not require changing any service to support the planning phase, but only changing the application logic defined in the coordinator(s). Thus, as business requirements change, developers can manage changes in the application logic without taking care of IoT service functionality [43].

TABLE II
SEPARATION BETWEEN CONTROL AND COMPUTATION IN SERVICE INTERACTION SCHEMAS.

		Separation between control and computation
Direct interactions		✗
Indirect interactions		✗
Event-driven interactions	P2P	N/A
	Broker-based	N/A
Exogenous interactions	One-level	✓
	Multi-level	✓

When events or control are originated in service computation, an application workflow is embedded in the code of plenty of services. This is in fact one of the reasons for which Netflix stop using event-driven interactions. Exogenous interactions is the only schema that enables the development of workflow-agnostic services, as a consequence of the separation between control and computation. For that reason, Netflix preferred the use of exogenous interactions to event-driven interactions.

C. RQ3: Decentralized Service Interactions

Service interactions can be *centralized* or *decentralized*. Centralized service interactions means that control, events (or even data) pass through a single central entity. By contrast,

decentralized service interactions means that control, events (or even data) are passed in a P2P fashion as workflow (expressed by control or events) is distributed over two or more entities.

Table III shows that indirect interactions, one-level exogenous interactions (i.e., orchestration) and broker-based event-driven interactions are centralized schemas. Indirect interactions require a service bus for passing control and data between services (see Fig. 4). Broker-based event-driven interactions use an event bus to handle events (see Fig. 5(b)). In one-level exogenous interactions, a central engine defines a workflow for passing control (and frequently data) between services (see Fig. 6(a)).

TABLE III
DECENTRALIZATION IN SERVICE INTERACTION SCHEMAS.

		Decentralization
Direct interactions		✓
Indirect interactions		✗
Event-driven interactions	P2P	✓
	Broker-based	✗
Exogenous interactions	One-level	✗
	Multi-level	✓

Even though a centralized approach facilitates the design and maintenance of an IoT application, it possesses several drawbacks that have been recognized by many researchers [35], [4], [46], [36]. For example, in Fig. 6(a) the data generated by *Wearable ECG Sensor* (which is important for *ECG Analysis*) will be routed through *EMoNet Workflow Engine*, even if this data is unimportant to that coordinator. In general, a centralized approach requires an extra network hop for service interactions.

Furthermore, IoT nodes usually generate a huge amount of data. Hence, a central entity may potentially become a performance bottleneck since all the communication will pass through it; thereby, leading to high consumption of network bandwidth, and therefore, unnecessary network traffic. A central entity can also become a single point of failure and attack, thereby impacting the availability of an IoT application.

No single organization should govern an entire workflow or data, as an IoT application may cross administrative domains and organizations may want control over their own part. For example, EMoNet could cross two administrative domains: a data analytics company that processes sensor data and a health telemetry company that monitors patients' heart rate.

According to [5], IoT nodes must possess the ability to interact among themselves with no mediator between them. Decentralized service interactions are more complex than their counterpart, but they bring up increased scalability, availability and reliability for an IoT application by:

- Improving concurrency, load balancing and fault-tolerance due to the use of multiple loci of control or multiple event handlers.
- Bringing performance enhacements (e.g., better throughput) for service interactions.
- Reducing network traffic and latency. as no extra hop is required for service interactions.

Table III shows that decentralization is present in direct interactions, multi-level exogenous interactions (i.e., hierarchical orchestration and exogenous connectors) and P2P event-driven interactions. Direct interactions do not require any mediator for passing control between services (see Fig. 3). In multi-level exogenous interactions, coordinators are the only entities that pass control to services or other coordinators (see Figs. 6(b) and 6(c)). Similarly, P2P event-driven interactions do not rely on a bus for event management, as every service is responsible of its own queue (see Fig. 5(b)).

D. RQ4: Location Transparency

IoT is highly dynamic due to the intermittent connection and spontaneous failures of IoT nodes, resulting in nodes (and ergo services) frequently changing locations over time. For that reason, churn is one of the main challenges of IoT applications as they usually operate in a dynamic and uncertain environment [6], [47]. For example, the *Wearable ECG Sensor* is a resource constrained-node that can run out of battery with the subsequent disconnection from the network. Similarly, an *Ambulance* may experience frequent disconnections due to its high mobility.

Service location transparency is crucial to mitigate churn in IoT applications, as it enables services to be unaware of the physical location of other services. Table IV shows that indirect interactions, broker-based event-driven interactions and exogenous interactions provide location transparency. In indirect interactions, the service bus is the only entity aware of services' locations. In broker-based event-driven interactions, publishers and subscribers do not know the location of one another, but they only know what events to produce and consume, respectively. In exogenous interactions, coordinators encapsulate services' locations as they are responsible for service interactions.

TABLE IV
LOCATION TRANSPARENCY IN SERVICE INTERACTION SCHEMAS.

		Location transparency
Direct interactions		✗
Indirect interactions		✓
Event-driven interactions	P2P	✗
	Broker-based	✓
Exogenous interactions	One-level	✓
	Multi-level	✓

Direct interactions and P2P event-driven interactions do not support location transparency, as they require senders to know the location of receivers a priori. The main problem of these schemas is that senders need to be changed every time the receivers' location change. Although this issue can be solved using a service discovery mechanism (e.g., querying a service registry) [8], it would require an extra network hop. In fact, centralized interaction schemas enclose a discovery component in the middleman [48]. Assuming there is no discovery mechanism for the EMoNet workflow based on direct interactions or P2P event-driven interactions, *Emergency* must be updated every time an *Ambulance* changes location. This is a frightening situation for EMoNet because there is a huge number of ambulances constantly changing locations.

TABLE V
ANALYSIS OF SERVICE INTERACTION SCHEMAS.

	Direct interactions	Indirect interactions	Event-driven interactions		Exogenous interactions	
			P2P	Broker-based	One-level	Multi-level
Explicit control flow	✓	✓	✗	✗	✓	✓
Separation between control and computation	✗	✗	NA	NA	✓	✓
Decentralized control flow	✓	✗	✓	✗	✗	✓
Service location transparency	✗	✓	✗	✓	✓	✓

V. DISCUSSION

Table V summarizes the results of our qualitative evaluation. It particularly shows how well service interaction schemas fulfill the scalability requirements of IoT applications: explicit control flow, separation between control and computation, decentralized interactions, and service location transparency.

Direct interactions and indirect interactions cover 50% of the requirements, respectively. Event-driven interactions is the worst schema since it only meets 25% of those requirements in both P2P and broker-based. Lacking only decentralization, one-level exogenous interactions cover 75% of the desiderata. Multi-level exogenous interactions is the only schema that fulfills all the scalability requirements of IoT applications.

In some scenarios, it could be useful to combine interaction schemas. For example, in order to provide asynchronous interactions in EMoNet, services can combine event-driven interactions with direct interactions. *ECG Analysis* can interact via an event bus with both *Heart Rate History* and *Emergency*, whilst *Emergency* can use direct interactions to request the *Assistance* service of the nearest ambulance.

A service bus can be [21]: (i) *distributed*, (ii) *with technical intelligence* or (iii) *with business intelligence*. Options (i) and (ii) are used only for data and control routing, whilst (iii) can be used to define coordination logic in addition [22]. Even though it is typically used only for straightforward workflows, (iii) is a special case of one-level exogenous interactions.

Although the Microservices community recommends the avoidance of (iii) as they do not want business logic embedded in a service bus [22], there is an increasing tendency to use exogenous interactions for Microservices in traditional SOA applications [34], [3]. By contrast, in the context of IoT, event-driven interactions are currently more popular. However, given the advantages of exogenous interactions, as evidenced by their increasing adoption in traditional SOA applications, we envision that exogenous interactions will increase in popularity in Microservice-based IoT applications in the next years.

A Distributed Service Bus (DSB) [49] is often seen as a decentralized approach due to the existence of a federation of brokers. However, it consists solely of a distribution of middleware components over different nodes. According to our view of decentralization presented in Sec. IV-C, the existence of an intermediary (or intermediaries) for service interactions leads to a centralized approach. As we noted in Sec. IV-C, a purely decentralized approach removes the need of a middleman (or middlemen) which, among other issues, introduces an extra network hop for service interactions.

In order to achieve decentralization, the Microservices community fosters direct interactions between Microservices. Nevertheless, direct interactions impact performance because a connection must be open for the entire duration of an interaction, and a Microservice participant needs a reference (i.e., a client library) for every Microservice it communicates directly with. Maintaining references to other Microservices is costly. Furthermore, a HTTP connection may become a bottleneck, especially for long running Microservices. This is undoubtedly a problem for resource-constrained IoT nodes which do not have communication and storage capabilities to support long-running transactions or to store multiple references. To solve this issue, the Internet Engineering Task Force (IETF) has developed the Constrained Application Protocol (CoAP) [50]. CoAP has been proved to be a simpler and more cost-efficient alternative to HTTP/REST in several IoT scenarios involving resource-constrained nodes [51]. Nevertheless, CoAP does not support the separation between control and computation.

The separation between control and data is also crucial for the scalability of IoT applications. It means that data is never passed alongside control, thereby allowing a separate reasoning between data flow and control flow, which could result in the development of an efficient data exchange approach. For instance, a P2P data exchange can be used to reduce the number of network hops, thereby avoiding network congestion as shown by [46]. The separation between control and data also enables the reuse of data flow without the need of modifying control flow. Hence, data flow and control flow can evolve separately. Exogenous connectors in multi-level exogenous interactions provide semantics for the separation between control and data. Although data typically follows control in orchestration approaches, the separation between control and data has already been done in such approaches [46], [52]. For the EMoNet workflow based on exogenous interactions, we assumed that there is no separation between control and data.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we classified and analyzed service interactions into four schemas, namely direct interactions, indirect interactions, event-driven interactions and exogenous interactions.

We conducted a qualitative evaluation to determine which interaction schema best fulfills the scalability requirements of IoT applications: explicit control flow, decentralized interactions, separation between control and computation, and service location transparency. We showed that multi-level exogenous

interactions is the most promising schema since it meets all the desiderata for the scalability of IoT applications.

Network performance is another aspect that needs to be considered when tackling scalability. We would like to conduct experiments to quantitatively evaluate the throughput of the service interaction schemas presented in this paper.

To the best of our knowledge, there are no IoT platforms based on multi-level exogenous interactions. As this is the most promising schema for IoT, we hope to see its realization in the coming years. In fact, we are currently working on the development of such a platform.

REFERENCES

[1] "Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)," https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2018.

[2] M. Abbott and M. Fisher, *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, 2nd ed. Addison-Wesley, 2015.

[3] Netflix, "Conductor," https://netflix.github.io/conductor/, 2016.

[4] D. Wutke *et al.*, "Model and infrastructure for decentralized workflow enactment," in *ACM Symposium on Applied Computing*, 2008, pp. 90–94.

[5] S. Roy and C. Chowdhury, "Integration of Internet of Everything (IoE) with Cloud," *Beyond the Internet of Things*, vol. 24, no. 6, pp. 199–222, 2017.

[6] R. Buyya and A. Dastjerdi, *Internet of Things: Principles and Paradigms*. Amsterdam Boston Heidelberg: Morgan Kaufmann, 2016.

[7] J. Soriano *et al.*, "Internet of Services," *Evolution of Telecommunication Services*, vol. 7768, pp. 283–325, 2013.

[8] D. Guinard *et al.*, "Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services," *IEEE Transactions on Services Computing*, vol. 3, no. 3, pp. 223–235, 2010.

[9] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014.

[10] K. Khanda *et al.*, "Microservice-Based IoT for Smart Buildings," in *31st Int. Conference on Advanced Information Networking and Applications Workshop*, 2017, pp. 302–308.

[11] O. Zimmermann, "Microservices tenets," *Comput Sci Res Dev*, vol. 32, no. 3, pp. 301–310, 2017.

[12] R. Dijkman and M. Dumas, "Service-oriented design: A multi-viewpoint approach," *Int. J. Coop. Info. Syst.*, vol. 13, no. 04, pp. 337–368, 2004.

[13] C. Pautasso *et al.*, "Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 805–814.

[14] "gRPC," https://grpc.io/, 2018.

[15] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," PhD Thesis, University of California, Irvine, 2000.

[16] A. Barros *et al.*, "Service Interaction Patterns," in *Int. Conference on Business Process Management*, 2005, pp. 302–318.

[17] Q. Sheng *et al.*, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, 2014.

[18] T. Ahmed and A. Srivastava, "Service Choreography: Present and Future," in *IEEE Int. Conference on Services Computing*, 2014, pp. 863–864.

[19] I. Nakagawa *et al.*, "Dripcast - Architecture and Implementation of Server-less Java Programming Framework for Billions of IoT Devices," *Journal of Information Processing*, vol. 23, no. 4, pp. 458–464, 2015.

[20] M. Schmidt *et al.*, "The Enterprise Service Bus: Making service-oriented architecture real," *IBM Systems Journal*, vol. 44, no. 4, pp. 781–797, 2005.

[21] N. Josuttis, *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[22] C. Pautasso *et al.*, "Microservices in Practice, Part 2: Service Integration and Sustainability," *IEEE Software*, vol. 34, no. 2, pp. 97–104, 2017.

[23] S. Newman, *Building Microservices*, 1st ed. O'Reilly Media, 2015.

[24] L. Gong, "A software architecture for open service gateways," *IEEE Internet Computing*, vol. 5, no. 1, pp. 64–70, 2001.

[25] D. Happ and A. Wolisz, "Limitations of the Pub/Sub pattern for cloud based IoT and their implications," in *Cloudification of the Internet of Things*, 2016, pp. 1–6.

[26] Y. Zhang *et al.*, "Integrating Events into SOA for IoT Services," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 180–186, 2017.

[27] M. Fowler, "What do you mean by "Event-Driven"?" https://martinfowler.com/articles/201701-event-driven.html, 2017.

[28] P. P. Ray, "A survey of IoT cloud platforms," *Future Computing and Informatics Journal*, vol. 1, no. 1, pp. 35–46, 2016.

[29] J. Soldatos *et al.*, "OpenIoT: Open Source Internet-of-Things in the Cloud," in *Interoperability and Open-Source Solutions for the Internet of Things*, 2015, pp. 13–25.

[30] A. Antonić *et al.*, "A mobile crowd sensing ecosystem enabled by CUPUS: Cloud-based publish/subscribe middleware for the Internet of Things," *Future Generation Computer Systems*, vol. 56, pp. 607–622, 2016.

[31] K.-K. Lau *et al.*, "Exogenous Connectors for Software Components," in *8th Int. Conference on Component-Based Software Engineering*, 2005, pp. 90–106.

[32] S.-S. Jongmans *et al.*, "Orchestrating web services using Reo: From circuits and behaviors to automatically generated code," *Service Oriented Computing and Applications*, vol. 8, no. 4, pp. 277–297, 2014.

[33] C. Lee *et al.*, "Blueprint Flow: A Declarative Service Composition Framework for Cloud Applications," *IEEE Access*, vol. 5, pp. 17 634–17 643, 2017.

[34] S. Alpers *et al.*, "Microservice Based Tool Support for Business Process Modelling," in *IEEE 19th Int. Enterprise Distributed Object Computing Workshop*, 2015, pp. 71–78.

[35] G. Chafle *et al.*, "Decentralized Orchestration of Composite Web Services," in *13th International World Wide Web Conference (WWW '04)*, 2004, pp. 134–143.

[36] W. Jaradat *et al.*, "Towards an autonomous decentralized orchestration system," *Concurrency Computat.: Pract. Exper.*, vol. 28, no. 11, pp. 3164–3179, 2016.

[37] D. Arellanes and K.-K. Lau, "Exogenous Connectors for Hierarchical Service Composition," in *10th IEEE International Conference on Service Oriented Computing and Applications (SOCA '17)*, 2017, pp. 125–132.

[38] D. Arellanes and K. K. Lau, "D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures," in *7th IEEE International Symposium on Cloud and Service Computing (SC2 '17)*, 2017, pp. 283–286.

[39] D. Arellanes and K.-K. Lau, "Algebraic Service Composition for User-Centric IoT Applications," in *3rd International Conference on Internet of Things*, 2018.

[40] Intel, "IoT Services Orchestration Layer," http://01org.github.io/intel-iot-services-orchestration-layer, 2016.

[41] A. Burattin *et al.*, "Control-flow discovery from event streams," in *IEEE Congress on Evolutionary Computation*, 2014, pp. 2420–2427.

[42] "Zipkin," https://zipkin.io/, 2018.

[43] "Amazon Simple Workflow Service (SWF)," https://aws.amazon.com/swf/, 2018.

[44] F. Leymann, "Web Services: Distributed Applications Without Limits." in *Database Systems for Business, Technology and Web*, vol. 26, 2003, pp. 2–23.

[45] K.-K. Lau *et al.*, "A Component Model for Separation of Control Flow from Computation in Component-Based Systems," *Electronic Notes in Theoretical Computer Science*, vol. 163, no. 1, pp. 57–69, 2006.

[46] A. Barker *et al.*, "The Circulate architecture: Avoiding workflow bottlenecks caused by centralised orchestration," *Cluster Computing*, vol. 12, no. 2, pp. 221–235, 2009.

[47] P. Barnaghi and A. Sheth, "On Searching the Internet of Things: Requirements and Challenges," *IEEE Intelligent Systems*, vol. 31, no. 6, pp. 71–75, 2016.

[48] A. Ngu *et al.*, "IoT Middleware: A Survey on Issues and Enabling Technologies," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, 2017.

[49] F. Baude *et al.*, "ESB federation for large-scale SOA," in *ACM Symposium on Applied Computing*, 2010, pp. 2459–2466.

[50] Z. Shelby *et al.*, "Constrained Application Protocol (CoAP)," https://datatracker.ietf.org/doc/rfc7252/, 2014.

[51] "Comparing the cost-efficiency of CoAP and HTTP in Web of Things applications," *Decision Support Systems*, vol. 63, pp. 23–38, 2014.

[52] C. Pautasso, "Composing RESTful services with JOpera," in *8th Int. Conference on Software Composition*, 2009, pp. 142–159.

4.7 Evaluating IoT Service Composition Mechanisms for the Scalability of IoT Systems

Damian Arellanes and Kung-Kiu Lau

Submitted to **Future Generation Computer Systems**,
Status: *Under revision* (as of 2019).

Impact Indicators:
• Journal Core ranking (2018): A,
• Journal impact factor (2018): 5.768,
• Journal SJR (2018): 0.835,
• Journal SNIP (2018): 2.464.

Cite as: [AL19b]

Summary: This paper qualitatively compares DX-MAN with existing IoT service composition mechanisms in terms of *compositionality* and the functional scalability requirements identified: *explicit control flow*, *distributed workflows*, *location transparency*, *decentralised data flows*, *workflow variability*, and *separation of control, data and computation*.

Comments on authorship: I proposed the main idea of the paper, investigated IoT service composition mechanisms, conducted a qualitative comparison, analysed results, provided and edited all graphics, participated in the entire writing process and addressed the reviewer's comments. My supervisor, Kung-Kiu Lau, also contributed to the idea, proofread the paper and approved the results. He also guided the whole research process.

Key contributions: Contribution 1 and Contribution 2 (see Section 1.5).

Evaluating IoT Service Composition Mechanisms for the Scalability of IoT Systems

Damian Arellanes*, Kung-Kiu Lau

Department of Computer Science, The University of Manchester, Manchester M13 9PL, United Kingdom

ARTICLE INFO

Keywords:
Scalability
IoT service composition
Orchestration
Choreography
Dataflows
Workflows

ABSTRACT

The Internet of Things (IoT) is an emerging paradigm where practically every (physical and virtual) thing will be interconnected through innovative distributed services. Since the number of connected things is rapidly growing, IoT systems will require the composition of plenty of services into complex workflows. Thus, scalability in terms of the size of IoT systems becomes a significant concern. In this paper, we review and evaluate the fundamental semantics of existing IoT service composition mechanisms to determine how well they fulfil the scalability requirements of IoT systems. We identify scalability desiderata and, accordingly, our findings show that dataflows, orchestration and choreography do not fully satisfy such desiderata, unlike a novel composition mechanism called DX-MAN.

1. Introduction

The Internet of Things (IoT) is an emerging paradigm that promises the interconnection of (physical and virtual) things through innovative distributed services. Like traditional enterprise services, IoT services interact in many different ways via the Internet, in order to realise a global system workflow. However, unlike traditional enterprise systems, IoT systems will require the interaction of billions of services as the number of connected things (and therefore services) is rapidly growing [1, 2]. Thus, scalability becomes a crucial concern.

Scalability is typically considered as a system capability to handle increasing workloads [3, 4, 5, 6, 7, 8]. In particular, vertical scalability [9, 10, 11] refers to the addition or removal of computing resources in a single IoT node, while horizontal scalability [2, 12, 13] involves the addition or removal of IoT nodes. These kinds of scalability have been addressed by an extensive body of research [4, 10, 11, 14, 15, 16, 17, 18, 19, 20], unlike scalability in terms of the number of services composed in an IoT system, which we refer to as *functional scalability*.

Existing service composition mechanisms were primarily designed for the integration of static enterprise services, not for the physical world. For that reason, they may not address the functional scalability challenges that IoT systems pose. Early IoT systems were deployed in closed environments, using private Application Programming Interfaces (APIs) and private data. However, future IoT systems will be deployed in open environments (also known as software ecosystems [21]) with an overwhelming number of available services, as a result of the huge number of connected things [22]. For that reason, billions of IoT services will be composed into complex IoT systems [23, 24, 25, 26, 27, 28]. This raises the challenging question of *How to construct IoT systems composed of an ultra-large number of services?*

In this paper, we study the ability of service composition mechanisms to handle any number of IoT services. Our aim is to determine *which service composition mechanism best fulfils the functional scalability requirements of IoT systems*. To answer this, we have reviewed the fundamental semantics (not specific implementation technologies) of existing composition mechanisms, and proposed an evaluation framework that considers six crucial desiderata: (i) explicit control flow [29, 30]; (ii) distributed workflows [23, 31, 32]; (iii) location transparency [20, 33]; (iv) decentralised data flows [23, 32, 34]; (v) separation of control, data and computation [35, 36, 37, 38]; and (vi) workflow variability [23, 39, 40, 41]. Our evaluation framework serves as a guideline for defining the semantics of future IoT service composition mechanisms.

The remainder of the paper is structured as follows. In the next section, we present an overview of IoT services, scalability and service composition. Section 3 outlines the related work on IoT service composition mechanisms. Section 4 introduces a motivating IoT scenario based on a smart parking system. The scenario is then considered in Section 5 to comprehensively describe the requirements for functional scalability and the rationale behind them. Section 6 analyses service composition mechanisms on the basis of the functional scalability requirements. Section 7 presents an evaluation that determines how well service composition mechanisms fulfil the scalability requirements. Our findings are then discussed in Section 8. Finally, Section 9 draws the conclusions and presents the directions for future research.

2. Background

This section presents a background for the rest of the paper, which includes an overview of IoT services, service composition and scalability.

2.1. IoT Services

Kevin Ashton coined the term *Internet of Things* (IoT) in a presentation made in 1999 at Procter and Gamble [42], referring to the interconnection of everything via the Internet for the creation of an ubiquitous computing environment [43]. As per the recommendation of ITU-T Y.4000, IoT has been

*Corresponding author.
damian.arellanesmolina@manchester.ac.uk (D. Arellanes); kung-kiu.lau@manchester.ac.uk (K. Lau)
ORCID(s): 0000-0002-0074-390X (D. Arellanes)

recently redefined as "*a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies*" [44].

A *thing* is practically a physical or virtual construct of the real-world, which is capable of being identified and integrated into communication networks through specific protocols. The difference between physical and virtual things lies in their tangibility [45]. A physical thing is a tangible object of the physical world, which is capable of being sensed, actuated and connected, e.g., home appliances, robots, buildings, plants and people. Contrastingly, a virtual thing is a non-tangible construct formed from a human idea which only exists in the information world, e.g., Clouds and enterprises.

The broad range of available things inevitably requires dealing with a high degree of heterogeneity in a distributed environment. Accordingly, a Service-Oriented Architecture (SOA) represents the best way of dealing with this issue [46]. It is *a logical way of designing a software system to provide services either to end-user applications or other services distributed in a network, via published and discoverable interfaces* [47]. Thus, it is expected that physical and virtual things will provide services to expose behaviour via interfaces [5, 48, 49, 50, 51, 52].

According to the Oxford dictionary, the word *service* can be a noun or a verb referring to the *action of helping or doing work for someone*. Considering a service as a noun allows the encapsulation of behaviour (i.e., actions) in the form of operations described as verbs. Thus, *a software service is a distributed software component that provides a set of operations through network-accessible endpoints* [53, 54]. In general, IoT services are virtual representations of the behaviour of things, which can be combined with other services into more complex behaviours to yield complex service-oriented IoT systems [52, 55, 56, 57]. Thus, things are integrated through the composition of the services they provide (see Section 2.2).

Resource-constrained things (e.g., pulse sensors) typically provide fine-grained services for basic functionality (e.g., fetching sensor data), whilst non-resource constrained things (e.g., a Cloud) may offer coarse-grained services in addition (e.g., services for geolocation or complex industrial processes). Enterprise services are typically coarse-grained as they are deployed on resource-rich infrastructures, whilst services of physical things are often fine-grained because they are usually deployed on resource-constrained things.

Figure 1 shows the relationship between things, services and operations. Figure 1(a) depicts a *washing machine* (i.e., a resource-constrained physical thing) that offers the *Washing* and *Drying* fine-grained services with two operations each (for starting and stopping the respective processes). Figure 1(b) shows a *City Council Cloud* (i.e., a non-resource constrained virtual thing) that offers the services *CouncilTax* and *Parking*. The *CouncilTax* service provides the operations *pay* (to pay a tax bill) and *check* (to query council tax information). The *Parking* service offers the operations *getNearest* (for getting the closest parking space from a driver's location) and *registerDisabled* (for registering an impaired driver).

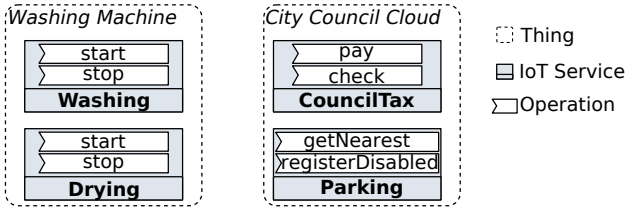


Figure 1: Relationship between things, IoT services and operations.

For the rest of the paper, we use the notation *S.O* to denote an operation *O* in service *S*, e.g., *Parking.getNearest* refers to the *getNearest* operation provided by the *Parking* service.

2.2. IoT Service Composition Revisited

An IoT service is a distributed unit of composition, which constitutes the virtual representation of a thing's behaviour, and can be either atomic or composite. An *atomic service* is a well-defined and self-contained piece of behaviour that cannot be divided into other services [58, 59, 60]. A *composite service*, on the other hand, is a more complex unit that provides value-added functionality and is formed by the combination of (atomic or composite) services [52, 55, 57, 58, 59, 61, 62]. For example, a humidity sensing service can be combined with a temperature service into an air conditioning composite [63].

The ability of combining services is referred to as *compositionality* and is realised by a *composition mechanism* [59]. Thus, an IoT system requires a things infrastructure, the definition of what a service is and the selection of a composition mechanism [64]. In any scenario, composition is done regardless of both the technologies being used and the things infrastructure. Service technologies include REST [65, 66], WS-* [67, 68], OSGi [69, 70] and many others.[1]

A service composition mechanism defines a meaningful interaction between services [59] by considering two functional dimensions: control flow and data flow [71, 72]. Control flow refers to the order in which interactions occur [73, 74], whilst data flow defines how data is moved among services [71]. In this paper, we focus on service composition mechanisms that define behaviour by workflows, namely (centralised and distributed) dataflows, (centralised and distributed) orchestration, choreography, and a novel composition mechanism called DX-MAN. Section 6 provides a detailed description of these mechanisms.

A workflow describes a series of discrete steps for the realisation of a computational activity, which can be control-driven, data-driven or hybrid [75]. In a control-driven workflow, steps (also known as tasks [76], actors [77], transitions [78], procedures [79], thorns [80], activities [81] and units [82]) are executed according to explicit control flow constructs that define sequencing, looping, branching or parallelising. A data-driven workflow invokes steps whenever

[1] In RESTful services, operations are exposed as resources [65].

data becomes available without explicitly defining any control flow constructs [83]. In a hybrid workflow, some parts are control-driven, while others are data-driven [82]. Figure 2 illustrates a generic workflow that executes the operation op_1, then decides to invoke either op_2 or op_3 and, finally, triggers the operations op_4 and op_5 in parallel. Operation invocations happen regardless of the workflow kind.

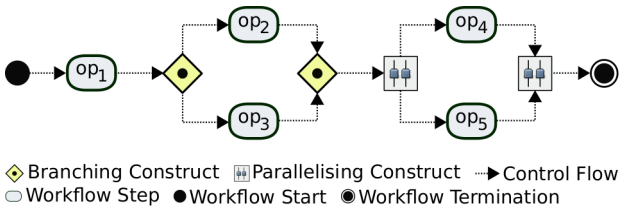


Figure 2: A generic workflow.

Workflows are increasingly important for IoT systems because they allow the integration of IoT services into complex tasks that automate a specific context [84, 85, 86, 87, 88]. For example, a smart home can be automated with a workflow that regulates the temperature of a room according to environmental changes. In the domain of smart agriculture, a workflow can be defined to analyse data coming from harvest sensors, predict diseases and react accordingly.

2.3. Scalability of IoT Systems

With the advent of hardware technologies, the number of IoT services is rapidly growing due to the excessive increase in the number of connected things. Currently, there are about 19 billion connected things, and it is predicted that this number will grow exponentially in the coming years [1, 2]. Thus, unlike traditional enterprise systems, scalability becomes a crucial concern for the full realisation of IoT systems.

Typically, scalability is the capability to handle increasing workloads in a IoT system [3, 4, 5, 6, 7, 8]. In this case, it is a metric that indicates how system performance improves over time. Workloads are typically measured in terms of either the number of requests dispatched [3] or the data streams generated [7]. The overall goal of scalable solutions is to enhance the Quality of Service (QoS) for guaranteeing a certain level of performance under the presence of high workloads, e.g., by minimising bandwidth, energy, latency and response time while maximising throughput. To quantitatively measure QoS, several network aspects of a service are considered such as jitter, throughput, packet loss and availability [8].

Currently, there are two kinds of scalability: vertical and horizontal.[2] Vertical scalability (or scaling up) [9, 10, 11] refers to the addition or removal of computing resources in a single thing, e.g., adding more memory to increase buffer size or adding more processor capacity to speed up processing. On the other hand, horizontal scalability (or scaling out) [2, 12, 13] involves the addition or removal of things in an IoT system. Its goal is to distribute the workload over multiple things to decrease individual loads, minimise response time and enhance concurrency. Figure 3 depicts the contrast between vertical and horizontal scalability.

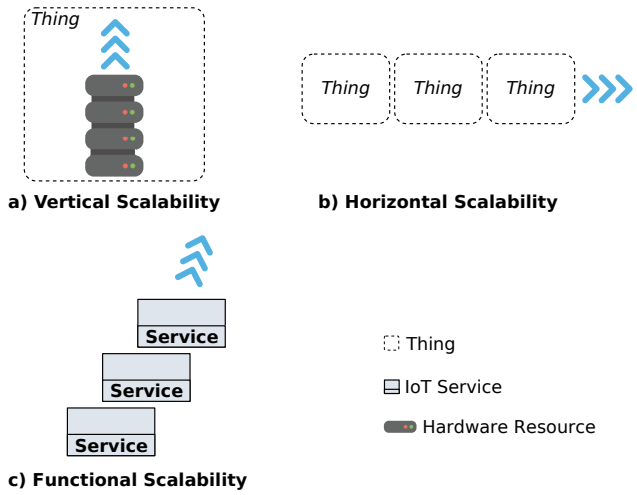


Figure 3: Scalability of IoT systems.

Both vertical and horizontal scalability have been extensively addressed in the literature [4, 10, 11, 14, 15, 16, 17, 18, 19, 20], unlike *functional scalability* which we refer to as the capability to accommodate growth in terms of the number of services composed in an IoT system (see Figure 3(c)). In particular, it enables the composition of any number of services, without severely impacting global system properties such as performance, maintenance, evolution and monitoring. Hence, functional scalability is crucial for dealing with IoT systems composed of billions of services. Figure 4 shows that it is orthogonal to vertical and horizontal scalability.

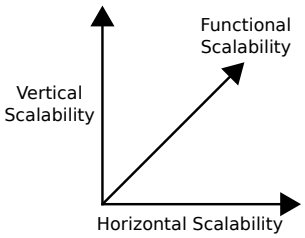


Figure 4: Scalability dimensions.

Like the other kinds of scalability, functional scalability requires the definition of metrics to measure the degree of satisfaction for accommodating new services. In this paper, we propose six qualitative metrics which we discuss in Section 5. We do not claim that such metrics are complete since quantitative metrics for QoS, identified for vertical and horizontal scalability, can also be important. However, quantitative metrics are only applicable to specific implementations. As service composition is an abstraction rather than a concrete implementation, we strongly argue that qualitative metrics are the best ones to measure the degree of satisfaction of functional scalability in service composition mechanisms. For the rest of the paper, the terms scalability and functional scalability are used interchangeably.

[2] IoT cloud environments benefit from dynamically scaling vertically, horizontally or both.

3. Related Work

This section presents the current IoT service composition mechanisms that define behaviour by workflows, namely (centralised and distributed) dataflows, (centralised and distributed) orchestration, choreography, and a novel composition mechanism called DX-MAN. We particularly focus on the fundamental semantics of these mechanisms instead of addressing specific implementation technologies. This is because semantics constitutes general theory that defines how to compose services conceptually rather than a concrete implementation (that can only be evaluated in specific scenarios). Significantly, fundamental semantics underlies so-called composition algorithms [8, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102], programming frameworks [84, 103, 104, 105, 106, 107, 108], languages [109, 110] and platforms [111, 112, 113, 114, 115, 116, 117], which have been somehow confusingly included in existing "IoT service composition" surveys [3, 5, 24, 118, 119].

It is also essential to mention that IoT service composition is just another name for traditional SOA composition and it is done regardless of so-called service "architectures" such as the ones defined for Microservices. Microservice Architecture [120, 121, 122] has gained considerable attention in the last few years and is becoming increasingly important and popular for the development of IoT systems [123, 124, 125, 126, 127]. Every Microservice Architecture is an SOA, but not the other way round [128]. Hence, the service composition mechanisms presented in this sub-section can be used interchangeably in both Microservices and traditional SOA services. In fact, there are no composition mechanisms specifically aimed for Microservices.

Dataflows, or *Flow-Based Programming* [129, 130, 131], is a composition mechanism that defines implicit workflows as directed graphs where vertices receive input data streams, carry out some computation and pass the result to other vertices via an edge for further processing [71, 132]. A centralised approach [83, 116] fully coordinates the exchange of data streams exogenously (i.e., from outside services), whereas a distributed one [37, 38, 133, 134] partitions a complex workflow dataflow over multiple coordinators. Currently, dataflows is the most popular IoT service composition mechanism, so there are plenty of available technologies for defining data-driven IoT workflows [83, 126, 135].

Orchestration defines explicit workflow control flow structures to coordinate the invocation of service operations exogenously and, like dataflows, it can be centralised or distributed. A centralised orchestration [72, 136, 137, 138] has full control over all the services composed, whereas a distributed approach (also known as "decentralised orchestration" [139, 140, 141, 142, 143, 144, 145, 146, 147, 148]) defines sub-workflows for a collaborative exchange of workflow control flow over the network. Although orchestration is not as popular as dataflows, there are some implementations available for this mechanism to support the definition of control-driven IoT workflows [149, 150, 151].

Choreography [136, 137, 138, 152, 153] is another composition mechanism that defines workflow control flows for the invocation of operations in services. Unlike orchestration, it relies on a public protocol which specifies a global "service conversation" via decentralised interactions, and it is modelled using a choreography modelling language [122, 138, 153]. When a choreography is enacted, the composed services exchange control according to the protocol to realise decentralised workflows. It is because of this decentralised nature that there is an increasing trend of implementing technologies for choreographing IoT services [154, 155, 156]. Notably, some of these technologies (e.g., [155]) implement choreographies based on the data-driven paradigm with no notion of public protocols, which do not define any composite service but a bunch of interactions via events or decentralised message passing. So, they do not follow the standard definition as noted by [35].

DX-MAN [157, 158, 159] is a model that uses exogenous composition operators [160, 161] to algebraically compose IoT services in a hierarchical bottom-up manner. The result of composition is not a workflow, but an IoT composite service which is semantically equivalent to a potentially infinite family of (explicit) workflow control flows. As DX-MAN takes the best properties from choreography and orchestration, it enables decentralised data exchanges over the network while decoupling services via (exogenous) workflow control flows [162]. Currently, there is only one platform available to support the definition of algebraic IoT systems [163].

To continue the discussion, Section 6 describes the above composition mechanisms in detail and presents a concrete analysis in terms of functional scalability requirements.

4. A Large-Scale IoT Scenario: Smart Parking System (SPark)

To illustrate the context for scalability requirements, this section presents a large-scale IoT scenario in the smart parking domain. The scenario tackles a common problem of large cities and is described as follows.

With the increase of population, large cities have had to deal with daily traffic congestion caused by drivers actively searching for parking spaces, especially during rush hours. As a consequence, there are increased carbon emissions as well as waste of commuters' time and money [164, 165]. This section presents a large-scale smart parking system, *SPark*, for self-driving vehicles which efficiently find (and reserve) the nearest parking space in a smart city. *SPark* thus helps to improve parking space utilisation, shorten parking search time, reduce environmental pollution, minimise parking costs and fuel consumption, and alleviate traffic congestion [164].

SPark operates in a smart city with plenty of parking spaces equipped with occupancy sensors whose data is managed by Infostations. Although it is an urban infrastructure device able to collect up-to-date occupancy status from all sensors in range, an infostation only pulls data from the sensors near a vehicle. Figure 4 illustrates *SPark* with four self-driving cars, where a vehicle gets its location and requests a parking space from the nearest Infostation. The InfoStation then pulls data from the nearby sensors to determine the near-

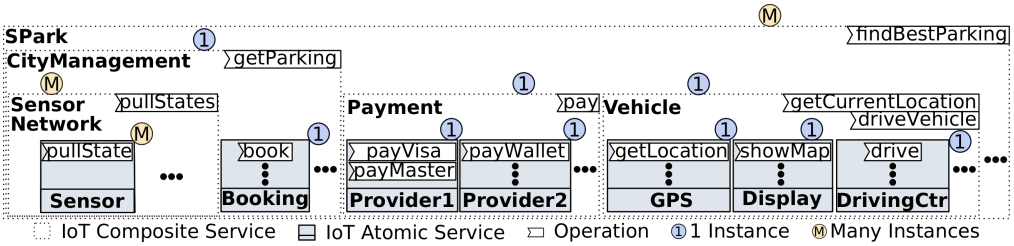


Figure 6: *SPark* services.

est parking space that is free. To avoid two different vehicles chasing the same parking space, the space is reserved and paid for in advance. Finally, the vehicle displays the desired route and drives towards the selected parking space. Figure 5 depicts the general workflow of SPark.

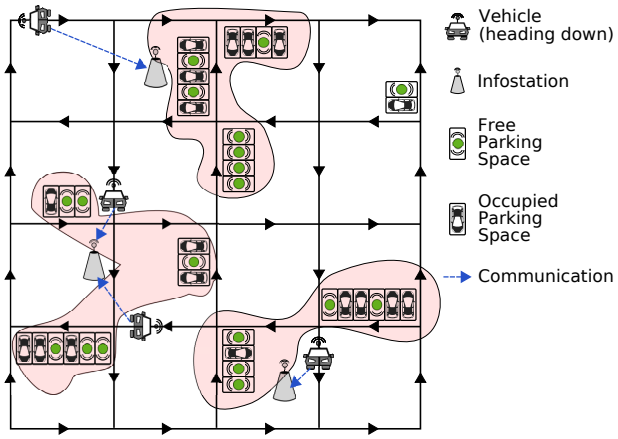


Figure 4: A smart city with *SPark*.

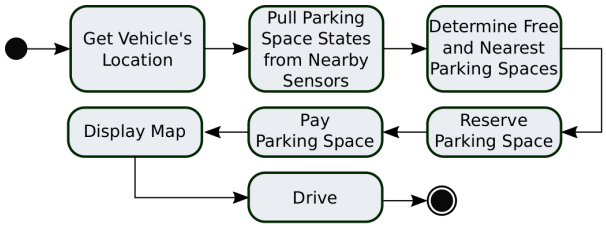


Figure 5: *SPark* workflow.

To achieve its goal, *SPark* composes a huge amount of IoT services distributed across a city. It is then an ultra large-scale system [23, 24, 25, 26] because sub-systems (i.e., services) may potentially integrate other sub-systems (i.e., services) and so on. Figure 6 depicts the relationship between services and sub-services in *SPark*. We distinguish between *atomic services* and *composite services*. An atomic service is the most primitive unit with no internal structure, whilst composite services integrate sub-services and can be integrated into even more complex composites.

To hide complexity and protect behaviour integrity, we assume that the composite services of our example are encapsulated. As IoT composite services potentially reside on different business domains [166], *SPark* cannot be "decomposed" using a top-down approach, but it should be composed in a bottom-up fashion by reusing existing compositions [27, 31]. Figure 6 shows the resulting hierarchical service relationship.

Note that ellipses indicate that services may provide multiple operations or compose many other services. Also note that, in practice, atomic services can be defined as composite services. For example, *DrivingCtr* could be a complex service that integrates services for planning a path and controlling a vehicle. As another example, the *Display* service may internally use a web mapping service and a visualisation service for displaying a route. An alternative (larger) view of *SPark* can be found in Appendix A. For clarity, this section only shows a simplified version of *SPark* which is incrementally described below.

The *SensorNetwork* composite is a wireless network that composes a group of dedicated atomic *Sensor* services. It provides the *SensorNetwork.pullStates* operation as an interface for collecting parking space status in parallel. The collection is done by invoking the *Sensor.pullState* operations from the sensors near the vehicle.

The *CityManagement* composite service composes multiple *SensorNetwork* composites and one atomic *Booking* service. It provides the *CityManagement.getParking* operation for finding and reserving the best (i.e., the free and nearest) parking space. To do so, it collects sensor states with the *SensorNetwork.pullStates* operation, and then determines which parking spaces are free. Finally, it reserves the nearest free parking space using the *Booking.book* operation.

The *Payment* composite is an online electronic payment service that composes two payment providers: *Provider1* and *Provider2*. It chooses which payment method to use when the operation *Payment.pay* is invoked. On the one hand, *Provider1* is an atomic service with the operations *payVisa* and *payMaster* (for paying with Visa or Mastercard). On the other hand, *Provider2* is an atomic service with the operation *payWallet* (for paying with an eWallet).

The *Vehicle* composite encompasses two different behaviours, and composes the atomic services *GPS*, *Display* and *DrivingCtr*. The *Vehicle.getCurrentLocation* operation is an interface for the behaviour of *GPS.getLocation*, which gets geospatial positioning information. The *Vehicle.driveVehicle* operation is more complex, as it invokes *Display.showMap* so as to plan, compute and visualise the route on the vehicle's display. Then, it executes *DrivingCtr.drive* to autonomously drive the vehicle towards the desired parking space.

SPark is the most complex composite service, since it composes the services *CityManagement*, *Payment* and *Vehicle*, and provides the *SPark.findBestParking* operation for the encapsulation of the whole system's behaviour. The behaviour is a workflow for the sequential invocation of *Vehicle.getCurrentLocation*, *CityManagement.getParking*, *Payment.pay* and *Vehicle.driveVehicle*. Appendix B describes the complete workflow of our scenario.

As the *SPark* workflow spans multiple administrative domains, our motivating example requires the distribution of services across different geographical locations. Table 1 shows a possible physical deployment configuration. In particular, *CityCouncilCloud* is a virtual thing where services *CityManagement*, *Payment* and *Booking* are deployed. *ParkingSpace* is a physical thing whose occupancy status is provided by the *Sensor* service. *Infostation* is a physical thing that collects data from sensors within a wireless range, via the *SensorNetwork* service. *ProviderServer1* and *ProviderServer2* are physical things of different payment provider companies, where services *Provider1* and *Provider2* are deployed. *Vehicle* is a physical thing that moves around the city, searching for a parking space, which provides the services *SPark*, *Vehicle*, *GPS*, *Display* and *DrivingCtr*. Note that deploying a composite service (e.g., *SPark*) does not necessarily mean that the composed services should be deployed on the same thing. This is because IoT services operate on different administrative domains.

Thing	Composite Services	Atomic Services
CityCouncilCloud	CityManagement, Payment	Booking
ParkingSpace		Sensor
Infostation	SensorNetwork	
ProviderServer1		Provider1
ProviderServer2		Provider2
Vehicle	SPark, Vehicle	GPS,Display, DrivingCtr

Table 1
SPark service distribution.

5. Functional Scalability Requirements

This section presents the functional scalability requirements of IoT systems in terms of *SPark*. These requirements were derived from an extensive review of the literature on large-scale IoT systems and serve as the foundation of our evaluation framework. The requirements are: (i) explicit control flow [29, 30]; (ii) distributed workflows [23, 31, 32]; (iii) location transparency [20, 33]; (iv) decentralised data flows [23, 32, 34]; (v) separation of control, data and computation [35, 36, 37, 38]; and (vi) workflow variability [23, 39, 40, 41].

5.1. Explicit Control Flow

Control flow defines the execution order of composed services [71], according to sequencing, branching or parallelising constructs [75]. It is explicit when there is a visible construct defining control flow in a service composition, whilst it is implicit when it needs to be inferred from the collaborative interaction of the composed services [71, 74, 138].[3] Figure 7 describes the difference between such concepts. The left side depicts visible constructs for the sequential execution of $A.op_{A1}$ and $B.op_{B1}$. Contrastingly, the right side does not show any control flow constructs, as the workflow logic is hardcoded in the computation of the composed services.

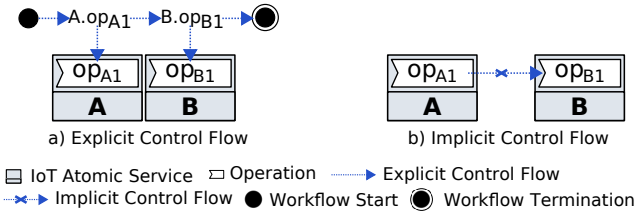


Figure 7: Explicit control flow vs implicit control flow.

The number of composed services may become overwhelmingly large because of both the huge number of things involved and the complexity of workflow control flow. Consequently, execution failures become unavoidable, more challenging to manage and may potentially unleash catastrophic consequences (for individuals or societies) [167]. For that reason, explicit control flow becomes crucial to tackle functional scalability, since it facilitates monitoring, tracking, verification, maintenance and evolution of large-scale IoT composite services [29, 30, 122, 168, 169]. For instance, we can leverage explicit control flow to detect abnormalities in a SPark execution [170, 171, 172], or we could easily obfuscate workflow control flow in order to avoid malicious reverse-engineering [173, 174].

Imagine that SPark suddenly stops working because of a bottleneck in some service, so developers want to analyse the system execution flow to find out where the problem is. For this, they can leverage explicit control flow to display a visual representation of the execution paths that SPark has taken [29, 74, 175, 176]. By looking at the blueprint, developers can identify the services that perform poorly and react accordingly.

Implicit control flow has historically been a barrier for functional scalability since it hinders control flow visualisation, especially when the number of services increases [29, 30, 121, 122, 177, 178]. Although there are attempts to visualise control flow [178, 179, 180, 181, 182], transforming implicit control flow into an explicit one is still a challenging task, especially when the system workflow is complex. This issue has made software development companies stop composing services with implicit control flow. Netflix [183] is the most prominent case, which has particularly expressed its concern for monitoring composite service executions when control flow is implicit. To tackle this, it has recently developed Conductor [29] to move from compositions with implicit control to compositions with explicit control.

[3] For example, control flow is explicit in imperative languages and implicit in declarative ones.

5.2. Location Transparency

Large-scale IoT systems are inherently dynamic and uncertain due to the presence of churn in the operating environment [20, 25, 27, 33, 184, 185, 186]. Churn means that things (and their services) dynamically connect and disconnect from the network as a result of auto-scaling, software upgrades, failures, poor connection and mobility. It is particularly evident when an IoT system uses resource-constrained things with a poor connection [187], or when there are mobile things involved [185, 188]. Hence, churn results in physical service locations (i.e., IP addresses) changing over time frequently. For example, the Spark's *Sensor* service is a resource-constrained thing with limited connectivity, which is likely to experience network disconnections. Similarly, the service *Vehicle* may change its IP address because of its high mobility. The same happens to *Provider1*, *Provider2* and *Booking* as they are deployed on the Cloud.

To deal with churn, IoT systems require location transparency, in order to ensure that atomic services are unaware of the physical location of other services [74]. Figure 8 illustrates this concept. In particular, the lower section shows that the (non-static) IP address of $B.op_{B1}$ is hardcoded in $A.op_{A1}$. The upper section shows a scenario where there is support for location transparency.

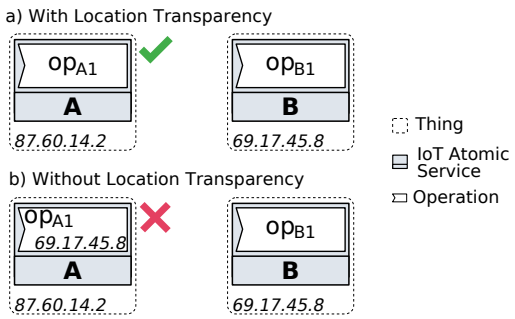


Figure 8: Location Transparency.

Location transparency is typically achieved with a service discovery mechanism that dynamically queries a central service registry [52]. In client-side discovery, service providers register at startup in a registry which is later queried by service consumers. In server-side discovery, a router (e.g., a service bus like a gateway [112]) queries the registry and forwards requests to an available service provider on behalf of service consumers.

A service composition with no location transparency requires service consumers to know the location of service providers in advance. For example, without any support for location transparency, *SPark* would have to be updated every time a *Vehicle* changes location. Nonetheless, this is worrying because there is a massive number of vehicles constantly moving around a city and, therefore, changing location. Addressing this issue with a service discovery mechanism entails the addition of "intrusive" elements to the composition (e.g., a service registry or a naming service) which are not part of the semantics of a composition mechanism. In fact, an atomic service computation would be tightly coupled with the "intrusive" element, since the former needs to be updated whenever the location of the latter changes. We shall keep this in mind in Section 6.

5.3. Distributed Workflows

IoT service composition can define a *centralised* or a *distributed* workflow. It is centralised when a single entity governs the entire workflow control flow. Contrastingly, it is distributed when multiple entities collaboratively define control flow [74]. Figure 9 illustrates the contrast between a centralised and a distributed workflow. The upper part shows a central composite service that defines a workflow for the sequential invocation of $A.op_{A1}$, $A.op_{B1}$, $A.op_{C1}$, $A.op_{D1}$ and $A.op_{E1}$. The lower part shows a possible distribution of the same workflow control flow over three composite services deployed on different things.

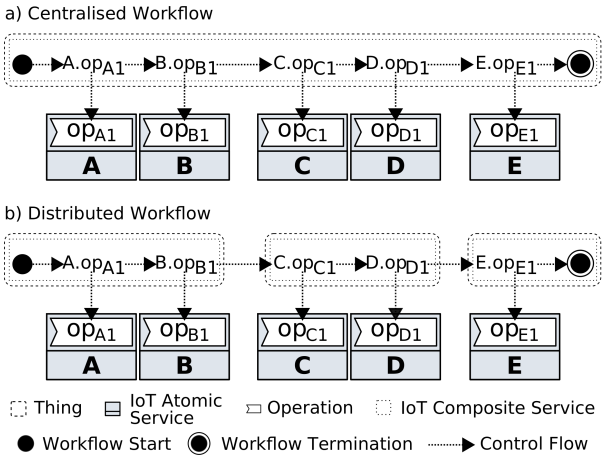


Figure 9: Centralised workflow vs distributed workflow.

Although a centralised workflow facilitates the design and maintenance of an IoT system, implementing a large-scale IoT system using such an approach is inefficient and infeasible [133]. This is because a central entity constitutes a single point of failure and attack, and leads to a performance bottleneck since all control and data goes through it [27, 34, 140, 142, 144, 148, 189, 190, 191]. Moreover, a central entity negatively impacts the availability of an IoT system.

The distributed nature of IoT requires control and data to pass through geographically dispersed entities (potentially deployed on different business domains) [32, 37, 133, 166]. For that reason, no single domain should govern the entire composition workflow, since multiple domains (perhaps in the order of millions) may want control over their own workflow part [74, 190, 192]. For instance, the *CityManagement* composite may potentially be controlled by the City Council, and a payment provider company could operate the *Payment* composite. This issue clearly implies that every administrative domain should be able to manage their own workflow definition. Thus, a *distributed workflow* should be part of any IoT service composition mechanism, since it allows interoperability between different domains and improves load balancing, throughput and availability of an IoT system [74, 87].

5.4. Decentralised Data Flows

Data flow defines the way data elements are moved from one service to another in a service composition [71]. This process can be done in three different ways: *centralised*, *distributed* or *decentralised*.

In the centralised approach [112, 136, 193, 194], a single composite mediates the exchange of data between services (see Figure 10(a)). Although it facilitates data management, the mediator becomes a potential bottleneck and introduces additional network hops for data exchange, since all data passes through it. This negatively impacts QoS by increasing the response time, and leads to network congestion [34, 141, 142, 190, 195], especially when there are plenty of IoT services exchanging huge amounts of data continuously [24, 25, 187, 190, 196].

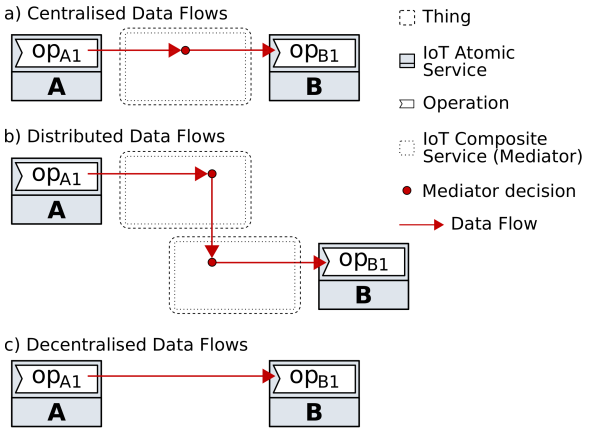


Figure 10: Data flow approaches.

To avoid a single bottleneck, the distributed approach [133, 139, 140, 141, 142, 143, 144, 146] removes the central composite and distributes the load of data over multiple composites (see Figure 10(b)). Although this improves load balancing, it introduces unnecessary network overhead as data passes through many mediators, even if data is unimportant for them, i.e., the more mediators, the more network overhead.

IoT services must exchange data as efficiently as possible (by minimising network hops), in order to avoid performance bottlenecks, achieve better response time and improve throughput [147]. A decentralised approach provides the most suitable data exchange for service composition since it requires only one network hop to pass data directly from a service producer to a service consumer (see Figure 10(c)) [34, 117, 152, 162, 195, 197, 198, 199]. For example, the data generated by *Booking.book* should be passed to *Display.showMap*, without passing through any other entity that does not require the produced data (e.g., the *CityManagement* composite or the *Vehicle* composite). In general, the amount of data transmitted in *SPark* may potentially be huge due to the large number of services involved of which sensors generate data continuously. Therefore, *SPark* requires a composition mechanism with support for decentralised data flows to provide an optimal QoS.

5.5. Separation of Control, Data and Computation

Large-scale IoT systems may potentially span multiple administrative domains and exhibit a high degree of heterogeneity in many different forms [25, 74, 200]. For instance, there may be different service providers (e.g., Amazon AWS IoT and IBM Watson), a wide variety of programming languages (e.g., Swift and embedded C), multiple operating systems (e.g., Contiki and TinyOS) and different network communication protocols (e.g., CoAP and MQTT). For that reason, service composition mechanisms must provide the means to deal with such heterogeneity, in order to compose large-scale cross-domain IoT systems. This is, in fact, one of the major challenges for building IoT systems [200].

To enable flexible service composition in such heterogeneous environments, control flow, data flow and computation should be orthogonal [35, 36, 38, 76, 144, 201, 202]. This would enable separate reasoning of concerns so system developers can focus on IoT service composition, while service developers focus on efficient service functionality [37, 74, 76]. Consequently, services are workflow agnostic because workflow control flow is never embedded in the computation of many services. Moreover, the separation of control and computation facilitates workflow monitoring [29].

Figure 11 illustrates a separation of control, data and computation. There are two computations defined in $A.opA1$ and $B.opB1$. The control flow part defines a workflow for the sequential invocation of $A.opA1$ and $B.opB1$, without considering data flow and computation. The data flow part defines intermediate processing of the output from $A.opA1$, before sending it to the input of $B.opB1$. When control is mixed with data, such processing represents an extra control flow step which is intrusive to the original workflow.

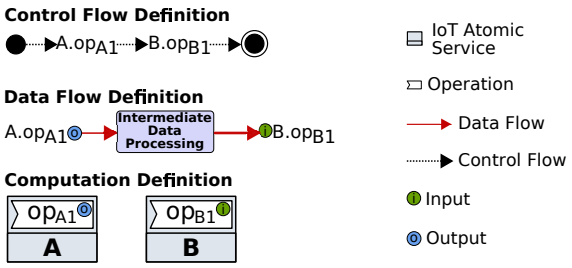


Figure 11: Separation of control, data and computation.

The separation of data flow will additionally enable separate data management for improving performance without considering control flow [35, 162]. Similarly, control flow can be used in isolation for defining efficient deployment strategies that do not consider data flow. Moreover, different technologies can be used for implementing control flow, data flow and computation separately [35]. For instance, developers of *SPark* could use CoAP for passing control between services, the Blockchain for managing data flows and embedded C for implementing service computation. Providing independent reasoning of concerns also enables a separate validation and verification (V & V) of services.

When control is mixed with computation, a service

provider and a service consumer are tightly coupled because invocations are originated in the consumer's computation. Thus, changing workflow control flow in one service requires further changes in the respective computation. This rigidity evidently limits the scalability of IoT systems because it is difficult to add, remove, change and replace services (and system workflow control flow). For that reason, reusing services is not possible at scale when control is mixed with computation [29, 37, 71, 74, 203].

Suppose there is a branching control flow structure in the computation of *Booking.book*, for invoking either *Provider1.payVisa* or *Provider2.payWallet*. As a result, *Booking.book* is not reusable as it is since it must be adapted to the requirements of other systems, e.g., it must be updated when two new payment providers come into play (e.g., *Provider3* and *Provider4*). As another example, SPark could need the parallel execution of *Payment.Pay* and *Vehicle.driveVehicle*, in order to improve concurrency. In this case, the sequence of invocations hardcoded in *SPark.findBestParking* must be replaced with a parallel control flow structure.

Generally speaking, large-scale IoT systems require separate reasoning of control, data and computation for independent maintenance, validation, verification, reuse and evolution of those concerns. This separation of concerns could also result in reduced time to market and reduced software production and maintenance costs [74].

5.6. Workflow Variability

Large-scale IoT systems operate in highly dynamic environments subjected to variability caused by external or internal factors [39, 41, 204, 205]. External factors are beyond the scope of the system and include changes in requirements and increasing workloads. Internal factors are associated with the system operation and include system failures and sub-optimal behaviours.

A service composition mechanism must support the definition of alternative behaviours, in order to adapt a composite to changes in both the external and the internal environment. As manually choosing alternative behaviours is a costly and inefficient management process, when there are many services in the composite, behaviours must be selected autonomously (i.e., with no human intervention) [206]. Thus, variability of behaviour is a crucial desideratum for the realisation of large-scale autonomous IoT systems [23, 39, 40, 41, 87, 207].

Workflow variability [157, 208] allows the definition of alternative control flow constructs (i.e., behaviours) in a service composition (see Figure 12), and it is particularly useful for autonomously changing workflows at runtime. For instance, the *SensorNetwork* composite may define variable parallel behaviours for pulling data, depending on a vehicle's location. For one vehicle, it may pull data from *Sensor1* and *Sensor2*, whilst for another one it could pull data from *Sensor10*, *Sensor29* and *Sensor34*. As another example, consider multiple payment service providers that offer different QoS (which is always fluctuating according to different workloads). For this, a variable branching construct can be used to dynamically choose the *Provider* service with the best QoS.

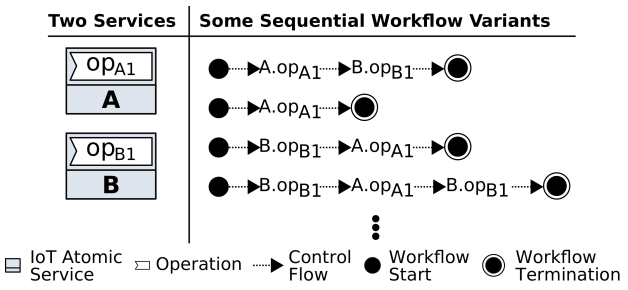


Figure 12: Workflow variability.

When there is a family of related compositions, a *Greatest Common Denominator* [208] is a base variability model that can be reused for defining alternative compositions (also known as configurations). For example, suppose a SPark vehicle can be either *manual* or *self-driving*. A manual vehicle requires the services *GPS* and *Display*, whilst a self-driving one uses the service *DrivingCtr* in addition. For this, the *Vehicle* composite of Figure 6 would be the *Greatest Common Denominator*, which can be reused for defining two alternative behaviours for two different vehicles.

Workflow variability is not only useful to accommodate the requirements of different vehicles, but also meaningful for different cities and users. Consider the case of Northern Ireland where many cities offer free on-street parking, and a few others (e.g., Belfast, Lisburn and Newry) impose a tariff [209]. Here, some *SPark* workflow variants may require a *Payment* composite, while others not. At the city scale, it would become very difficult to define workflows from scratch since many services would need to be changed and customised. Users can also require different workflows according to their needs, e.g., one user may require pre-payment, whereas another one may require post-payment. However, manually changing behaviour at runtime to accommodate different user requirements is infeasible. Thus, workflow variability represents a suitable solution for tackling the above scenarios.

In order to define workflow variability, a service composition mechanism must enable *total compositionality* by which all behaviours (e.g., operations) from the composed services are semantically available in a composite service [157, 159]. This concept contrasts with *partial compositionality* where only named and selected behaviours are semantically preserved, leading to a combinatorial explosion of behaviours as the number of services increases [159].[4] Figure 13 illustrates the dichotomy between total and partial compositionality. The lower section shows the preservation of operations $A.op_{A1}$ and $B.op_{B2}$ in the composite service, for the creation of a single workflow involving those operations. Remarkably, the upper section depicts a composite service that preserves all the operations from the sub-services, which means that alternative workflows can be created. For instance, we could define a sequential workflow for the invocation of $B.op_{B1}$

[4] For example, there are $2^k - 1$ possible scenarios for pulling sensor data, where k is the number of sensors.

and $A.op_{A2}$, or an alternative sequence for the execution of $A.op_{A1}$, $B.op_{B2}$ and $A.op_{A1}$. These behaviours are impossible to achieve with partial compositionality which allows the definition of only one fixed workflow at a time.

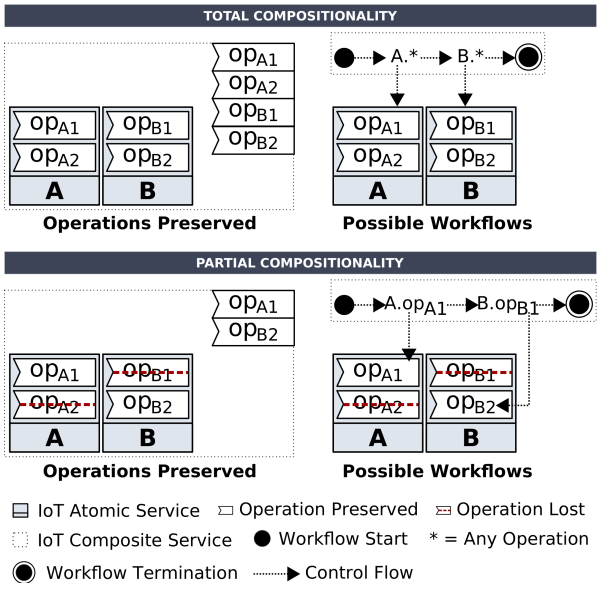


Figure 13: Total compositionality vs partial compositionality.

6. Analysis of IoT Service Composition Mechanisms

This section reviews and analyses the fundamental semantics of current IoT service composition mechanisms, namely (centralised and distributed) dataflows, (centralised and distributed) orchestration, choreography and DX-MAN. The goal is to determine how well they fulfil the functional scalability requirements of IoT systems. For this analysis, we investigate the following research questions per mechanism:

- **RQ1:** Is there any architectural entity explicitly defining control flow?
- **RQ2:** Are atomic services location-agnostic of other atomic services?
- **RQ3:** Is it possible to distribute a workflow over multiple entities?
- **RQ4:** Do atomic services exchange data directly?
- **RQ5:** Do atomic services perform computation without passing any data or control to other atomic services?
- **RQ6:** Is there any notion of workflow variants?

For *R1*-*R4* and *R6*, we use a tick mark to indicate that a specific mechanism fulfils the requirement being analysed, or a cross mark to indicate the opposite. *RQ5* is the only requirement that uses a textual representation for showing which concerns (i.e., control, data and computation) are independent. To answer *RQ6*, we need to determine the compositionality of a mechanism which can be either total or partial (see Section 5.6). To do so, we investigate the following research questions:

- **RQ7:** What is the resulting type from composition?
- **RQ8:** How many workflows does the composition mechanism define?

6.1. Dataflows

Dataflows, or *Flow-Based Programming* [129, 130, 131], is a composition mechanism that defines a workflow using data transformations (e.g., filter, split, union and sort) as well as exogenous data exchange between services [71, 132]. A dataflow description is a directed graph where vertices are asynchronous data processing units (invoking service operations), and edges are connections for passing data streams between vertices via the network (by message passing or events). A vertex explicitly defines input ports and output ports. When it receives data from all inputs, it performs some computation and writes results in output ports. The resulting data is then moved to other vertices via an edge. This process is illustrated in Figure 14.

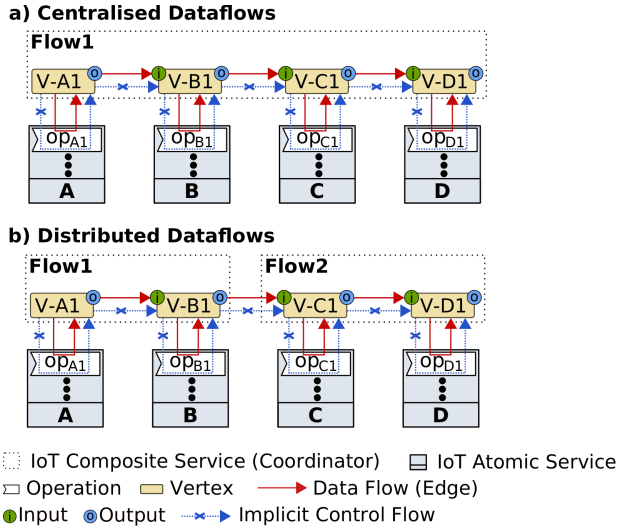


Figure 14: Composition by dataflows.

Dataflows can be centralised or distributed. A centralised dataflow [83, 116] defines a single coordinator for managing an entire graph and exogenously invokes service operations. A distributed dataflow [37, 38, 133, 134] partitions and distributes a complex graph over multiple coordinators that interact directly by exchanging data between vertices. Figure 14(a) shows a centralised dataflow for a pipeline of services *A*, *B*, *C* and *D*. When the coordinator *Flow1* is triggered, vertex *V-A1* invokes $A.op_{A1}$ and passes the result to the vertex *V-B1* which, in turn, executes $B.op_{B1}$. Next, the result of *V-B1* is passed to the vertex *V-C1* which executes $C.op_{C1}$. Finally, the data of *V-C1* is moved to the vertex *V-D1* and then processed by $D.op_{D1}$. A distributed version of the same pipeline

Composition Mechanism	Composition Unit	Resulting Type from Composition	Compositionality	Number of Workflows
Centralised Dataflows	IoT Service	Workflow	Partial	1
Distributed Dataflows	IoT Service	Workflow	Partial	1

Table 2
Compositionality of dataflows.

	Centralised Dataflows	Distributed Dataflows
Explicit Control Flow	✗	✗
Service Location Transparency	✓	✓
Distributed Workflows	✗	✓
Decentralised Data Flows	✗	✗
Separation of Control/Data/Computation	Data/Computation	Data/Computation
Workflow Variability	✗	✗

Table 3
Analysis of composition by dataflows w.r.t. the scalability desiderata.

is shown in Figure 14(b), where there is an edge between *V-B1* and *V-C1* for moving data from the *Flow1* composite to the *Flow2* composite.

Dataflows are increasingly popular for composing IoT systems. In particular, they are widely used for the Internet of Data (IoD) [210] which involves data collection from multiple sources (e.g., sensors), data analysis and control of the physical world. This paradigm has been referred to as Sense-Compute-Control (SCC) [211].[5] Currently, there are many platforms for composing IoT services using dataflows. Examples include Node-RED [83], COMPOSE [214], Glue.Things [116], LabVIEW [135], Paraimpu [215], Virtual Sensors [111], SpaceBrew [216], FogFlow [18], ASU VIPLE [217], ThingNet [126], Calvin [218], IoT Services Orchestration Layer [219], NoFlo [220] and many others [221, 222, 223, 224].

As the Web 2.0 became more data-centric and user-friendly [65, 71], dataflows have gained popularity for IoD through mashups. Mashups [225] are realised by dataflows [71, 226], and they allow the composition and visualisation of data streams on a graphical user interface displayed on the Web [65, 71, 227]. Examples of mashup tools include WoTKit [228], IoTMaaS [196] and Clickscript [56].

Dataflows have been accepted as coordination languages since a graph is defined in a coordinator that exogenously invokes services according to a dataflow description [129, 130, 133]. A dataflow graph is typically created with a graphical editor and executed by an engine (i.e., the coordinator), and it is triggered by either timing constraints or events.

A dataflow coordinator is the only entity aware of service locations, and provides separation between data and computation. This separation allows service developers to focus on data stream computation while system developers wire up vertices exogenously [133]. However, despite the aforementioned advantages, decentralised data flows are not supported as data streams always pass through data flow coordinators.

Although passing data between vertices is explicitly defined in a dataflow graph, control flow is implicit in the collaborative data stream exchange. This is because control flow statements are not visible in the graph specification.

In general, a graph specification is a single flat workflow of named and selected service operations. It might seem that a distributed dataflow provides multiple workflows. However, this is not true because a distributed dataflow graph is just a single nested workflow, distributed over different coordinators. Thus, dataflows only provide partial compositionality and do not support workflow variability (see Table 2).

Table 3 summarises the results of our analysis of data flows w.r.t. the scalability desiderata. Both centralised dataflows and distributed dataflows have similar characteristics. The only difference is that the latter provides support for distributed workflows by partitioning a dataflow graph over multiple coordinators.

6.2. Orchestration

Orchestration can be centralised or distributed. Centralised orchestration [72, 136, 137, 138] describes interactions between services from the perspective of a central coordinator (also known as orchestrator) which has control over all parties involved. It explicitly defines workflow control flow to coordinate the invocation of service operations, in order to realise some complex function that cannot be achieved by any individual service [69, 70, 229]. In a distributed orchestration, also known as "decentralised orchestration" [139, 140, 141, 142, 143, 144, 145, 146, 147, 148], multiple coordinators collaboratively define workflow control flow.

An orchestration is typically defined using a workflow language such as BPEL [70, 81, 230, 231, 232, 233, 234, 235] or BPMN [86, 236]. The resulting workflow has tasks for passing control among services according to explicit control flow constructs (for sequencing, parallelising, branching and

[5] Do not confuse data analysis tools [212, 213] with dataflows. A data analysis tool is a software that allows the collection, storing, indexing, processing, monitoring and visualisation of data. On the other hand, dataflows specify how data is passed between services according to a dataflow graph specification.

Composition Mechanism	Composition Unit	Resulting Type from Composition	Compositionality	Number of Workflows
Centralised Orchestration	IoT Service	Workflow	Partial	1
Distributed Orchestration	IoT Service	Workflow	Partial	1

Table 4
Compositionality of orchestration.

	Centralised Orchestration	Distributed Orchestration
Explicit Control Flow	✓	✓
Service Location Transparency	✓	✓
Distributed Workflows	✗	✓
Decentralised Data Flows	✗	✗
Separation of Control/Data/Computation	Control/Computation	Control/Computation
Workflow Variability	✗	✗

Table 5
Analysis of composition by orchestration w.r.t. the scalability desiderata.

looping) [71]. In distributed orchestration, the interaction between coordinators can be done in three different ways: (i) *with an extra task* [140], (ii) *with two extra tasks* [141, 142, 148] or (iii) *without any extra task* [139, 144, 145]. In (i), an orchestration invokes the interface of another one using an external task to the system's workflow control flow. In (ii), there are two different tasks for receiving and passing control (and data) between two orchestrations. Finally, in (iii), two orchestrations interact by moving control (and data) directly between the tasks of the system's workflow control flow.

An orchestration engine is responsible for executing a workflow process by invoking service operations in a given order. Although traditional engines can be used (e.g., Camunda BPM workflow engine [86, 237], Activiti [238, 239] and AWS Step Functions [151]), recently we have seen the emergence of orchestration engines particularly designed for IoT systems (e.g., PROtEUS [149] and [150]).[6]

Figure 15(a) illustrates a centralised orchestration for the services A, B, C and D, where the coordinator *Orch1* defines a "composite service" for the sequential invocation of $A.op_{A1}$, $B.op_{B1}$, $C.op_{C1}$ and $D.op_{D1}$. Three distributed versions are depicted in Figure 15(b). In the former, *Orch1* defines a "composite service" for the sequential execution of $A.op_{A1}$ and $B.op_{B1}$, and then uses an extra task to pass control (and data) to *Orch2*. *Orch2* defines another "composite service" to sequentially invoke $C.op_{C1}$ and $D.op_{D1}$. The second distributed version uses two extra tasks for passing and receiving control (and data) between *Orch1* and *Orch2*. Finally, in the last distributed version, control and data are passed directly from $B.op_{B1}$ to $C.op_{C1}$.

A glance at Figure 15 reveals that services are workflow agnostic because an orchestrator is the only entity aware of the location of other services. Having workflow agnostic services implies that an orchestrator provides separation between control flow and computation. This separation allows service developers to focus on efficient service functionality (i.e., computation) while system developers focus on workflow control flow in orchestrator(s).

Although a central coordinator facilitates the management of control flow logic, it easily becomes a performance

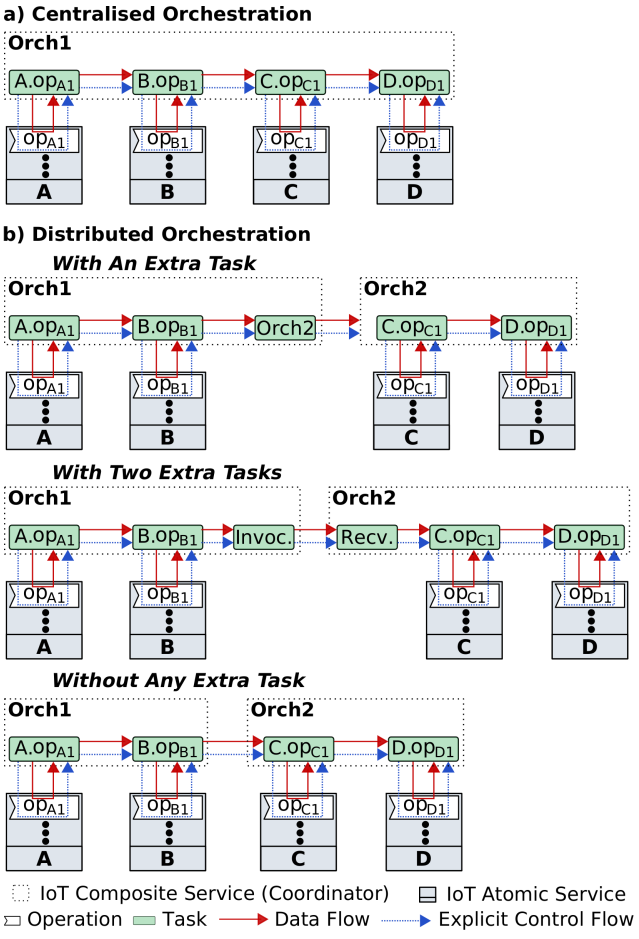


Figure 15: Composition by orchestration.

[6] A workflow engine can be deployed on either a specialised server [237] or a service bus like a Gateway [112, 193, 194].

Composition Mechanism	Composition Unit	Resulting Type from Composition	Compositionality	Number of Workflows
Choreography	IoT Service	Workflow	Partial	1

Table 6
Compositionality of choreography.

bottleneck because all data passes through it [140, 142]. This is because data follows control [71, 162]. Furthermore, the resulting composite service is a single flat workflow, for the invocation of named and selected operations, which must be transformed into a service [159, 240]. For that reason, orchestration only provides partial compositionality and does not support workflow variability (see Table 4).

Table 5 summarises the results of our analysis of orchestration w.r.t. the scalability desiderata, where we can see that centralised orchestration and distributed orchestration have similar characteristics. The only difference is that the latter supports distributed workflows via the partition of workflow control flow over multiple coordinators.

6.3. Choreography

A choreography describes service interactions from a global perspective using a public contract (also known as protocol) [136, 137, 138, 152, 153]. The contract specifies a "conversation" among participants via decentralised message exchanges, which can be modelled by a global observer using a choreography modelling language [122, 138, 153].[7] An interaction-based model allows the definition of event-driven or request-response messages by connecting required and provided interfaces. Examples include WS-CDL [241] and Let's Dance [242]. An interconnected interface model, on the other hand, allows the specification of control flow per participant. Examples include BPEL4Chor [240], Web Service Choreography Interface (WSCI) [243] and BPMN [244].[8]

A protocol defines roles for the collaborative realisation of a global workflow with no control over the internal details of the participants involved [229]. A role explicitly describes a participant workflow control flow in terms of expected and produced messages. When a concrete service instance plays a role, it must behave accordingly by exchanging messages with other instances, using either direct message passing (e.g., invoking REST APIs) or events [63, 74, 117, 122].[9] This process is known as choreography enactment.

IoT is moving towards a more decentralised environment to reduce the bottleneck caused by centralised environments. As choreographies represent a natural way of dealing with such decentralisation, there are currently some platforms for composing IoT services by choreographies, e.g., CHOReVOLUTION [154], ChorSystem [246], Actorsphere [156], BeC^3 [117] and TraDE [35].

Figure 16 illustrates a sequential choreography for the services A, B, C and D, where a protocol (defined with standard BPMN 2.0 notation [236]) specifies that $B.op_{B1}$ expects a message from $A.op_{A1}$, $C.op_{C1}$ a message from $B.op_{B1}$ and $D.op_{D1}$ a message from $C.op_{C1}$. When the choreography is enacted, there is a chain reaction that starts with the invocation of $A.op_{A1}$ and finishes with the execution of $D.op_{D1}$.

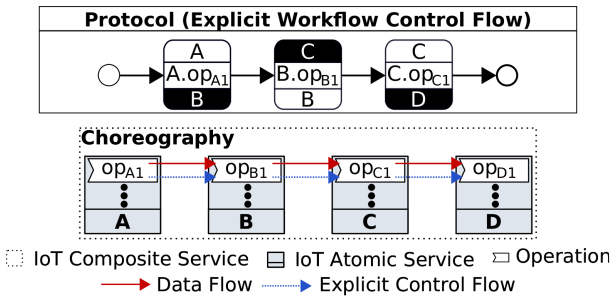


Figure 16: Composition by choreography.

Figure 16 shows that workflow control flow is explicitly defined in the protocol. During enactment, control is passed alongside data in every invocation, so services need to be aware of the location of other services. Of course, a service registry can be used, but this entails adding an "intrusive" element external to the composition (see Section 5.2).

Like orchestration, the resulting composition is a flat workflow for the invocation of selected and named operations (specified in the protocol). If the resulting workflow needs to be further composed, a choreography needs to be transformed into a service [159, 240]. For that reason, a choreography is partially compositional and workflow variability is not supported (see Table 6). Table 7 summarises the results of our analysis of choreography w.r.t. the scalability desiderata.

	Choreography
Explicit Control Flow	✓
Service Location Transparency	✗
Distributed Workflows	✓
Decentralised Data Flows	✓
Separation of Control/Data/Computation	None
Workflow Variability	✗

Table 7
Analysis of composition by choreography w.r.t. the scalability desiderata.

[7] A Microservice architecture prefers choreography over orchestration to support decentralised workflows [122].

[8] The choice of the contract depends on the type of participants involved which can be either atomic services or orchestrations.

[9] Service participants are tightly coupled in terms of dependencies. In choreographies based on direct message-passing, services hardcode invocation calls in service computation. In event-driven choreographies, services are tightly coupled because senders and receivers agree a topic queue in advance [245].

6.4. DX-MAN

DX-MAN [157, 158, 159] is an algebraic model where IoT services and service composition operators are first-class semantic entities. A service is a stateless distributed unit of composition which can be either atomic or composite, and its semantics is a workflow space (i.e., a family of workflow variants). A *composition operator* defines variable control flows between families of workflow variants (see Figure 17(b)).

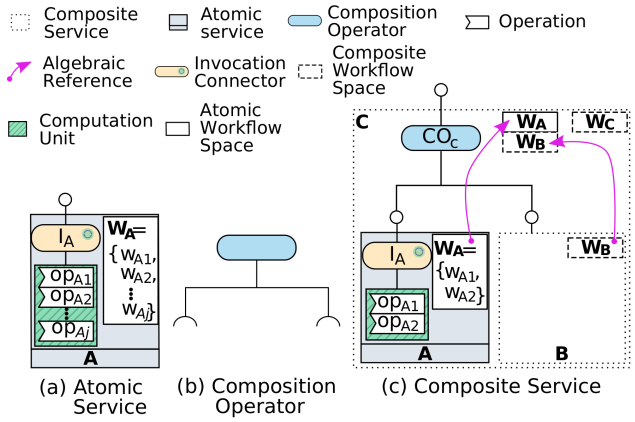


Figure 17: DX-MAN model.

An *atomic service* is formed by connecting an *invocation connector* with a *computation unit* (see Figure 17(a)). It is a finite workflow space whose elements invoke a different operation implemented in the computation unit via an invocation connector. As a computation unit is semantically identical to a traditional SOA service (because it is a set of operations), DX-MAN atomic services and SOA atomic services are semantically different (cf. Section 2.1).

A *composite service* connects a composition operator with multiple (atomic or composite) sub-services (see Figure 17(c)), which is equivalent to connecting a composition operator with multiple sub-workflow spaces. The result is a composite workflow space whose workflow variants invoke elements of atomic sub-workflow spaces and/or entire composite sub-workflow spaces, according to the control flow definition of the composition operator being used. There are composition operators for sequencing (i.e., *sequencer*), branching (i.e., *inclusive selector* and *exclusive selector*) and parallelism (i.e., *paralleliser*). Figure 18 shows that the sequencer and paralleliser operators define infinite workflow variants, whilst the branching operators define $2^n - 1$ variants s.t. n is the total number of atomic sub-service operations plus the number of composite sub-workflow spaces. For further details on this matter, see [157].

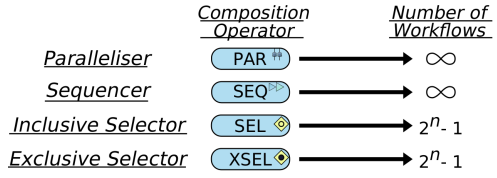


Figure 18: DX-MAN composition operators.

In addition to composition operators, DX-MAN provides special transformation operators (called adapters) for sequencing, branching, parallelising, looping and guarding over individual workflow spaces. Hence, DX-MAN is Turing complete [247]. Currently, there is only one platform that implements the DX-MAN model [163], and it is available at `https://github.com/damianarellanes/dxman`.

In DX-MAN, a composition is done incrementally in a bottom-up fashion. So, a hierarchical connection structure of operators sits on top of atomic services. Figure 19(a) shows a DX-MAN composition that involves four atomic services and three composite services. The first step is to model the atomic services A, B, C and D with the invocation connectors I_A, I_B, I_C and I_D. This process results in the atomic workflow spaces W_A, W_B, W_C and W_D, respectively. In the next hierarchy level, we create the composite services E and F. To do so, we use the composition operator SEQ_E to define a composite workflow space W_E from the atomic sub-workflow spaces W_A and W_B. Likewise, we define the composite workflow space W_F using the operator SEQ_F which operates on the atomic sub-workflow spaces W_C and W_D . Finally, we create the top-level composite service G. For this, we use the composition operator SEQ_G which defines the composite workflow space W_G from W_E and W_F.

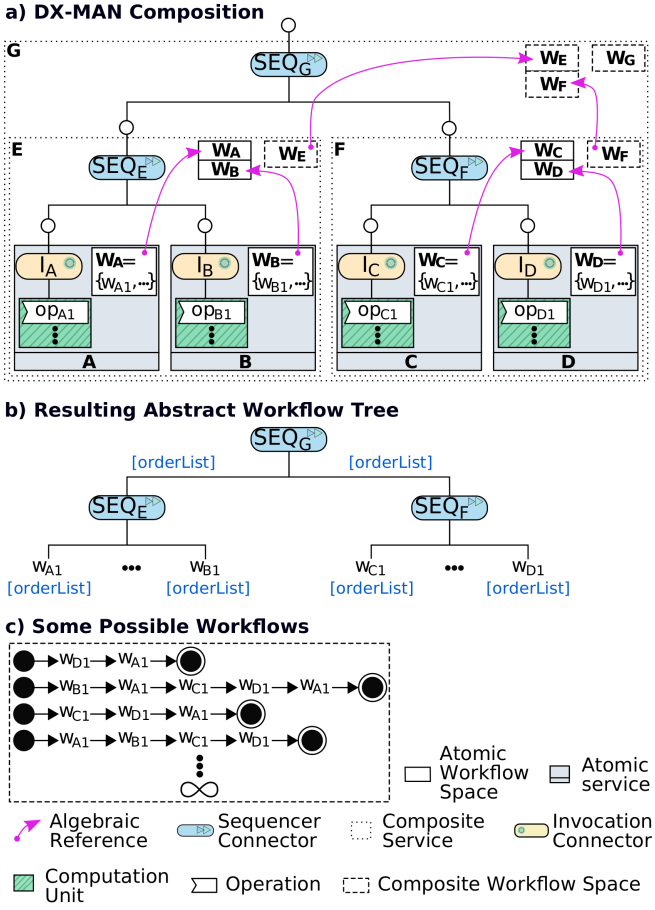


Figure 19: Composition by DX-MAN.

Note that workflow spaces from sub-services are available

Composition Mechanism	Composition Unit	Resulting Type from Composition	Compositionality	Number of Workflows
DX-MAN	IoT Service	IoT Service	Total	[1,∞]

Table 8
Compositionality of DX-MAN.

in the next hierarchy level thanks to the algebraic semantics of the model. Consequently, the result of algebraic composition is not a single workflow, but a (potentially infinite) workflow space (i.e., a service) of workflow variants. Thus, DX-MAN supports total compositionality (see Table 8).

When a composition is done, an abstract workflow tree is automatically derived, which represents the hierarchical control flow structure of the composite (see Figure 19(b)). To select a particular variant, a concrete workflow tree must be created, which is just a projection function over a composite workflow space. See [157, 158] for further details.

Figure 19(b) shows the abstract workflow tree of our example from which we can choose infinite sequential workflow variants (see Figure 19(c)). One case is depicted in Figure 20, where a concrete workflow tree is created for the sequential execution of $A.op_{A1}$, $B.op_{B1}$, $C.op_{C1}$ and $D.op_{D1}$. Figure 21 shows another variant where a different concrete workflow tree is defined to invoke $A.op_{A1}$ and $D.op_{D1}$ sequentially.

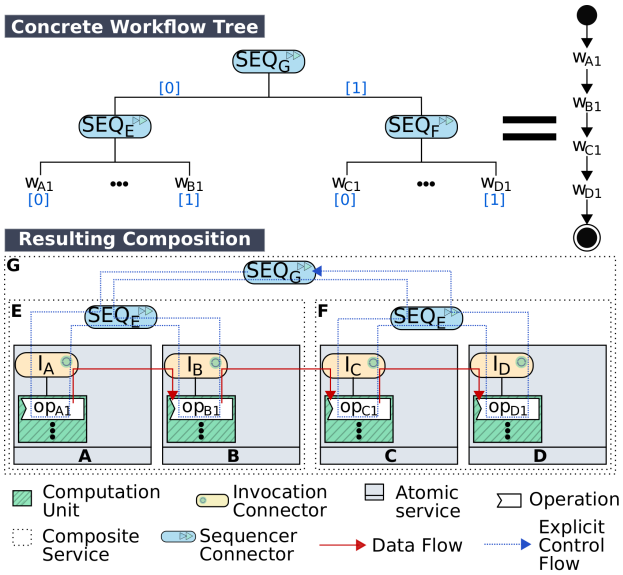


Figure 20: A workflow variant derived from Figure 19 for sequentially executing $A.op_{A1}$, $B.op_{B1}$, $C.op_{C1}$ and $D.op_{D1}$.

Note that a DX-MAN composition has composition operators for the coordination of workflow control flow only. This is because data flow and control flow are modelled separately for each workflow variant. The paper [162] describes further details on how to define data flows per variant.

A glance at Figures 20 and 21 reveals that data, control and computation are independent concerns. Thus, data is exchanged in a P2P fashion between atomic services, while composition operators coordinate workflow execution by passing control only. The coordination can be done in a distributed

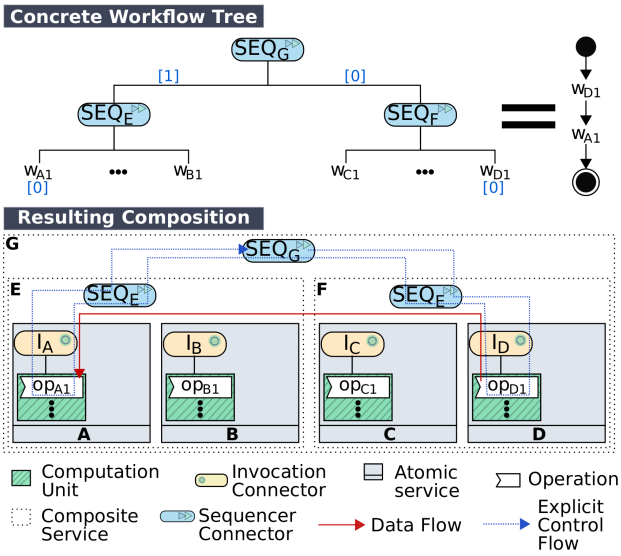


Figure 21: A workflow variant derived from Figure 19 for sequentially executing $D.op_{D1}$ and $A.op_{A1}$.

way since composition operators can be deployed on different things [159]. As coordination happens from outside services, computation units do not interact with one another, which results in independent distributed computations.

When an invocation connector receives control, it reads data from a decentralised data space (i.e., the Blockchain in the current implementation), invokes a service operation and writes results in the space. For that reason, only invocation connectors know the location of the connected computation units (i.e., service implementations). Furthermore, as invocation connectors perform operations on the data space, composition operators never exchange data during workflow execution. This (transparent) decentralised data exchange is achieved by the separation of control and data [162].

Table 9 summarises the results of our analysis of DX-MAN w.r.t. the scalability desiderata.

	DX-MAN
Explicit Control Flow	✓
Service Location Transparency	✓
Distributed Workflows	✓
Decentralised Data Flows	✓
Separation of Control/Data/Computation	Control/Data/ Computation
Workflow Variability	✓

Table 9
Analysis of composition by DX-MAN w.r.t. the scalability desiderata.

Composition Mechanism	Composition Unit	Resulting Type from Composition	Compositionality	Number of Workflows
Centralised Dataflows	IoT Service	Workflow	Partial	1
Distributed Dataflows	IoT Service	Workflow	Partial	1
Centralised Orchestration	IoT Service	Workflow	Partial	1
Distributed Orchestration	IoT Service	Workflow	Partial	1
Choreography	IoT Service	Workflow	Partial	1
DX-MAN	IoT Service	IoT Service	Total	[1,∞]

Table 10
Compositionality of IoT composition mechanisms.

	Centralised Dataflows	Distributed Dataflows	Centralised Orchestration	Distributed Orchestration	Choreography	DX-MAN
Explicit Control Flow	✗	✗	✓	✓	✓	✓
Service Location Transparency	✓	✓	✓	✓	✗	✓
Distributed Workflows	✗	✓	✗	✓	✓	✓
Decentralised Data Flows	✗	✗	✗	✗	✓	✓
Separation of Control/Data/Computation	Data/Computation	Data/Computation	Control/Computation	Control/Computation	None	Control/Data/ Computation
Workflow Variability	✗	✗	✗	✗	✗	✓

Table 11
Analysis of scalability desiderata of IoT composition mechanisms.

7. Evaluation

This section presents an evaluation of service composition mechanisms w.r.t. compositionality and the functional scalability requirements of IoT systems.

Table 10 summarises the analysis on compositionality presented in Section 6 to answer the research questions *RQ7* and *RQ8*. It shows that DX-MAN is the only mechanism that enables total compositionality since algebraic composition yields a service with a potentially infinite number of workflow variants. The other mechanisms define only one workflow at a time as the composition result is not a service, but a single flat workflow that invokes selected and named operations.

Research questions *RQ1-RQ6* have also been studied in Section 6. They enable us to analyse how well service composition mechanisms fulfil the scalability requirements: (i) explicit control flow; (ii) location transparency; (iii) distributed workflows; (iv) decentralised data flows; (v) separation of control, data and computation; and (vi) workflow variability.

Requirements (i), (ii), (iii), (iv) and (vi) are binary because they can be either supported (i.e., a tick mark) or not supported (i.e., a cross mark). Accordingly, we use Equation 1 to determine the satisfaction degree of binary requirements for a specific composition mechanism,

$$r_b(b) = \frac{b}{5} \tag{1}$$

where r_b is the satisfaction degree of binary requirements and b is the number of supported binary requirements by the mechanism s.t. $b \in (\mathbb{N} \cap [0, 5])$ and $r_b \in (\mathbb{Q} \cap [0, 1])$.

The requirement (v) is quinary since it admits $2^3 - 3$ possible results: *None*, *Control/Data*, *Control/Computation*, *Data/Computation* and *Control/Data/Computation*. This is because (v) considers three different concerns (i.e., control, data and computation) and discards options involving only one concern. When (v) is *None*, the requirement support becomes zero. Accordingly, Equation 2 determines the degree of separation of concerns for a specific composition mechanism,

$$r_c(c) = \frac{c}{3} \tag{2}$$

where r_c is the degree of separation of concerns and c is the number of independent concerns supported by the mechanism s.t. $c \in (\mathbb{N} \cap \{0, 2, 3\})$ and $r_c \in (\mathbb{Q} \cap [0, 1])$.

To determine the satisfaction degree of a composition mechanism w.r.t. scalability desiderata, Equations 1 and 2 are used in Equation 3. The result is a percentage that takes into account five binary requirements and one quinary requirement. A higher percentage means a higher satisfaction.

$$s(r_b, r_c) = (r_b \times \frac{5}{6} + r_c \times \frac{1}{6}) \times 100 \tag{3}$$

where s is the overall satisfaction degree w.r.t. all scalability requirements s.t. $s \in [0, 100]$.

Table 11 summarises our analysis of scalability desiderata and Table 12 shows the respective satisfaction degrees. Our interpretation of the results is presented below.

Centralised dataflows is the worst mechanism because it supports only one binary requirement (i.e., location transparency) and separates two concerns (i.e., data and computation). This means that the satisfaction degree of binary requirements is 0.20, while the satisfaction degree of separation of concerns is $0.\overline{66}$. Thus, the overall satisfaction degree

	b	c	r_b	r_c	s
Centralised Dataflows	1	2	0.20	$0.\overline{66}$	27.78%
Distributed Dataflows	2	2	0.40	$0.\overline{66}$	44.44%
Centralised Orchestration	2	2	0.40	$0.\overline{66}$	44.44%
Distributed Orchestration	3	2	0.60	$0.\overline{66}$	61.11%
Choreography	3	0	0.60	0.00	50.00%
DX-MAN	5	3	1.00	1.00	100.00%

Table 12
Satisfaction degree of IoT composition mechanisms w.r.t. scalabiility desiderata.

of centralised dataflows is 27.78%. Distributed dataflows provide distributed workflows in addition. Consequently, it possesses a (higher) satisfaction degree of binary requirements equal to 0.40 and, therefore, a (higher) overall satisfaction degree of 44.44%.

Although centralised orchestration has the same satisfaction degree as distributed dataflows, it supports different binary requirements (i.e., explicit control flow and location transparency) and separates different concerns (i.e., control and computation). Distributed orchestration is similar, yet different. It offers distributed workflows in addition for a (higher) satisfaction of binary requirements of 0.60 and, therefore, a (higher) overall satisfaction degree of 61.11%.

Choreography covers three binary requirements (i.e., explicit control flow, distributed workflows and decentralised data flows) and does not provide any separation of concerns since control and data are mixed in service computation. This means that the satisfaction degree of binary requirements is 0.60, with a null separation of concerns. Overall, choreography fulfils scalability requirements to a degree of 50%.

DX-MAN is the only mechanism that fulfils all binary requirements and provides the separation of control, data and computation. It is also the only one that supports workflow variability because of total compositionality (see Table 10). Accordingly, the satisfaction degrees of binary requirements and the separation of concerns are both 1. Thus, DX-MAN best fulfils the desiderata with a satisfaction degree of 100%.

8. Discussion

This section discusses the results presented in Section 7 and covers additional issues concerning compositionality, scalability requirements and the relationship between them.

Our results show that explicit control flow is supported by the majority of the mechanisms. In particular, orchestration defines control flow in orchestrators, choreography in a protocol and DX-MAN in composition operators. Dataflows is the only one that does no support such a requirement.

Workflow distribution is also meet by the majority of the mechanisms, except centralised dataflows and centralised orchestration where a coordinator fully governs a workflow.

Almost all composition mechanisms use a coordinator to exogenously define workflow(s) and, therefore, ensuring the separation of at least two concerns. In particular, orchestration and DX-MAN separate control and computation, while dataflows orthogonalises data and computation. Such separation enables location transparency as only coordinators are aware of atomic service locations. Choreography is the only mechanism without any support for location transparency, since control and data are mixed with computation.

There is a special choreography implementation based on the data-driven paradigm in which some computation is triggered once input data becomes available [199]. As a protocol must explicitly define workflow control flow, the analysis presented in Section 6 is also applicable.

Generally speaking, avoiding coordinators allows choreography styles to support decentralised data flows. So, there is a trade-off between coordination and decentralised data flows. DX-MAN obviates this problem by separating data in addition to control and computation. This separation enables the realisation of decentralised data flows, without considering neither control nor computation. So, data never passes through composition operators (i.e., coordinators) [162].

Some orchestration approaches [34, 146, 195, 198] partially separate control from data algorithmically. To achieve this, coordinators pass data references alongside control, rather than exchanging data values. However, analysing data dependencies to extract references is a challenging task because data and control are still semantically entangled.

For orchestration and dataflows, a coordinator is commonly referred to as a composite service. However, it is just a composition of specific operations (i.e., a workflow) that must be transformed into a service, rather than a composition of entire services (with all their provided operations). Only DX-MAN achieves an actual composition of services (not operations) as it defines a composite service without any transformation step while preserving all service operations from which multiple workflow variants can be derived. Thus, algebraic composition equates to total compositionality which, in turn, implies workflow variability. A DX-MAN composite service is in fact semantically equivalent to a (potentially infinite) set of orchestrations. This is because each element in a composite workflow space is a different composition workflow as regarded by existing mechanisms. For any existing composition mechanism to be equivalent to a composite workflow space, it would require to include all possible combinations of execution paths in the workflow definition, leading to a combinatorial explosion problem. While it is true a configurable workflow [208] can deal with this issue, it does not define multiple workflows at a time but just a single workflow that can be manually configured.

In this paper, we analyse the fundamental semantics of current service composition mechanisms that allow the definition of IoT workflows. However, other mechanisms must be considered even if they do not allow the explicit definition of behaviour, e.g., port connections [248] and ensemble-based composition [249]. Moreover, we did not analyse service interactions where there is no definition of composite services such as direct message passing [250], broker-based interactions [112] and event-driven interactions [251]. For this, we

refer the reader to another paper [74].

Different composition mechanisms can coexist in the same system. For example, it is possible to build a system where some services are composed using orchestration while others are composed by choreography [155, 252]. Our intention is not to analyse the combination of different composition mechanisms, but to analyse them individually. Clearly, our analysis results apply to such a combination.

9. Conclusions

Functional scalability becomes a crucial concern for IoT systems as the number of available services increases. For that reason, we need to look back at the foundations of service composition in order to tackle this challenging problem. In this paper, we analysed the semantics of current IoT service composition mechanisms, using an evaluation framework that considers six functional scalability requirements: (i) explicit control flow; (ii) distributed workflows; (iii) location transparency; (iv) decentralised data flows; (v) separation of control, data and computation; and (vi) workflow variability. Our results suggest that DX-MAN is the composition mechanism that best fulfils the functional scalability requirements of IoT systems. This is not surprising since such a model was designed with scalability desiderata in mind.

It is important to note that, except DX-MAN, there are no new composition mechanisms developed in the last decade. This crisis is a worrying situation that must be prioritised and remedied because of the inherent scale that IoT systems pose. Only DX-MAN was designed with this in mind, but we expect further developments on this in the coming years.

We do not claim that the scalability requirements we analyse here are complete, as other characteristics must also be considered. Nevertheless, we believe our evaluation framework has included the most critical ones and provides a useful starting point for further extensions that consider other functional scalability desiderata such as dynamic reconfiguration, the number of messages exchanged and the support for implicit/explicit data flows. The framework can thus be refined by other researchers when conducting further studies on IoT service composition mechanisms.

Appendix A

A larger view of *SPark* is shown in Figure A.1, where there are new composite services representing *Private Parkings* of a smart city. Also, the *Display* service has been converted into a composite that uses the atomic services *WebMap* and *Visual*. There is also a new composite service called *PathPlanning* that composes multiple *TrafficMonitor* composites and multiple atomic *People* services. In particular, a *TrafficMonitor* composite has multiple atomic services that represent congestion sensors and cameras (distributed across a city). Furthermore, the *Vehicle* composite has been endowed with multiple atomic *ChargSt* services (for charging the battery of a self-driving vehicle). Finally, the *DrivingCtr* service has been converted into a composite service with the atomic services *Brake* and *Steering*.

It is important to note that the larger view shown in Figure A.1 can be expanded into an even larger composition which may become infeasible to depict in a single document.

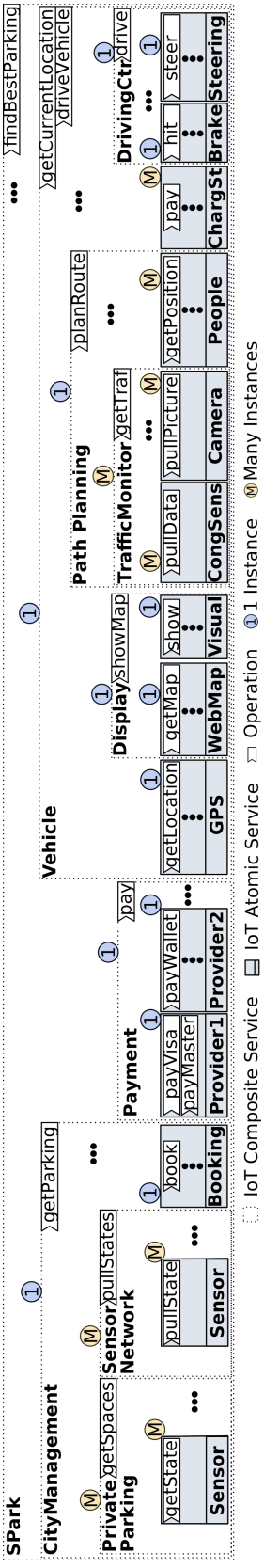


Figure A.1: A larger view of *SPark* services.

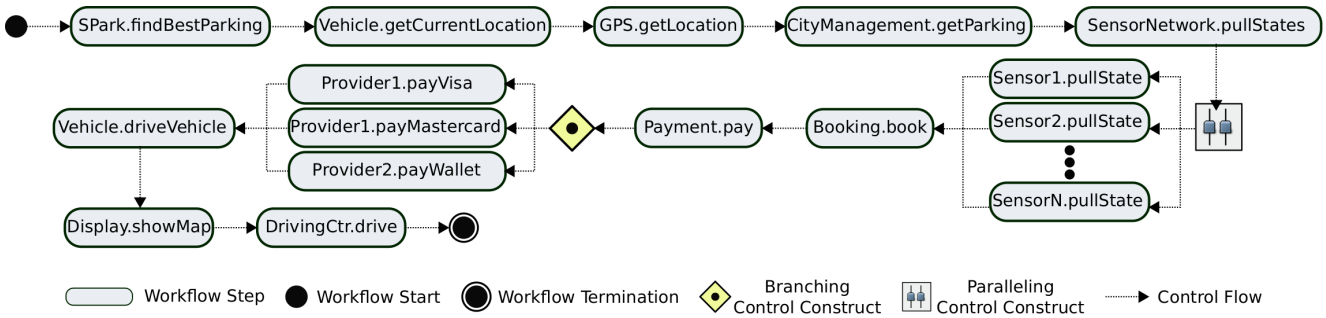


Figure B.1: *SPark* workflow control flow.

Appendix B

Figure B.1 illustrates the complete workflow of *SPark* which starts with the on-demand execution of *SPark.findBestParking*. Next, the *SPark* composite gets the current vehicle's location by invoking the operation *Vehicle.getCurrentLocation* which, in turn, invokes *GPS.getLocation*. Then, it invokes the *CityManagement.getParking* operation which internally executes the *SensorNetwork.pullStates* operation from the nearest InfoStation. In particular, *SensorNetwork.pullStates* pulls the states (in parallel) from the sensors close to the vehicle's location, using the respective *Sensor.pullState* operations. Next, the *CityManagement.getParking* operation uses the sensor states to determine the best (i.e., the free and nearest) parking space and then reserves that parking space using *Booking.book*. Then, control is passed to the *Payment.pay* operation which decides to invoke *Provider1.payVisa*, *Provider1.payMastercard* or *Provider2.payWallet*. Finally, the *Vehicle.driveVehicle* is invoked for implicitly calling *Display.showMap* and *DrivingCtr.drive*, in that order.

References

[1] IoT Analytics. State of the IoT 2018: Number of IoT devices now at 7b – Market accelerating, 2018. URL https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/.

[2] Chayan Sarkar, Akshay Uttama Nambi S. N, R. Venkatesha Prasad, Abdur Rahim, Ricardo Neisse, and Gianmarco Baldini. DIAT: A Scalable Distributed Architecture for IoT. *IEEE Internet of Things Journal*, 2(3):230–239, 2015.

[3] Marzieh Hamzei and Nima Jafari Navimipour. Toward Efficient Service Composition Techniques in the Internet of Things. *IEEE Internet of Things Journal*, 5(5):3774–3787, 2018.

[4] Asrin Vakili and Nima Jafari Navimipour. Comprehensive and systematic review of the service composition mechanisms in the cloud environments. *Journal of Network and Computer Applications*, 81: 24–36, 2017.

[5] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Service composition approaches in IoT: A systematic review. *Journal of Network and Computer Applications*, 120:61–77, 2018.

[6] Steen Maarten Van and Andrew S. Tanenbaum. *Distributed Systems*. CreateSpace, London, UK, 3rd edition, 2017.

[7] Xiang Sun and Nirwan Ansari. EdgeIoT: Mobile Edge Computing for the Internet of Things. *IEEE Communications Magazine*, 54(12): 22–29, 2016.

[8] Ling Li, Shancang Li, and Shanshan Zhao. QoS-Aware Scheduling of Services-Oriented Internet of Things. *IEEE Transactions on Industrial Informatics*, 10(2):1497–1505, 2014.

[9] Stelios Sotiriadis, Nik Bessis, Cristiana Amza, and Rajkumar Buyya. Elastic Load Balancing for Dynamic Virtual Machine Reconfiguration Based on Vertical and Horizontal Scaling. *IEEE Transactions on Services Computing*, 12(2):319–334, 2019.

[10] Jesús Alejandro Cárdenes Cabré, Doina Precup, and Ricardo Sanz. Horizontal and Vertical Self-Adaptive Cloud Controller with Reward Optimization for Resource Allocation. In *International Conference on Cloud and Autonomic Computing (ICCAC)*, pages 184–185. IEEE, 2017.

[11] T. Ramalingeswara Rao, Pabitra Mitra, Ravindara Bhatt, and A. Goswami. The big data system, components, tools, and technologies: a survey. *Knowledge and Information Systems*, pages 1–81, 2018. Advance online publication.

[12] Chii Chang, Satish Narayana Srirama, and Rajkumar Buyya. Internet of Things (IoT) and New Computing Paradigms. In Rajkumar Buyya and Satish Narayana Srirama, editors, *Fog and Edge Computing*, pages 3–23. Wiley Publishing, Hoboken, NJ, USA, 2019.

[13] Jian Wu, Qianhui Liang, and Elisa Bertino. Improving Scalability of Software Cloud for Composite Web Services. In *International Conference on Cloud Computing (CLOUD)*, pages 143–146. IEEE, 2009.

[14] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011.

[15] Altti Ilari Maarala, Xiang Su, and Jukka Riekki. Semantic Reasoning for Context-Aware Internet of Things Applications. *IEEE Internet of Things Journal*, 4(2):461–473, 2017.

[16] Yi Xu and Abdelsalam Helal. Scalable Cloud–Sensor Architecture for the Internet of Things. *IEEE Internet of Things Journal*, 3(3): 285–298, 2016.

[17] Roberto Girau, Salvatore Martis, and Luigi Atzori. Lysis: A Platform for IoT Distributed Applications Over Socially Connected Objects. *IEEE Internet of Things Journal*, 4(1):40–51, 2017.

[18] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Journal*, 5(2):696–707, 2018.

[19] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. Elasticity in cloud computing: a survey. *Annals of Telecommunications*, 70(7):289–309, 2015.

[20] Rajkumar Buyya and Amir Dastjerdi. *Internet of Things: principles and paradigms*. Morgan Kaufmann, Cambridge, MA, USA, 2016.

[21] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 1st edition, 2003.

[22] Roy Want, Bill N. Schilit, and Scott Jenson. Enabling the Internet of Things. *Computer*, 48(1):28–35, 2015.

[23] Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger,

Tom Longstaff, Rick Kazman, Mark Klein, Linda Northrop, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. Ultra-Large-Scale Systems The Software Challenge of the Future. Technical report, Software Engineering Institute, Carnegie Mellon University, 2006.

[24] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[25] Flávia C. Delicato, Paulo F. Pires, and Thais Batista. The Resource Management Challenge in IoT. In Flávia C. Delicato, Paulo F. Pires, and Thais Batista, editors, *Resource Management for Internet of Things*, pages 7–18. Springer, Cham, Switzerland, 2017.

[26] Damian Roca, Daniel Nemirovsky, Mario Nemirovsky, Rodolfo Milito, and Mateo Valero. Emergent Behaviors in the Internet of Things: The Ultimate Ultra-Large-Scale System. *IEEE Micro*, 36(6):36–44, 2016.

[27] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, New York, NY, USA, 2nd edition, 2011.

[28] Martin Törngren and Paul T. Grogan. How to Deal with the Complexity of Future Cyber-Physical Systems? *Designs*, 2(4):1–16, 2018.

[29] Netflix. Conductor, 2019. URL https://netflix.github.io/conductor/.

[30] Martin Fowler. What do you mean by "Event-Driven"?, 2017. URL https://martinfowler.com/articles/201701-event-driven.html.

[31] Ian Sommerville, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Kwiatkowska, John Mcdermid, and Richard Paige. Large-scale Complex IT Systems. *Communications of the ACM*, 55(7):71–77, 2012.

[32] Reza Rezaei, Thiam Kian Chiew, and Sai Peck Lee. An interoperability model for ultra large scale systems. *Advances in Engineering Software*, 67:22–46, 2014.

[33] Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Panos Vassiliadist, Marco Autili, Marco Aurélio Gerosa, and Amira Ben Hamida. Service-oriented middleware for the Future Internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011.

[34] Adam Barker, Jon B. Weissman, and Jano I. Van Hemert. Reducing Data Transfer in Service-Oriented Architectures: The Circulate Approach. *IEEE Transactions on Services Computing*, 5(3):437–449, 2012.

[35] Michael Hahn, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Modeling and execution of data-aware choreographies: an overview. *Computer Science - Research and Development*, 33(3): 329–340, 2018.

[36] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Towards Megaprogramming: A Paradigm for Component-Based Programming. *Communications of the ACM*, 35(11):89–99, 1992.

[37] Nam Ky Giang, Rodger Lea, and Victor C. M. Leung. Exogenous Coordination for Building Fog-Based Cyber Physical Social Computing and Networking Systems. *IEEE Access*, 6:31740–31749, 2018.

[38] Nam Ky Giang, Rodger Lea, and Victor C. M. Leung. Developing applications in large scale, dynamic fog computing: A case study. *Software: Practice and Experience*, pages 1–14, 2019. Advance online publication.

[39] Matthias Galster, Uwe Zdun, Danny Weyns, Rick Rabiser, Bo Zhang, Michael Goedicke, and Gilles Perrouin. Variability and Complexity in Software Design: Towards a Research Agenda. *SIGSOFT Software Engineering Notes*, 41(6):27–30, 2017.

[40] Danny Weyns, Gowri Sankar Ramachandran, and Ritesh Kumar Singh. Self-managing Internet of Things. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*, volume 10706 of *Lecture Notes in Computer Science*, pages 67–84, Cham, Switzerland, 2018. Springer.

[41] Gerald Holl, Paul Grünbacher, and Rick Rabiser. A systematic review and an expert survey on capabilities supporting multi product lines. *Information and Software Technology*, 54(8):828–852, 2012.

[42] Kevin Ashton. That 'Internet of Things' Thing. *RFID Journal*, page 1, 2009.

[43] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–105, 1991.

[44] ITU-T. Overview of the Internet of things. Technical Report ITU-T Y.4000/Y.2060, International Telecommunication Union, 2012.

[45] Karl Popper. Three Worlds. The Tanner Lecture on Human Values, 1978.

[46] Ing-Ray Chen, Jia Guo, and Fenye Bao. Trust Management for SOA-Based IoT and Its Application to Service Composition. *IEEE Transactions on Services Computing*, 9(3):482–495, 2016.

[47] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.

[48] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, and Taieb Znati. Cloud of Things for Sensing-as-a-Service: Architecture, Algorithms, and Use Case. *IEEE Internet of Things Journal*, 3(6): 1099–1112, 2016.

[49] Robert Brzoza-Woch and Piotr Nawrocki. FPGA-Based Web Services – Infinite Potential or a Road to Nowhere? *IEEE Internet Computing*, 20(1):44–51, 2016.

[50] Michele Nitti, Virginia Pilloni, Giuseppe Colistra, and Luigi Atzori. The Virtual Object as a Major Element of the Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials*, 18(2):1228–1240, 2016.

[51] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, 2014.

[52] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services. *IEEE Transactions on Services Computing*, 3(3): 223–235, 2010.

[53] Karl D. Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[54] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[55] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, pages 253–266. ACM, 2008.

[56] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. In Dieter Uckelmann, Mark Harrison, and Florian Michahelles, editors, *Architecting the Internet of Things*, pages 97–129. Springer, Berlin, Germany, 2011.

[57] Sylvain Cherrier and Yacine M. Ghamri-Doudane. The "Object-as-a-Service" paradigm. In *Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–7. IEEE, 2014.

[58] Douglas K. Barry and David Dick. *Web Services, Service-Oriented Architectures, and Cloud Computing: The Savy Manager's Guide*. Morgan Kaufmann, Burlington, MA, USA, 2nd edition, 2013.

[59] Kung-Kiu Lau and Simone Di Cola. *An Introduction to Component-based Software Development*. World Scientific, Singapore, 1st edition, 2017.

[60] Michael Bell. *SOA Modeling Patterns for Service Oriented Discovery and Analysis*. Wiley Publishing, Hoboken, NJ, USA, 1st edition, 2010.

[61] Chandrashekar Jatoth, G. R. Gangadharan, and Rajkumar Buyya. Computational Intelligence Based QoS-Aware Web Service Composition: A Systematic Literature Review. *IEEE Transactions on Services Computing*, 10(3):475–492, 2017.

[62] Dominique Guinard. Towards the Web of Things: Web Mashups for Embedded Devices. In *International World Wide Web Conference (WWW)*, pages 1–8. ACM, 2009.

[63] Martin Bauer, Nicola Bui, Jourik De Loof, Carsten Magerkurth,

Andreas Nettsträter, Julinda Stefa, and Joachim W. Walewski. IoT Reference Model. In Alessandro Bassi, Martin Bauer, Martin Fiedler, Thorsten Kramp, Rob van Kranenburg, Sebastian Lange, and Stefan Meissner, editors, *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*, pages 113–162. Springer, Berlin, Germany, 2013.

[64] João Rufino, Muhammad Alam, Joaquim Ferreira, Abdur Rehman, and Kim Fung Tsang. Orchestration of containerized microservices for IIoT using Docker. In *International Conference on Industrial Technology (ICIT)*, pages 1532–1536. IEEE, 2017.

[65] Cesare Pautasso. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.

[66] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Thesis, University of California, Irvine, 2000.

[67] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision. In *International Conference on World Wide Web (WWW)*, pages 805–814. ACM, 2008.

[68] Dominique Guinard, Iulia Ion, and Simon Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective. In Alessandro Puiatti and Tao Gu, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, volume 104 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 326–337, Berlin, Germany, 2012. Springer.

[69] Choonhwa Lee, Chengyang Wang, Eunsam Kim, and Sumi Helal. Blueprint Flow: A Declarative Service Composition Framework for Cloud Applications. *IEEE Access*, 5:17634–17643, 2017.

[70] Rebeca P. Diaz Redondo, Ana Fernandez Vilas, Manuel Ramos Cabrer, Jose J. Pazos Arias, and Marta Rey Lopez. Enhancing Residential Gateways: OSGi Service Composition. *IEEE Transactions on Consumer Electronics*, 53(1):87–95, 2007.

[71] Hye-Young Paik, Angel Lagares Lemos, Moshe Chai Barukh, Boualem Benatallah, and Aarthi Natarajan. *Web Service Implementation and Composition Techniques*. Springer, Cham, Switzerland, 1st edition, 2017.

[72] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web Service Composition: A Survey of Techniques and Tools. *ACM Computing Surveys*, 48(3):1–41, 2015.

[73] Remco Dijkman and Marlon Dumas. Service-oriented design: a multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.

[74] Damian Arellanes and Kung-Kiu Lau. Analysis and Classification of Service Interactions for the Scalability of the Internet of Things. In *International Congress on Internet of Things (ICIOT)*, pages 80–87. IEEE, 2018.

[75] Matthew Shields. Control- Versus Data-Driven Workflows. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 167–173. Springer, London, UK, 2007.

[76] Amazon. Amazon Simple Workflow Service (SWF), 2019. URL https://aws.amazon.com/swf/.

[77] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[78] Carl Adam Petri. *Kommunikation mit Automaten*. PhD Thesis, Institut für Instrumentelle Mathematik, 1962.

[79] Yong Zhao, Michael Wilde, and Ian Foster. Virtual Data Language: A Typed Workflow Notation for Diversely Structured Scientific Data. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 258–275. Springer, London, UK, 2007.

[80] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. The Cactus Framework and Toolkit: Design and Applications. In José M. L. M. Palma, A. Augusto Sousa, Jack Dongarra, and Vicente Hernández, editors, *High Performance Computing for Computational Science — VECPAR 2002*, volume 2565 of *Lecture Notes in Computer Science*, pages 197–227, Berlin, Germany, 2003. Springer.

[81] OASIS. Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[82] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The Triana Workflow Environment: Architecture and Applications. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 320–339. Springer, London, UK, 2007.

[83] OpenJS Foundation. Node-RED: Flow-based programming for the Internet of Things, 2019. URL https://nodered.org/.

[84] Li Da Xu and Wattana Viriyasitavat. A Novel Architecture for Requirement-Oriented Participation Decision in Service Workflows. *IEEE Transactions on Industrial Informatics*, 10(2):1478–1485, 2014.

[85] Nathaniel Palmer, Swenson Keith, Reddy Surendra, Fingar Peter, Setrag Khoshafian, and Larry Hawes. *BPM Everywhere: Internet of Things, Process of Everything*. Future Strategies Inc., Lighthouse Point, FL, USA, 1st edition, 2015.

[86] Jakob Mass, Chii Chang, and Satish N. Srirama. Workflow Model Distribution or Code Distribution? Ideal Approach for Service Composition of the Internet of Things. In *International Conference on Services Computing (SCC)*, pages 649–656. IEEE, 2016.

[87] Ronny Seiger, Christine Keller, Florian Niebling, and Thomas Schlegel. Modelling complex and flexible processes for smart cyber-physical environments. *Journal of Computational Science*, 10:137–148, 2015.

[88] Ronny Seiger, Steffen Huber, Peter Heisig, and Uwe Aßmann. Toward a framework for self-adaptive workflows in cyber-physical systems. *Software & Systems Modeling*, 18(2):1117–1134, 2019.

[89] Shangguang Wang, Ao Zhou, Mingzhe Yang, Lei Sun, Ching-Hsien Hsu, and Fangchun Yang. Service Composition in Cyber-Physical-Social Systems. *IEEE Transactions on Emerging Topics in Computing*, pages 1–11, 2019. Advance online publication.

[90] Alexander Jungmann and Felix Mohr. An approach towards adaptive service composition in markets of composed services. *Journal of Internet Services and Applications*, 6(1):1–18, 2015.

[91] Mohamed Essaid Khanouche, Yacine Amirat, Abdelghani Chibani, Moussa Kerkar, and Ali Yachir. Energy-Centered and QoS-Aware Services Selection for Internet of Things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1256–1269, 2016.

[92] Shuiguang Deng, Zhengzhe Xiang, Jianwei Yin, Javid Taheri, and Albert Y. Zomaya. Composition-Driven IoT Service Provisioning in Distributed Edges. *IEEE Access*, 6:54258–54269, 2018.

[93] Rong Yang, Bing Li, and Can Cheng. A Petri Net-Based Approach to Service Composition and Monitoring in the IOT. In *Asia-Pacific Services Computing Conference (APSCC)*, pages 16–22. IEEE, 2014.

[94] Mengyu Sun, Zhensheng Shi, Shengjun Chen, Zhangbing Zhou, and Yucong Duan. Energy-Efficient Composition of Configurable Internet of Things Services. *IEEE Access*, 5:25609–25622, 2017.

[95] Osama Alsaryrah, Ibrahim Mashal, and Tein-Yaw Chung. Bi-Objective Optimization for Energy Aware Internet of Things Service Composition. *IEEE Access*, 6:26809–26819, 2018.

[96] Lei Huo and Zhiliang Wang. Service composition instantiation based on cross-modified artificial Bee Colony algorithm. *China Communications*, 13(10):233–244, 2016.

[97] Zhangbing Zhou, Deng Zhao, Lu Liu, and Patrick C. K. Hung. Energy-aware composition for wireless sensor networks as a service. *Future Generation Computer Systems*, 80:299–310, 2018.

[98] Qian Li, Runliang Dou, Fuzan Chen, and Guofang Nan. A QoS-oriented Web service composition approach based on multi-population genetic algorithm for Internet of things. *International Journal of Computational Intelligence Systems*, 7(sup2):26–34, 2014.

[99] Safina Showkat Ara, Zia Ush Shamszaman, and Ilyoung Chong. Web-of-Objects Based User-Centric Semantic Service Composition

Methodology in the Internet of Things. *International Journal of Distributed Sensor Networks*, 10(5):1–11, 2014.

[100] Mahmoud M. Badawy, Zainab H. Ali, and Hesham A. Ali. QoS provisioning framework for service-oriented internet of things (IoT). *Cluster Computing*, pages 1–17, 2019. Advance online publication.

[101] Samir Berrani, Ali Yachir, Badis Djemaa, and Mohamed Aissani. Extended multi-agent system based service composition in the Internet of things. In *International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–8. IEEE, 2018.

[102] A. Urbieta, A. González-Beltrán, S. Ben Mokhtar, M. Anwar Hossain, and L. Capra. Adaptive and context-aware service composition for IoT-based smart cities. *Future Generation Computer Systems*, 76: 262–274, 2017.

[103] Tomasz Szydlo, Robert Brzoza-Woch, Joanna Sendorek, Mateusz Windak, and Chris Gniady. Flow-Based Programming for IoT Leveraging Fog Computing. In *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 74–79. IEEE, 2017.

[104] Jonathan Lee, Shin-Jie Lee, and Ping-Feng Wang. A Framework for Composing SOAP, Non-SOAP and Non-Web Services. *IEEE Transactions on Services Computing*, 8(2):240–250, 2015.

[105] Farzad Khodadadi, Amir Vahid Dastjerdi, and Rajkumar Buyya. Simurgh: A framework for effective discovery, programming, and integration of services exposed in IoT. In *International Conference on Recent Advances in Internet of Things (RIoT)*, pages 1–6. IEEE, 2015.

[106] Panagiotis Vlacheas, Raffaele Giaffreda, Vera Stavroulaki, Dimitris Kelaidonis, Vassilis Foteinos, George Poulios, Panagiotis Demestichas, Andrey Somov, Abdur Rahim Biswas, and Klaus Moessner. Enabling smart cities through a cognitive management framework for the internet of things. *IEEE Communications Magazine*, 51(6): 102–111, 2013.

[107] Hongming Cai, Yizhi Gu, Athanasios V. Vasilakos, Boyi Xu, and Jianzhong Zhou. Model-Driven Development Patterns for Mobile Services in Cloud of Things. *IEEE Transactions on Cloud Computing*, 6(3):771–784, 2018.

[108] In-Young Ko, Han-Gyu Ko, Angel Jimenez Molina, and Jung-Hyun Kwon. SoIoT: Toward A User-Centric IoT-Based Service Framework. *ACM Transactions on Internet Technology*, 16(2):1–21, 2016.

[109] Alfred Åkesson, Görel Hedin, Mattias Nordahl, and Boris Magnusson. ComPOS: Composing Oblivious Services. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 132–138. IEEE, 2019.

[110] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, New York, NY, USA, 2014.

[111] Le Kim-Hung, Soumya Kanti Datta, Christian Bonnet, François Hamon, and Alexandre Boudonne. A scalable IoT framework to design logical data flow using virtual sensor. In *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–7. IEEE, 2017.

[112] Anne H. Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z. Sheng. IoT Middleware: A Survey on Issues and Enabling Technologies. *IEEE Internet of Things Journal*, 4(1):1–20, 2017.

[113] Federico Montori, Luca Bedogni, and Luciano Bononi. A Collaborative Internet of Things Architecture for Smart Cities and Environmental Monitoring. *IEEE Internet of Things Journal*, 5(2):592–605, 2018.

[114] Antonio Marcos Alberti, Gabriel Dias Scarpioni, Vaner Magalhães, Arismar Cerqueira Sodre, Joel Rodrigues, and Rodrigo da Rosa Righi. Advancing NovaGenesis Architecture Towards Future Internet of Things. *IEEE Internet of Things Journal*, 6(1):215–229, 2019.

[115] Sandra Rodríguez-Valenzuela, Juan A. Holgado-Terriza, José M. Gutiérrez-Guerrero, and Jesús L. Muros-Cobos. Distributed service-based approach for sensor data fusion in IoT environments. *Sensors*, 14(10):19200–19228, 2014.

[116] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. Glue.Things: A Mashup Platform for Wiring the Internet of Things with the Internet of Services. In *International Workshop on Web of Things (WoT)*, pages 16–21. ACM, 2014.

[117] Sylvain Cherrier, Ismail Salhi, Yacine M. Ghamri-Doudane, Stéphane Lohier, and Philippe Valembois. BeC 3: Behaviour Crowd Centric Composition for IoT applications. *Mobile Networks and Applications*, 19(1):18–32, 2014.

[118] Alexis Huf and Frank Siqueira. Composition of heterogeneous web services: A systematic review. *Journal of Network and Computer Applications*, 143:89–110, 2019.

[119] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Internet of Things applications: A systematic review. *Computer Networks*, 148:241–261, 2019.

[120] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.

[121] Cloves Carneiro and Tim Schmelmer. Microservices: The What and the Why. In Cloves Carneiro and Tim Schmelmer, editors, *Microservices From Day One*, pages 3–18. Apress, Berkeley, CA, USA, 2016.

[122] Sam Newman. *Building Microservices*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2015.

[123] Ion-Dorinel Filip, Florin Pop, Cristina Serbanescu, and Chang Choi. Microservices Scheduling Model Over Heterogeneous Cloud-Edge Environments As Support for IoT Applications. *IEEE Internet of Things Journal*, 5(4):2672–2681, 2018.

[124] Ahmed E. Khaled, Abdelsalam Helal, Wyatt Lindquist, and Choonhwa Lee. IoT-DDL–Device Description Language for the "T" in IoT. *IEEE Access*, 6:24048–24063, 2018.

[125] Lorenzo Carnevale, Antonio Celesti, Antonino Galletta, Schahram Dustdar, and Massimo Villari. From the Cloud to Edge and IoT: a Smart Orchestration Architecture for Enabling Osmotic Computing. In *International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 419–424. IEEE, 2018.

[126] Yuansong Qiao, Robert Nolani, Saul Gill, Guiming Fang, and Brian Lee. ThingNet: A micro-service based IoT macro-programming platform over edges and cloud. In *Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 1–4. IEEE, 2018.

[127] Kleanthis Thramboulidis, Danai C. Vachtsevanou, and Alexandros Solanos. Cyber-physical microservices: An IoT-based framework for manufacturing systems. In *Industrial Cyber-Physical Systems (ICPS)*, pages 232–239. IEEE, 2018.

[128] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, 2017.

[129] J. Paul Morrison. *Flow-Based Programming: A New Approach to Application Development*. CreateSpace, Paramount, CA, USA, 2nd edition, 2010.

[130] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[131] J. Paul Morrison. Data Stream Linkage Mechanism. *IBM Systems Journal*, 17(4):383–408, 1978.

[132] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web Services. In Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju, editors, *Web Services: Concepts, Architectures and Applications*, pages 123–149. Springer, Berlin, Germany, 2004.

[133] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. Developing IoT applications in the Fog: A Distributed Dataflow approach. In *International Conference on the Internet of Things (IOT)*, pages 155–162. IEEE, 2015.

[134] Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, and Mani Srivastava. DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks. In *International Conference on Internet of Things Design and Implementation (IoTDI)*, pages 172–177. ACM, 2019.

[135] National Instruments. What Is LabView?, 2019. URL `http://www.ni.com/en-gb/shop/labview.html`.

[136] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[137] Alistair Barros, Marlon Dumas, and Phillipa Oaks. Standards for Web Service Choreography and Orchestration: Status and Perspectives. In Christop J. Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 61–74, Berlin, Germany, 2005. Springer.

[138] Quan Sheng, Xiaoqiang Qiao, Athanasios Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014.

[139] Pieter Hens, Monique Snoeck, Geert Poels, and Manu De Backer. Process fragmentation, distribution and execution using an event-based interaction scheme. *Journal of Systems and Software*, 89: 170–192, 2014.

[140] Ward Jaradat, Alan Dearle, and Adam Barker. Towards an autonomous decentralized orchestration system. *Concurrency and Computation: Practice and Experience*, 28(11):3164–3179, 2016.

[141] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing Execution of Composite Web Services. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 170–187. ACM, 2004.

[142] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *International World Wide Web conference (WWW)*, pages 134–143. ACM, 2004.

[143] Michael Pantazoglou, Ioannis Pogkas, and Aphrodite Tsalgatidou. Decentralized Enactment of BPEL Processes. *IEEE Transactions on Services Computing*, 7(2):184–197, 2014.

[144] Daniel Wutke, Daniel Martin, and Frank Leymann. Model and infrastructure for decentralized workflow enactment. In *ACM Symposium on Applied Computing (SAC)*, pages 90–94. ACM, 2008.

[145] Daniel Martin, Daniel Wutke, and Frank Leymann. Tuplespace middleware for Petri net-based workflow execution. *International Journal of Web and Grid Services*, 6(1):35–57, 2010.

[146] Mirko Sonntag, Katharina Gorlach, Dimka Karastoyanova, Frank Leymann, and Michael Reiter. Process space-based scientific workflow enactment. *International Journal of Business Process Integration and Management*, 5(1):32–44, 2010.

[147] Xitong Kang, Xudong Liu, Hailong Sun, Yanjiu Huang, and Chao Zhou. Improving Performance for Decentralized Execution of Composite Web Services. In *World Congress on Services (SERVICES)*, pages 582–589. IEEE, 2010.

[148] Walid Fdhila, Marlon Dumas, Claude Godart, and Luciano García-Bañuelos. Heuristics for composite Web service decentralization. *Software & Systems Modeling*, 13(2):599–619, 2014.

[149] Ronny Seiger, Steffen Huber, and Thomas Schlegel. PROtEUS: An Integrated System for Process Execution in Cyber-Physical Systems. In Khaled Gaaloul, Rainer Schmidt, Selmin Nurcan, Sérgio Guerreiro, and Qin Ma, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 214 of *Lecture Notes in Business Information Processing*, pages 265–280, Cham, Switzerland, 2015. Springer.

[150] Yongyang Cheng, Shuai Zhao, Bo Cheng, and Junliang Chen. A Service-Based Fog Execution Environment for the IoT-Aware Business Process Applications. In *International Conference on Web Services (ICWS)*, pages 323–326. IEEE, 2018.

[151] Amazon. AWS Step Functions, 2019. URL http://aws.amazon.com/step-functions/.

[152] Adam Barker, Christopher D. Walton, and David Robertson. Choreographing Web Services. *IEEE Transactions on Services Computing*, 2(2):152–166, 2009.

[153] Gero Decker, Oliver Kopp, and Alistair Barros. An Introduction to Service Choreographies. *Information Technology*, 52(2):122–127, 2008.

[154] OW2 Consortium. CHOReVOLUTION. URL http://www.chorevolution.eu.

[155] S. Cherrier and R. Langar. Services organisation in IoT: mixing Orchestration and Choreography. In *Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–4. IEEE, 2018.

[156] ActnConnect. The choreographic platform Actorsphere, 2019. URL http://actnconnect.de/actorsphere_en.

[157] Damian Arellanes and Kung-Kiu Lau. Workflow Variability for Autonomic IoT Systems. In *International Conference on Autonomic Computing (ICAC)*. IEEE, 2019.

[158] Damian Arellanes and Kung-Kiu Lau. Algebraic Service Composition for User-Centric IoT Applications. In Dimitrios Georgakopoulos and Liang-Jie Zhang, editors, *Internet of Things – ICIOT 2018*, volume 10972 of *Lecture Notes in Computer Science*, pages 56–69, Cham, Switzerland, 2018. Springer.

[159] Damian Arellanes and Kung-Kiu Lau. Exogenous Connectors for Hierarchical Service Composition. In *International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 125–132. IEEE, 2017.

[160] Kung-Kiu Lau, Perla Velasco Elizondo, and Zheng Wang. Exogenous Connectors for Software Components. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106, Berlin, Germany, 2005. Springer.

[161] Perla Velasco Elizondo and Kung-Kiu Lau. A catalogue of component connectors to support development with reuse. *Journal of Systems and Software*, 83(7):1165–1178, 2010.

[162] Damian Arellanes and Kung-Kiu Lau. Decentralized Data Flows in Algebraic Service Compositions for the Scalability of IoT Systems. In *World Forum on Internet of Things (WF-IoT)*, pages 668–673. IEEE, 2019.

[163] Damian Arellanes and Kung-Kiu Lau. D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures. In *International Symposium on Cloud and Service Computing (SC2)*, pages 283–286. IEEE, 2017.

[164] T. Lin, H. Rivano, and F. Le Mouël. A Survey of Smart Parking Solutions. *IEEE Transactions on Intelligent Transportation Systems*, 18(12):3229–3253, 2017.

[165] Tullio Giuffrè, Sabato Marco Siniscalchi, and Giovanni Tesoriere. A Novel Architecture of Parking Management for Smart Cities. *Procedia - Social and Behavioral Sciences*, 53:16–28, 2012.

[166] Jan Höller, Vlasios Tsiatsis, Catherine Mulligan, Stamatis Karnouskos, Stefan Avesand, and David Boyle. *From Machine-To-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Academic Press, Oxford, UK, 1st edition, 2014.

[167] Tariq Samad and Anuradha Annaswamy. The Impact of Control Technology: Overview, Success Stories, and Research Challenges. Technical report, IEEE Control Systems Society, 2011.

[168] Jasmine Sekhon and Cody Fleming. Towards improved testing for deep learning. In *International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[169] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. dAmorim, and M. D. Ernst. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *International Conference on Automated Software Engineering (ASE)*, pages 669–679. IEEE, 2015.

[170] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B. Dasgupta, and Subhrajit Bhattacharya. Anomaly Detection Using Program Control Flow Graph Mining From Execution Logs. In *International ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 215–224. ACM, 2016.

[171] Fábio Bezerra, Jacques Wainer, and W. M. P. van der Aalst. Anomaly Detection Using Process Mining. In Terry Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer, and Roland Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *Lecture Notes in Business Information Processing*, pages 149–161, Berlin, Germany, 2009. Springer.

[172] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[173] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 184–196. ACM, 1998.

[174] Babak Yadegari, Jon Stephens, and Saumya Debray. Analysis of Exception-Based Control Transfers. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 205–216. ACM, 2017.

[175] Zipkin. OpenZipkin · A distributed tracing system, 2019. URL https://zipkin.io/.

[176] Alexandre Bergel, Felipe Bañados, Romain Robbes, and Walter Binder. Execution profiling blueprints. *Software: Practice and Experience*, 42(9):1165–1192, 2012.

[177] Joshua Garcia, Daniel Popescu, Gholamreza Safi, William G. J. Halfond, and Nenad Medvidovic. Identifying Message Flow in Distributed Event-based Systems. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 367–377. ACM, 2013.

[178] A. Burattin, A. Sperduti, and W. M. P. van der Aalst. Control-flow discovery from event streams. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2420–2427. IEEE, 2014.

[179] Wil van der Aalst. *Process Mining: Data Science in Action*. Springer, Berlin, Germany, second edition, 2016.

[180] Volodymyr Leno, Abel Armas-Cervantes, Marlon Dumas, Marcello La Rosa, and Fabrizio M. Maggi. Discovering Process Maps from Event Streams. In *International Conference on Software and System Process (ICSSP)*, pages 86–95. ACM, 2018.

[181] Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD Thesis, Carnegie Mellon University, 1991.

[182] Ryan Cunningham and Eddie Kohler. Making Events Less Slippery with Eel. In *Conference on Hot Topics in Operating Systems (HOTOS)*, pages 1–6. USENIX Association, 2005.

[183] Josh Evans. Mastering Chaos - A Netflix Guide to Microservices, 2016. URL https://www.infoq.com/presentations/netflix-chaos-microservices.

[184] Payam Barnaghi and Amit Sheth. On Searching the Internet of Things: Requirements and Challenges. *IEEE Intelligent Systems*, 31(6):71–75, 2016.

[185] Zhenyu Wen, Renyu Yang, Peter Garraghan, Tao Lin, Jie Xu, and Michael Rovatsos. Fog Orchestration for Internet of Things Services. *IEEE Internet Computing*, 21(2):16–24, 2017.

[186] Michele Zorzi, Alexander Gluhak, Sebastian Lange, and Alessandro Bassi. From today's INTRAnet of things to a future INTERnet of things: a wireless- and mobility-related view. *IEEE Wireless Communications*, 17(6):44–51, 2010.

[187] Jörg Heuer, Johannes Hund, and Oliver Pfaff. Toward the Web of Things: Applying Web Technologies to the Physical World. *Computer*, 48(5):34–42, 2015.

[188] Samir Tata, Rakesh Jain, Heiko Ludwig, and Sandeep Gopisetty. Living in the Cloud or on the Edge: Opportunities and Challenges of IOT Application Architecture. In *International Conference on Services Computing (SCC)*, pages 220–224. IEEE, 2017.

[189] Adam Barker and Rajkumar Buyya. Decentralised Orchestration of Service-oriented Scientific Workflows. In *International Conference on Cloud Computing and Services Science (CLOSER)*, pages 222–231. SciTePress, 2011.

[190] Tanveer Ahmed, Abhinav Tripathi, and Abhishek Srivastava. Rain4service: An Approach towards Decentralized Web Service Composition. In *International Conference on Services Computing (SCC)*, pages 267–274, Anchorage, AK, USA, 2014. IEEE.

[191] T. Ahmed, M. Mrissa, and A. Srivastava. MagEl: A Magneto-Electric Effect-Inspired Approach for Web Service Composition. In *International Conference on Web Services (ICWS)*, pages 455–462. IEEE, 2014.

[192] Walid Fdhila, Ustun Yildiz, and Claude Godart. A Flexible Approach for Automatic Process Decentralization Using Dependency Tables. In *International Conference on Web Services (ICWS)*, pages 847–855. IEEE, 2009.

[193] M. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The Enterprise Service Bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, 2005.

[194] Nicolai Josuttis. *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2007.

[195] Walter Binder, Ion Constantinescu, and Boi Faltings. Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration. *International Journal of High Performance Computing and Networking*, 6(1):81–90, 2009.

[196] Janggwan Im, Seonghoon Kim, and Daeyoung Kim. IoT Mashup as a Service: Cloud-Based Mashup Service for the Internet of Things. In *International Conference on Services Computing (SCC)*, pages 462–469. IEEE, 2013.

[197] Felipe Pontes Guimaraes, Eduardo Hideo Kuroda, and Daniel Macedo Batista. Performance Evaluation of Choreographies and Orchestrations with a New Simulator for Service Compositions. In *International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 140–144. IEEE, 2012.

[198] David Liu. Data-flow Distribution in FICAS Service Composition Infrastructure. In *International Conference on Parallel and Distributed Computing Systems (PDCAT)*, pages 1–6, 2002.

[199] Jan Seeger, Rohit A. Deshmukh, Vasil Sarafov, and Arne Bröring. Dynamic IoT Choreographies. *IEEE Pervasive Computing*, 18(1): 19–27, 2019.

[200] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, 24(3):796–809, 2019.

[201] Frank Leymann. Web Services: Distributed Applications Without Limits - An Outline. In *Database Systems for Business, Technology and Web (BTW)*, pages 1–22, 2003.

[202] Kung-Kiu Lau, Vladyslav Ukis, Perla Velasco, and Zheng Wang. A Component Model for Separation of Control Flow from Computation in Component-Based Systems. In *International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB)*, pages 57–69, 2006.

[203] Farhad Arbab. Composition of Interacting Computations. In Dina Goldin, Scott A. Smolka, and Peter Wegner, editors, *Interactive Computation*, pages 277–321. Springer, Berlin, Germany, 2006.

[204] Ahmed Safwat and M. B. Senousy. Addressing Challenges of Ultra Large Scale System on Requirements Engineering. In *International Conference on Communications, Management, and Information Technology (ICCMIT)*, International Conference on Communications, management, and Information technology (ICCMIT'2015), pages 442–449, 2015.

[205] Mehdi Mirakhorli, Amir Azim Sharifloo, and Fereidoon Shams. Architectural challenges of ultra large scale systems. In *International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS)*, pages 45–48. ACM, 2008.

[206] Andrea Ceccarelli, Andrea Bondavalli, Bernhard Froemel, Oliver Hoeftberger, and Hermann Kopetz. Basic Concepts on Systems of Systems. In Andrea Bondavalli, Sara Bouchenak, and Hermann Kopetz, editors, *Cyber-Physical Systems of Systems: Foundations – A Conceptual Model and Some Derivations: The AMADEOS Legacy*, pages 1–39. Springer, Cham, Switzerland, 2016.

[207] Amel Bennaceur, Ciaran McCormick, Jesús García-Galán, Charith Perera, Andrew Smith, Andrea Zisman, and Bashar Nuseibeh. Feed Me, Feed Me: An Exemplar for Engineering Adaptive Software. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 89–95. IEEE, 2016.

[208] Marcello La Rosa, Wil M. P. Van Der Aalst, Marlon Dumas, and Fredrik P. Milani. Business Process Variability Modeling: A Survey. *ACM Computing Surveys*, 50(1):1–45, 2017.

[209] Official government website for Northern Ireland citizens. Off-street and on-street parking, 2019. URL https://www.nidirect.gov.uk/articles/street-and-street-parking.

[210] Wei Fan, Zhengyong Chen, Zhang Xiong, and Hui Chen. The Internet of data: a new idea to extend the IOT in the digital world. *Frontiers of Computer Science*, 6(6):660–667, 2012.

[211] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Soft-*

ware Architecture: Foundations, Theory, and Practice. Wiley Publishing, Hoboken, NJ, USA, 1st edition, 2009.

[212] Paolo Nesi, Gianni Pantaleo, Michela Paolucci, and Imad Zaza. Auditing and Assessment of Data Traffic Flows in an IoT Architecture. In *International Conference on Collaboration and Internet Computing (CIC)*, pages 388–391. IEEE, 2018.

[213] Giacomo Ghidini, Sajal K. Das, and Vipul Gupta. Fuseviz: A framework for web-based data fusion and visualization in smart environments. In *International Conference on Mobile Ad-Hoc and Sensor Systems (MASS)*, pages 468–472. IEEE, 2012.

[214] Juan Luis Pérez, Álvaro Villalba, David Carrera, Iker Larizgoitia, and Vlad Trifa. The COMPOSE API for the Internet of Things. In *International Conference on World Wide Web (WWW)*, pages 971–976. ACM, 2014.

[215] Antonio Pintus, Davide Carboni, and Andrea Piras. Paraimpu: a platform for a social web of things. In *International Conference on World Wide Web (WWW)*, pages 401–404. ACM, 2012.

[216] Spacebrew. What is Spacebrew?, 2019. URL spacebrew.cc.

[217] Gennaro De Luca, Zhongtao Li, Sami Mian, and Yinong Chen. Visual programming language environment for different IoT and robotics platforms in computer science education. *CAAI Transactions on Intelligence Technology*, 3(2):119–130, 2018.

[218] Per Persson and Ola Angelsmark. Calvin – Merging Cloud and IoT. In *International Conference on Ambient Systems, Networks and Technologies (ANT)*, pages 210–217. Elsevier, 2015.

[219] Intel. IoT Services Orchestration Layer, 2019. URL https://github.com/intel/intel-iot-services-orchestration-layer.

[220] NoFlo. NoFlo: Flow-Based Programming for JavaScript, 2019. URL noflojs.org.

[221] Gábor Paller, Endri Bezati, Nebojša Taušan, Gábor Farkas, and Gábor Élő. Dataflow-based Heterogeneous Code Generator for IoT Applications. pages 428–434. IEEE, 2019.

[222] Yuuichi Teranishi, Takashi Kimata, Hiroaki Yamanaka, Eiji Kawai, and Hiroaki Harai. Dynamic Data Flow Processing in Edge Computing Environments. In *Annual Computer Software and Applications Conference (COMPSAC)*, pages 935–944. IEEE, 2017.

[223] Marco Mesiti, Stefano Valtolina, Luca Ferrari, Minh-Son Dao, and Koji Zettsu. An editable live ETL system for Ambient Intelligence environments. In *World Forum on Internet of Things (WF-IoT)*, pages 393–394. IEEE, 2015.

[224] Mahmoud Hussein, Shuai Li, and Ansgar Radermacher. Model-driven development of adaptive IoT systems. In *International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp)*, pages 20–27, 2017.

[225] Christian Prehofer and Ilias Gerostathopoulos. Modeling RESTful Web of Things Services: Concepts and Tools. In Quan Z. Sheng, Yongrui Qin, Lina Yao, and Boualem Benatallah, editors, *Managing the Web of Things*, pages 73–104. Morgan Kaufmann, Boston, MA, USA, 2017.

[226] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. Responsive Decentralized Composition of Service Mashups for the Internet of Things. In *International Conference on the Internet of Things (IoT)*, IoT'16, pages 53–61. ACM, 2016.

[227] Florian Daniel and Maristella Matera. *Mashups: Concepts, Models and Architectures*. Data-Centric Systems and Applications. Springer, Heidelberg, Germany, 1st edition, 2014.

[228] Michael Blackstock and Rodger Lea. IoT mashups with the WoTKit. In *International Conference on the Internet of Things (IoT)*, pages 1–8. IEEE, 2012.

[229] Florian Daniel and Barbara Pernici. Insights into Web Service Orchestration and Choreography:. *International Journal of E-Business Research*, 2(1):58–77, 2006.

[230] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá De Souza, and Vlad Trifa. SOA-based integration of the internet of things in enterprise services. In *International Conference on Web Services (ICWS)*, pages 968–975. IEEE, 2009.

[231] Antonio Pintus, Davide Carboni, Andrea Piras, and Alessandro Giordano. Connecting Smart Things through Web Services Orchestrations. In Florian Daniel and Federico Michele Facca, editors, *Current Trends in Web Engineering*, volume 6385 of *Lecture Notes in Computer Science*, pages 431–441, Berlin, Germany, 2010. Springer.

[232] Nils Glombitza, Sebastian Ebers, Dennis Pfisterer, and Stefan Fischer. Using BPEL to Realize Business Processes for an Internet of Things. In Hannes Frey, Xu Li, and Stefan Ruehrup, editors, *Ad-hoc, Mobile, and Wireless Networks*, volume 6811 of *Lecture Notes in Computer Science*, pages 294–307, Berlin, Germany, 2011. Springer.

[233] Hagen Overdick. Towards Resource-Oriented BPEL. In Thomas Gschwind and Cesare Pautasso, editors, *Emerging Web Services Technology*, pages 129–140. Birkhäuser, Basel, Switzerland, 2008.

[234] Feng Chen, Changrui Ren, Qinhua Wang, and Bing Shao. A process definition language for Internet of things. In *International Conference on Service Operations and Logistics, and Informatics (SOLI)*, pages 107–110. IEEE, 2012.

[235] Dulce Domingos, Francisco Martins, Ricardo Martinho, and Mário Silva. Ad-hoc changes in IoT-aware business processes. In *Internet of Things (IOT)*, pages 1–7. IEEE, 2010.

[236] OMG. Business Process Model And Notation (BPMN) Version 2.0, 2011. URL https://www.omg.org/spec/BPMN/2.0/.

[237] Camunda. BPMN Workflow Engine, 2019. URL https://camunda.com/products/bpmn-engine/.

[238] Tijs Rademakers. *Activiti in Action: Executable Business Processes in BPMN 2.0*. Manning Publications, Greenwich, CT, USA, 1st edition, 2012.

[239] Minjae Park, Hyunah Kim, Hyun Ahn, and Kwanghoon Pio Kim. A process-aware IoT application execution environment design. In *International Conference on Advanced Communication Technology (ICACT)*, pages 724–727. IEEE, 2018.

[240] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4chor: Extending BPEL for Modeling Choreographies. In *International Conference on Web Services (ICWS)*, pages 296–303. IEEE, 2007.

[241] Steve Ross-Talbot and Tony Fletcher. Web Services Choreography Description Language: Primer, 2006. URL https://www.w3.org/TR/ws-cdl-10-primer/.

[242] Johannes Maria Zaha, Alistair Barros, Marlon Dumas, and Arthur ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162, Berlin, Germany, 2006. Springer.

[243] Assaf Arkin, Sid Askary, and Scott Fordin. Web Service Choreography Interface (WSCI) 1.0, 2002. URL https://www.w3.org/TR/2002/NOTE-wsci-20020808/.

[244] Gero Decker and Alistair Barros. Interaction Modeling Using BPMN. In Arthur ter Hofstede, Boualem Benatallah, and Hye-Young Paik, editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 208–219, Berlin, Germany, 2007. Springer.

[245] Michael Zimmermann, Uwe Breitenbücher, and Frank Leymann. A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 121–131. SciTePress, 2017.

[246] Andreas Weiß, Vasilios Andrikopoulos, Santiago Gómez Sáez, Michael Hahn, and Dimka Karastoyanova. ChorSystem: A Message-Based System for the Life Cycle Management of Choreographies. In Christophe Debruyne, Hervé Panetto, Robert Meersman, Tharam Dillon, eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences*, volume 10033 of *Lecture Notes in Computer Science*, pages 503–521, Cham, Switzerland, 2016. Springer.

[247] F. L. Traversa and M. Di Ventra. Universal Memcomputing Machines. *IEEE Transactions on Neural Networks and Learning Systems*, 26 (11):2702–2715, 2015.

[248] A. Rajhans, A. Bhave, I. Ruchkin, B. H. Krogh, D. Garlan, A. Platzer, and B. Schmerl. Supporting Heterogeneity in Cyber-Physical Systems

Architectures. *IEEE Transactions on Automatic Control*, 59(12): 3178–3193, 2014.

[249] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: An Ensemble-based Component System. In *International ACM Sigsoft symposium on Component-based software engineering (CBSE)*, pages 81–90. ACM, 2013.

[250] Ikuo Nakagawa, Masahiro Hiji, and Hiroshi Esaki. Dripcast - Architecture and Implementation of Server-less Java Programming Framework for Billions of IoT Devices. *Journal of Information Processing*, 23(4):458–464, 2015.

[251] Y. Zhang, J. L. Chen, and B. Cheng. Integrating Events into SOA for IoT Services. *IEEE Communications Magazine*, 55(9):180–186, 2017.

[252] F. Halili, E. Rufati, and I. Ninka. Styles of Service Composition – Analysis and Comparison Methods. In *International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*, pages 302–307. IEEE, 2013.

Chapter 5

Concluding Remarks

This chapter provides the overall conclusion of the thesis, outlines the most notable limitations of the DX-MAN model and discusses future work.

5.1 Conclusions

Existing IoT service composition mechanisms (i.e., orchestration, choreography and dataflows) do not provide the requisite semantics for tackling the functional scale that future service-oriented IoT systems pose. This thesis fills such a gap by investigating the possibility of leveraging algebraic composition to support six functional scalability requirements, namely explicit control flow, distributed workflows, location transparency, decentralised data flows, separation of control, data and computation, and workflow variability. The result is an algebraic service composition model, DX-MAN, that uses a hierarchical bottom-up approach to incrementally compose services by exogenous composition operators.

Qualitative and empirical evaluations show that DX-MAN fully fulfils the requirements. In particular, composition operators define total compositionality for the explicit definition of variable control flows between families of workflow variants. In other words, a composition operator produces a composite service that is equivalent to a potentially infinite number of Turing machines. In this context, a Turing machine is expressed as an IoT workflow that is executed through the coordinated exchange of control among distributed composition operators (deployed on different *things* over the network). As coordination occurs from outside services, computation units do not interact with one another. This separation of control and computation results in

independent computations performed by location-agnostic services. When an invocation connector receives control, it reads data from a decentralised data space, invokes some computation and writes results in the space. Thus, exogenous connectors never exchange data during workflow execution. This (transparent) decentralised data exchange is mainly achieved by the separation of control and data. Remarkably, DX-MAN is the only service composition model that semantically separates control, data and computation.

This thesis does not claim that the functional scalability requirements identified are complete since other aspects need to be considered, e.g., *dynamic reconfiguration* and support for *implicit/explicit data flows*. But it takes into account the most critical desiderata which serve as an (initial) evaluation framework for future service composition mechanisms. Evidently, the framework can be refined to consider further requirements.

Although we were not able to validate the proposed model in a real scenario, this thesis showed that DX-MAN is a novel model that potentially alleviates the crisis that has shaken service composition over the last decade and it is, therefore, a remarkable contribution towards the construction of large-scale, service-oriented IoT systems that, clearly, are yet to come in the following years.

5.2 Limitations and Future Work

The current DX-MAN semantics covers six functional scalability desiderata, but it has some limitations as discussed below.

5.2.1 Data Flow Variability

One of the major barriers for the realisation of fully autonomous software systems is that the current DX-MAN semantics require the manual definition of data flows per workflow variant. Hence, we plan to investigate novel ways of defining *data flow variability* by leveraging the separation of control, data and computation. With this support, a feedback loop can select both the workflow control flow and data flow variants that best adapt to the current context. In particular, the DX-MAN semantics can be extended with the notion of *data flow spaces*.

5.2.2 Workflow Validation

The separation of control and data allows independent reasoning of such functional concerns, but it also brings out new challenges. For example, manually ensuring consistency of a workflow is not trivial and becomes infeasible as the number of composed services increases. To overcome this issue, we plan to develop an efficient approach that validates control flow consistency with respect to data flow for a given workflow variant. Consistency validation is crucial to ensure correct behaviour.

5.2.3 Evolution of Algebraic Compositions

A DX-MAN composition is static because it is defined at design-time only. Therefore, it is only suitable for closed environments where the system designer is fully aware of the system's operating environment. However, the dynamic, uncertain and complex nature of IoT makes it difficult to predict all possible scenarios that may potentially arise. In this regard, we are currently investigating an approach that dynamically evolves a DX-MAN composition to enable the run-time emergence of workflow spaces. Hence, composite services shall exhibit self-organising capabilities inspired in biological systems such as ant colonies, flocks of birds and mycelium. *Self-organisation* is indeed another crucial requirement to tackle the functional scale of future IoT systems.

5.2.4 Concurrency Support

DX-MAN currently defines passive services only and provides basic concurrency in parallel workflow variants. In the future, we plan to extend the DX-MAN semantics by considering an orthogonal dimension for concurrency management that will enable the modelling (per workflow variant) of *timing constraints* via topological ordering and semantic annotations. We will also investigate if *active services* can coexist with passive services in a particular DX-MAN composition. Also, we plan to provide asynchronous operators (e.g., an scheduler) for the aggregation of service behaviour.

Bibliography

[AAF02] Assaf Arkin, Sid Askary, and Scott Fordin. Web Service Choreography Interface (WSCI) 1.0, 2002.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web Services. In Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju, editors, *Web Services: Concepts, Architectures and Applications*, pages 123–149. Springer, Berlin, Germany, 2004.

[Act19] ActnConnect. The choreographic platform Actorsphere, 2019.

[AHGZ16] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, and Taieb Znati. Cloud of Things for Sensing-as-a-Service: Architecture, Algorithms, and Use Case. *IEEE Internet of Things Journal*, 3(6):1099–1112, 2016.

[ÅHNM19] Alfred Åkesson, Görel Hedin, Mattias Nordahl, and Boris Magnusson. ComPOS: Composing Oblivious Services. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 132–138. IEEE, 2019.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[AL17a] Damian Arellanes and Kung-Kiu Lau. D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures. In *International Symposium on Cloud and Service Computing (SC2)*, pages 283–286. IEEE, 2017.

[AL17b] Damian Arellanes and Kung-Kiu Lau. Exogenous Connectors for Hierarchical Service Composition. In *International Conference*

on Service-Oriented Computing and Applications (SOCA), pages 125–132. IEEE, 2017.

[AL18a] Damian Arellanes and Kung-Kiu Lau. Algebraic Service Composition for User-Centric IoT Applications. In Dimitrios Georgakopoulos and Liang-Jie Zhang, editors, *Internet of Things – ICIOT 2018*, volume 10972 of *Lecture Notes in Computer Science*, pages 56–69, Cham, Switzerland, 2018. Springer.

[AL18b] Damian Arellanes and Kung-Kiu Lau. Analysis and Classification of Service Interactions for the Scalability of the Internet of Things. In *International Congress on Internet of Things (ICIOT)*, pages 80–87. IEEE, 2018.

[AL19a] Damian Arellanes and Kung-Kiu Lau. Decentralized Data Flows in Algebraic Service Compositions for the Scalability of IoT Systems. In *World Forum on Internet of Things (WF-IoT)*, pages 668–673. IEEE, 2019.

[AL19b] Damian Arellanes and Kung-Kiu Lau. Evaluating IoT Service Composition Mechanisms for the Scalability of IoT Systems. *Manuscript submitted to the Future Generation Computer Systems Journal*, 2019.

[AL19c] Damian Arellanes and Kung-Kiu Lau. Workflow Variability for Autonomic IoT Systems. In *International Conference on Autonomic Computing (ICAC)*, pages 24–30. IEEE, 2019.

[Ama19a] Amazon. Amazon Simple Workflow Service (SWF), 2019.

[Ama19b] Amazon. AWS Step Functions, 2019.

[AMC18] Osama Alsaryrah, Ibrahim Mashal, and Tein-Yaw Chung. Bi-Objective Optimization for Energy Aware Internet of Things Service Composition. *IEEE Access*, 6:26809–26819, 2018.

[ARJ18] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Service composition approaches in IoT: A systematic review. *Journal of Network and Computer Applications*, 120:61–77, 2018.

[ARJ19] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Internet of Things applications: A systematic review. *Computer Networks*, 148:241–261, 2019.

[ASC14] Safina Showkat Ara, Zia Ush Shamszaman, and Ilyoung Chong. Web-of-Objects Based User-Centric Semantic Service Composition Methodology in the Internet of Things. *International Journal of Distributed Sensor Networks*, 10(5):1–11, 2014.

[Ash09] Kevin Ashton. That 'Internet of Things' Thing. *RFID Journal*, page 1, 2009.

[ASM+19] Antonio Marcos Alberti, Gabriel Dias Scarpioni, Vaner Magalhães, Arismar Cerqueira Sodre, Joel Rodrigues, and Rodrigo da Rosa Righi. Advancing NovaGenesis Architecture Towards Future Internet of Things. *IEEE Internet of Things Journal*, 6(1):215–229, 2019.

[ATS14] Tanveer Ahmed, Abhinav Tripathi, and Abhishek Srivastava. Rain4service: An Approach towards Decentralized Web Service Composition. In *International Conference on Services Computing (SCC)*, pages 267–274, Anchorage, AK, USA, 2014. IEEE.

[BAA19] Mahmoud M. Badawy, Zainab H. Ali, and Hesham A. Ali. QoS provisioning framework for service-oriented internet of things (IoT). *Cluster Computing*, pages 1–17, 2019. Advance online publication.

[BBDL+13] Martin Bauer, Nicola Bui, Jourik De Loof, Carsten Magerkurth, Andreas Nettsträter, Julinda Stefa, and Joachim W. Walewski. IoT Reference Model. In Alessandro Bassi, Martin Bauer, Martin Fiedler, Thorsten Kramp, Rob van Kranenburg, Sebastian Lange, and Stefan Meissner, editors, *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*, pages 113–162. Springer, Berlin, Germany, 2013.

[BCF09] Walter Binder, Ion Constantinescu, and Boi Faltings. Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration. *International Journal of High Performance Computing and Networking*, 6(1):81–90, 2009.

[BD13] Douglas K. Barry and David Dick. *Web Services, Service-Oriented Architectures, and Cloud Computing: The Savy Manager's Guide*. Morgan Kaufmann, Burlington, MA, USA, 2nd edition, 2013.

[BD16] Rajkumar Buyya and Amir Dastjerdi. *Internet of Things: principles and paradigms*. Morgan Kaufmann, Cambridge, MA, USA, 2016.

[BDO05] Alistair Barros, Marlon Dumas, and Phillipa Oaks. Standards for Web Service Choreography and Orchestration: Status and Perspectives. In Christop J. Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 61–74, Berlin, Germany, 2005. Springer.

[Bel10] Michael Bell. *SOA Modeling Patterns for Service Oriented Discovery and Analysis*. Wiley Publishing, Hoboken, NJ, USA, 1st edition, 2010.

[BL12] Michael Blackstock and Rodger Lea. IoT mashups with the WoTKit. In *International Conference on the Internet of Things (IoT)*, pages 1–8. IEEE, 2012.

[BMGG+16] Amel Bennaceur, Ciaran McCormick, Jesús García-Galán, Charith Perera, Andrew Smith, Andrea Zisman, and Bashar Nuseibeh. Feed Me, Feed Me: An Exemplar for Engineering Adaptive Software. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 89–95. IEEE, 2016.

[BS16] Payam Barnaghi and Amit Sheth. On Searching the Internet of Things: Requirements and Challenges. *IEEE Intelligent Systems*, 31(6):71–75, 2016.

[BWN16] Robert Brzoza-Woch and Piotr Nawrocki. FPGA-Based Web Services – Infinite Potential or a Road to Nowhere? *IEEE Internet Computing*, 20(1):44–51, 2016.

[BWR09] Adam Barker, Christopher D. Walton, and David Robertson. Choreographing Web Services. *IEEE Transactions on Services Computing*, 2(2):152–166, 2009.

[BWVH12] Adam Barker, Jon B. Weissman, and Jano I. Van Hemert. Reducing Data Transfer in Service-Oriented Architectures: The Circulate Approach. *IEEE Transactions on Services Computing*, 5(3):437–449, 2012.

[BYDA18] Samir Berrani, Ali Yachir, Badis Djemaa, and Mohamed Aissani. Extended multi-agent system based service composition in the Internet of things. In *International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–8. IEEE, 2018.

[Cam19] Camunda. BPMN Workflow Engine, 2019.

[CBF+16] Andrea Ceccarelli, Andrea Bondavalli, Bernhard Froemel, Oliver Hoeftberger, and Hermann Kopetz. Basic Concepts on Systems of Systems. In Andrea Bondavalli, Sara Bouchenak, and Hermann Kopetz, editors, *Cyber-Physical Systems of Systems: Foundations – A Conceptual Model and Some Derivations: The AMADEOS Legacy*, pages 1–39. Springer, Cham, Switzerland, 2016.

[CBZF16] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. Responsive Decentralized Composition of Service Mashups for the Internet of Things. In *International Conference on the Internet of Things (IoT)*, IoT'16, pages 53–61. ACM, 2016.

[CCG+18] Lorenzo Carnevale, Antonio Celesti, Antonino Galletta, Schahram Dustdar, and Massimo Villari. From the Cloud to Edge and IoT: a Smart Orchestration Architecture for Enabling Osmotic Computing. In *International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 419–424. IEEE, 2018.

[CCMN04] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *International World Wide Web conference (WWW)*, pages 134–143. ACM, 2004.

[CdCSR+15] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. Elasticity in cloud computing: a survey. *Annals of Telecommunications*, 70(7):289–309, 2015.

[CDK+02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[CGB16] Ing-Ray Chen, Jia Guo, and Fenye Bao. Trust Management for SOA-Based IoT and Its Application to Service Composition. *IEEE Transactions on Services Computing*, 9(3):482–495, 2016.

[CGD14] Sylvain Cherrier and Yacine M. Ghamri-Doudane. The "Object-as-a-Service" paradigm. In *Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–7. IEEE, 2014.

[CGK+11] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011.

[CGV+18] Hongming Cai, Yizhi Gu, Athanasios V. Vasilakos, Boyi Xu, and Jianzhong Zhou. Model-Driven Development Patterns for Mobile Services in Cloud of Things. *IEEE Transactions on Cloud Computing*, 6(3):771–784, 2018.

[Com18] Computing Research and Education Association of Australasia. Core: Computing Research & Education, 2018.

[Coo16] Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior (CAPES). Qualis, 2016.

[CPS17] Jesús Alejandro Cárdenes Cabré, Doina Precup, and Ricardo Sanz. Horizontal and Vertical Self-Adaptive Cloud Controller with Reward Optimization for Resource Allocation. In *International Conference on Cloud and Autonomic Computing (ICCAC)*, pages 184–185. IEEE, 2017.

[CRWS12] Feng Chen, Changrui Ren, Qinhua Wang, and Bing Shao. A process definition language for Internet of things. In *International Conference on Service Operations and Logistics, and Informatics (SOLI)*, pages 107–110. IEEE, 2012.

[CS16] Cloves Carneiro and Tim Schmelmer. Microservices: The What and the Why. In Cloves Carneiro and Tim Schmelmer, editors, *Microservices From Day One*, pages 3–18. Apress, Berkeley, CA, USA, 2016.

[CSB19] Chii Chang, Satish Narayana Srirama, and Rajkumar Buyya. Internet of Things (IoT) and New Computing Paradigms. In Rajkumar Buyya and Satish Narayana Srirama, editors, *Fog and Edge Computing*, pages 3–23. Wiley Publishing, Hoboken, NJ, USA, 2019.

[CSC+18] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Journal*, 5(2):696–707, 2018.

[CSGD+14] Sylvain Cherrier, Ismail Salhi, Yacine M. Ghamri-Doudane, Stéphane Lohier, and Philippe Valembois. BeC 3: Behaviour Crowd Centric Composition for IoT applications. *Mobile Networks and Applications*, 19(1):18–32, 2014.

[CZCC18] Yongyang Cheng, Shuai Zhao, Bo Cheng, and Junliang Chen. A Service-Based Fog Execution Environment for the IoT-Aware Business Process Applications. In *International Conference on Web Services (ICWS)*, pages 323–326. IEEE, 2018.

[DB07] Gero Decker and Alistair Barros. Interaction Modeling Using BPMN. In Arthur ter Hofstede, Boualem Benatallah, and Hye-Young Paik, editors, *Business Process Management Workshops*,

volume 4928 of *Lecture Notes in Computer Science*, pages 208–219, Berlin, Germany, 2007. Springer.

[DD04] Remco Dijkman and Marlon Dumas. Service-oriented design: a multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.

[Dev04] Keith Devlin. *Sets, Functions, and Logic: An Introduction to Abstract Mathematics*. CRC Press, Florida, USA, 3rd edition, 2004.

[DKB08] Gero Decker, Oliver Kopp, and Alistair Barros. An Introduction to Service Choreographies. *Information Technology*, 52(2):122–127, 2008.

[DKLW07] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4chor: Extending BPEL for Modeling Choreographies. In *International Conference on Web Services (ICWS)*, pages 296–303. IEEE, 2007.

[DLLMC18] Gennaro De Luca, Zhongtao Li, Sami Mian, and Yinong Chen. Visual programming language environment for different IoT and robotics platforms in computer science education. *CAAI Transactions on Intelligence Technology*, 3(2):119–130, 2018.

[DM14] Florian Daniel and Maristella Matera. *Mashups: Concepts, Models and Architectures*. Data-Centric Systems and Applications. Springer, Heidelberg, Germany, 1st edition, 2014.

[DMMS10] Dulce Domingos, Francisco Martins, Ricardo Martinho, and Mário Silva. Ad-hoc changes in IoT-aware business processes. In *Internet of Things (IOT)*, pages 1–7. IEEE, 2010.

[DP06] Florian Daniel and Barbara Pernici. Insights into Web Service Orchestration and Choreography:. *International Journal of E-Business Research*, 2(1):58–77, 2006.

[DPB17] Flávia C. Delicato, Paulo F. Pires, and Thais Batista. The Resource Management Challenge in IoT. In Flávia C. Delicato, Paulo F. Pires, and Thais Batista, editors, *Resource Management for Internet of Things*, pages 7–18. Springer, Cham, Switzerland, 2017.

[DXY+18] Shuiguang Deng, Zhengzhe Xiang, Jianwei Yin, Javid Taheri, and Albert Y. Zomaya. Composition-Driven IoT Service Provisioning in Distributed Edges. *IEEE Access*, 6:54258–54269, 2018.

[Els19] Elsevier. Future Generation Computer Systems, 2019.

[FCXC12] Wei Fan, Zhengyong Chen, Zhang Xiong, and Hui Chen. The Internet of data: a new idea to extend the IOT in the digital world. *Frontiers of Computer Science*, 6(6):660–667, 2012.

[FDGGB14] Walid Fdhila, Marlon Dumas, Claude Godart, and Luciano García-Bañuelos. Heuristics for composite Web service decentralization. *Software & Systems Modeling*, 13(2):599–619, 2014.

[FGG+06] Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Linda Northrop, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. Ultra-Large-Scale Systems The Software Challenge of the Future. Technical report, Software Engineering Institute, Carnegie Mellon University, 2006.

[Fie00] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Thesis, University of California, Irvine, 2000.

[FPSC18] Ion-Dorinel Filip, Florin Pop, Cristina Serbanescu, and Chang Choi. Microservices Scheduling Model Over Heterogeneous Cloud-Edge Environments As Support for IoT Applications. *IEEE Internet of Things Journal*, 5(4):2672–2681, 2018.

[FYG09] Walid Fdhila, Ustun Yildiz, and Claude Godart. A Flexible Approach for Automatic Process Decentralization Using Dependency Tables. In *International Conference on Web Services (ICWS)*, pages 847–855. IEEE, 2009.

[GAL+03] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. The Cactus Framework and Toolkit: Design and Applications. In José M. L. M. Palma, A. Augusto Sousa, Jack Dongarra, and Vicente Hernández, editors, *High Performance Computing for Computational Science*

— *VECPAR 2002*, volume 2565 of *Lecture Notes in Computer Science*, pages 197–227, Berlin, Germany, 2003. Springer.

[Gal11] Jean Gallier. *Discrete Mathematics*. Universitext. Springer, New York, NY, USA, 1st edition, 2011.

[GBLL15] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. Developing IoT applications in the Fog: A Distributed Dataflow approach. In *International Conference on the Internet of Things (IOT)*, pages 155–162. IEEE, 2015.

[GDG12] Giacomo Ghidini, Sajal K. Das, and Vipul Gupta. Fuseviz: A framework for web-based data fusion and visualization in smart environments. In *International Conference on Mobile Ad-Hoc and Sensor Systems (MASS)*, pages 468–472. IEEE, 2012.

[GEPF11] Nils Glombitza, Sebastian Ebers, Dennis Pfisterer, and Stefan Fischer. Using BPEL to Realize Business Processes for an Internet of Things. In Hannes Frey, Xu Li, and Stefan Ruehrup, editors, *Ad-hoc, Mobile, and Wireless Networks*, volume 6811 of *Lecture Notes in Computer Science*, pages 294–307, Berlin, Germany, 2011. Springer.

[GGKS02] Karl D. Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[GIM12] Dominique Guinard, Iulia Ion, and Simon Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective. In Alessandro Puiatti and Tao Gu, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, volume 104 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 326–337, Berlin, Germany, 2012. Springer.

[GKB12] Felipe Pontes Guimaraes, Eduardo Hideo Kuroda, and Daniel Macedo Batista. Performance Evaluation of Choreographies and Orchestrations with a New Simulator for Service Compositions. In *International Workshop on Computer Aided*

Modeling and Design of Communication Links and Networks (CAMAD), pages 140–144. IEEE, 2012.

[GLL18] Nam Ky Giang, Rodger Lea, and Victor C. M. Leung. Exogenous Coordination for Building Fog-Based Cyber Physical Social Computing and Networking Systems. *IEEE Access*, 6:31740–31749, 2018.

[GLL19] Nam Ky Giang, Rodger Lea, and Victor C. M. Leung. Developing applications in large scale, dynamic fog computing: A case study. *Software: Practice and Experience*, pages 1–14, 2019. Advance online publication.

[GMA17] Roberto Girau, Salvatore Martis, and Luigi Atzori. Lysis: A Platform for IoT Distributed Applications Over Socially Connected Objects. *IEEE Internet of Things Journal*, 4(1):40–51, 2017.

[Goo19] Google. Google Scholar, 2019.

[GTK+10] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services. *IEEE Transactions on Services Computing*, 3(3):223–235, 2010.

[GTMW11] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. In Dieter Uckelmann, Mark Harrison, and Florian Michahelles, editors, *Architecting the Internet of Things*, pages 97–129. Springer, Berlin, Germany, 2011.

[Gui09] Dominique Guinard. Towards the Web of Things: Web Mashups for Embedded Devices. In *International World Wide Web Conference (WWW)*, pages 1–8. ACM, 2009.

[GZW+17] Matthias Galster, Uwe Zdun, Danny Weyns, Rick Rabiser, Bo Zhang, Michael Goedicke, and Gilles Perrouin. Variability and Complexity in Software Design: Towards a Research Agenda. *SIGSOFT Software Engineering Notes*, 41(6):27–30, 2017.

[HBKL18] Michael Hahn, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Modeling and execution of data-aware choreographies: an overview. *Computer Science - Research and Development*, 33(3):329–340, 2018.

[HGR12] Gerald Holl, Paul Grünbacher, and Rick Rabiser. A systematic review and an expert survey on capabilities supporting multi product lines. *Information and Software Technology*, 54(8):828–852, 2012.

[HHP15] Jörg Heuer, Johannes Hund, and Oliver Pfaff. Toward the Web of Things: Applying Web Technologies to the Physical World. *Computer*, 48(5):34–42, 2015.

[HLR17] Mahmoud Hussein, Shuai Li, and Ansgar Radermacher. Model-driven development of adaptive IoT systems. In *International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp)*, pages 20–27, 2017.

[HN18] Marzieh Hamzei and Nima Jafari Navimipour. Toward Efficient Service Composition Techniques in the Internet of Things. *IEEE Internet of Things Journal*, 5(5):3774–3787, 2018.

[HS19] Alexis Huf and Frank Siqueira. Composition of heterogeneous web services: A systematic review. *Journal of Network and Computer Applications*, 143:89–110, 2019.

[HSPDB14] Pieter Hens, Monique Snoeck, Geert Poels, and Manu De Backer. Process fragmentation, distribution and execution using an event-based interaction scheme. *Journal of Systems and Software*, 89:170–192, 2014.

[HTM+14] Jan Höller, Vlasios Tsiatsis, Catherine Mulligan, Stamatis Karnouskos, Stefan Avesand, and David Boyle. *From Machine-To-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Academic Press, Oxford, UK, 1st edition, 2014.

[HW16] Lei Huo and Zhiliang Wang. Service composition instantiation based on cross-modified artificial Bee Colony algorithm. *China Communications*, 13(10):233–244, 2016.

[IGH+11] Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Panos Vassiliadist, Marco Autili, Marco Aurélio Gerosa, and Amira Ben Hamida. Service-oriented middleware for the Future Internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011.

[IKK13] Janggwan Im, Seonghoon Kim, and Daeyoung Kim. IoT Mashup as a Service: Cloud-Based Mashup Service for the Internet of Things. In *International Conference on Services Computing (SCC)*, pages 462–469. IEEE, 2013.

[Int19] Intel. IoT Services Orchestration Layer, 2019.

[IoT18] IoT Analytics. State of the IoT 2018: Number of IoT devices now at 7b – Market accelerating, 2018.

[ITU12] ITU-T. Overview of the Internet of things. Technical Report ITU-T Y.4000/Y.2060, International Telecommunication Union, 2012.

[JDB16] Ward Jaradat, Alan Dearle, and Adam Barker. Towards an autonomous decentralized orchestration system. *Concurrency and Computation: Practice and Experience*, 28(11):3164–3179, 2016.

[JGB17] Chandrashekar Jatoth, G. R. Gangadharan, and Rajkumar Buyya. Computational Intelligence Based QoS-Aware Web Service Composition: A Systematic Literature Review. *IEEE Transactions on Services Computing*, 10(3):475–492, 2017.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[JM15] Alexander Jungmann and Felix Mohr. An approach towards adaptive service composition in markets of composed services. *Journal of Internet Services and Applications*, 6(1):1–18, 2015.

[Jos07] Nicolai Josuttis. *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2007.

[KAC+16] Mohamed Essaid Khanouche, Yacine Amirat, Abdelghani Chibani, Moussa Kerkar, and Ali Yachir. Energy-Centered and QoS-Aware Services Selection for Internet of Things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1256–1269, 2016.

[KDB15] Farzad Khodadadi, Amir Vahid Dastjerdi, and Rajkumar Buyya. Simurgh: A framework for effective discovery, programming, and integration of services exposed in IoT. In *International Conference on Recent Advances in Internet of Things (RIoT)*, pages 1–6. IEEE, 2015.

[KHDB+17] Le Kim-Hung, Soumya Kanti Datta, Christian Bonnet, François Hamon, and Alexandre Boudonne. A scalable IoT framework to design logical data flow using virtual sensor. In *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–7. IEEE, 2017.

[KHLL18] Ahmed E. Khaled, Abdelsalam Helal, Wyatt Lindquist, and Choonhwa Lee. IoT-DDL–Device Description Language for the "T" in IoT. *IEEE Access*, 6:24048–24063, 2018.

[KKMK16] In-Young Ko, Han-Gyu Ko, Angel Jimenez Molina, and Jung-Hyun Kwon. SoIoT: Toward A User-Centric IoT-Based Service Framework. *ACM Transactions on Internet Technology*, 16(2):1–21, 2016.

[KLS+10] Xitong Kang, Xudong Liu, Hailong Sun, Yanjiu Huang, and Chao Zhou. Improving Performance for Decentralized Execution of Composite Web Services. In *World Congress on Services (SERVICES)*, pages 582–589. IEEE, 2010.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, New York, NY, USA, 2nd edition, 2011.

[KSRD14] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. Glue.Things: A Mashup Platform for

Wiring the Internet of Things with the Internet of Services. In *International Workshop on Web of Things (WoT)*, pages 16–21. ACM, 2014.

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[LDB15] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web Service Composition: A Survey of Techniques and Tools. *ACM Computing Surveys*, 48(3):1–41, 2015.

[LDC17] Kung-Kiu Lau and Simone Di Cola. *An Introduction to Component-based Software Development*. World Scientific, Singapore, 1st edition, 2017.

[LDCN14] Qian Li, Runliang Dou, Fuzan Chen, and Guofang Nan. A QoS-oriented Web service composition approach based on multi-population genetic algorithm for Internet of things. *International Journal of Computational Intelligence Systems*, 7(sup2):26–34, 2014.

[Liu02] David Liu. Data-flow Distribution in FICAS Service Composition Infrastructure. In *International Conference on Parallel and Distributed Computing Systems (PDCAT)*, pages 1–6, 2002.

[LLW15] Jonathan Lee, Shin-Jie Lee, and Ping-Feng Wang. A Framework for Composing SOAP, Non-SOAP and Non-Web Services. *IEEE Transactions on Services Computing*, 8(2):240–250, 2015.

[LLZ14] Ling Li, Shancang Li, and Shanshan Zhao. QoS-Aware Scheduling of Services-Oriented Internet of Things. *IEEE Transactions on Industrial Informatics*, 10(2):1497–1505, 2014.

[LR10] Kung-Kiu Lau and Tauseef Rana. A Taxonomy of Software Composition Mechanisms. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 102–110. IEEE, 2010.

[LSCPE18] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.

[LT12] Kung-Kiu Lau and Cuong M. Tran. X-MAN: An MDE Tool for Component-Based System Development. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 158–165. IEEE, 2012.

[LVEW05] Kung-Kiu Lau, Perla Velasco Elizondo, and Zheng Wang. Exogenous Connectors for Software Components. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106, Berlin, Heidelberg, 2005. Springer.

[LWKH17] Choonhwa Lee, Chengyang Wang, Eunsam Kim, and Sumi Helal. Blueprint Flow: A Declarative Service Composition Framework for Cloud Applications. *IEEE Access*, 5:17634–17643, 2017.

[MASS08] Mehdi Mirakhorli, Amir Azim Sharifloo, and Fereidoon Shams. Architectural challenges of ultra large scale systems. In *International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS)*, pages 45–48. ACM, 2008.

[MBB18] Federico Montori, Luca Bedogni, and Luciano Bononi. A Collaborative Internet of Things Architecture for Smart Cities and Environmental Monitoring. *IEEE Internet of Things Journal*, 5(2):592–605, 2018.

[MCS16] Jakob Mass, Chii Chang, and Satish N. Srirama. Workflow Model Distribution or Code Distribution? Ideal Approach for Service Composition of the Internet of Things. In *International Conference on Services Computing (SCC)*, pages 649–656. IEEE, 2016.

[MGZ14] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, New York, NY, USA, 2014.

[Mor78] J. Paul Morrison. Data Stream Linkage Mechanism. *IBM Systems Journal*, 17(4):383–408, 1978.

[Mor10] J. Paul Morrison. *Flow-Based Programming: A New Approach to Application Development*. CreateSpace, Paramount, CA, USA, 2nd edition, 2010.

[MS03] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 1st edition, 2003.

[MSR17] Altti Ilari Maarala, Xiang Su, and Jukka Riekki. Semantic Reasoning for Context-Aware Internet of Things Applications. *IEEE Internet of Things Journal*, 4(2):461–473, 2017.

[MVF+15] Marco Mesiti, Stefano Valtolina, Luca Ferrari, Minh-Son Dao, and Koji Zettsu. An editable live ETL system for Ambient Intelligence environments. In *World Forum on Internet of Things (WF-IoT)*, pages 393–394. IEEE, 2015.

[MVT17] Steen Maarten Van and Andrew S. Tanenbaum. *Distributed Systems*. CreateSpace, London, UK, 3rd edition, 2017.

[MWL10] Daniel Martin, Daniel Wutke, and Frank Leymann. Tuplespace middleware for Petri net-based workflow execution. *International Journal of Web and Grid Services*, 6(1):35–57, 2010.

[NAG19] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, 24(3):796–809, 2019.

[Nat19] National Instruments. What Is LabView?, 2019.

[NCS04] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing Execution of Composite Web Services. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 170–187. ACM, 2004.

[New15] Sam Newman. *Building Microservices*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2015.

[NGM+17] Anne H. Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z. Sheng. IoT Middleware: A Survey on Issues and Enabling Technologies. *IEEE Internet of Things Journal*, 4(1):1–20, 2017.

[NoF19] NoFlo. NoFlo: Flow-Based Programming for JavaScript, 2019.

[NPCA16] Michele Nitti, Virginia Pilloni, Giuseppe Colistra, and Luigi Atzori. The Virtual Object as a Major Element of the Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials*, 18(2):1228–1240, 2016.

[NPPZ18] Paolo Nesi, Gianni Pantaleo, Michela Paolucci, and Imad Zaza. Auditing and Assessment of Data Traffic Flows in an IoT Architecture. In *International Conference on Collaboration and Internet Computing (CIC)*, pages 388–391. IEEE, 2018.

[NTGS19] Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, and Mani Srivastava. DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks. In *International Conference on Internet of Things Design and Implementation (IoTDI)*, pages 172–177. ACM, 2019.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0, 2007.

[OMG11] OMG. Business Process Model And Notation (BPMN) Version 2.0, 2011.

[Ope19] OpenJS Foundation. Node-RED: Flow-based programming for the Internet of Things, 2019.

[Ove08] Hagen Overdick. Towards Resource-Oriented BPEL. In Thomas Gschwind and Cesare Pautasso, editors, *Emerging Web Services Technology*, pages 129–140. Birkhäuser, Basel, Switzerland, 2008.

[OW2] OW2 Consortium. CHOReVOLUTION.

[PA15] Per Persson and Ola Angelsmark. Calvin – Merging Cloud and IoT. In *International Conference on Ambient Systems, Networks and Technologies (ANT)*, pages 210–217. Elsevier, 2015.

[Pau09] Cesare Pautasso. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.

[PBT+19] Gábor Paller, Endri Bezati, Nebojša Taušan, Gábor Farkas, and Gábor Élő. Dataflow-based Heterogeneous Code Generator for IoT Applications. pages 428–434. IEEE, 2019.

[PCP12] Antonio Pintus, Davide Carboni, and Andrea Piras. Paraimpu: a platform for a social web of things. In *International Conference on World Wide Web (WWW)*, pages 401–404. ACM, 2012.

[PCPG10] Antonio Pintus, Davide Carboni, Andrea Piras, and Alessandro Giordano. Connecting Smart Things through Web Services Orchestrations. In Florian Daniel and Federico Michele Facca, editors, *Current Trends in Web Engineering*, volume 6385 of *Lecture Notes in Computer Science*, pages 431–441, Berlin, Germany, 2010. Springer.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD Thesis, Institut für Instrumentelle Mathematik, 1962.

[PG17] Christian Prehofer and Ilias Gerostathopoulos. Modeling RESTful Web of Things Services: Concepts and Tools. In Quan Z. Sheng, Yongrui Qin, Lina Yao, and Boualem Benatallah, editors, *Managing the Web of Things*, pages 73–104. Morgan Kaufmann, Boston, MA, USA, 2017.

[PKAK18] Minjae Park, Hyunah Kim, Hyun Ahn, and Kwanghoon Pio Kim. A process-aware IoT application execution environment design. In *International Conference on Advanced Communication Technology (ICACT)*, pages 724–727. IEEE, 2018.

[PKGZ08] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, pages 253–266. ACM, 2008.

[PKS+15] Nathaniel Palmer, Swenson Keith, Reddy Surendra, Fingar Peter, Setrag Khoshafian, and Larry Hawes. *BPM Everywhere: Internet of Things, Process of Everything*. Future Strategies Inc., Lighthouse Point, FL, USA, 1st edition, 2015.

[PLB+17] Hye-Young Paik, Angel Lagares Lemos, Moshe Chai Barukh, Boualem Benatallah, and Aarthi Natarajan. *Web Service Implementation and Composition Techniques*. Springer, Cham, Switzerland, 1st edition, 2017.

[Pop78] Karl Popper. Three Worlds. The Tanner Lecture on Human Values, 1978.

[PPT14] Michael Pantazoglou, Ioannis Pogkas, and Aphrodite Tsalgatidou. Decentralized Enactment of BPEL Processes. *IEEE Transactions on Services Computing*, 7(2):184–197, 2014.

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.

[PVC+14] Juan Luis Pérez, Álvaro Villalba, David Carrera, Iker Larizgoitia, and Vlad Trifa. The COMPOSE API for the Internet of Things. In *International Conference on World Wide Web (WWW)*, pages 971–976. ACM, 2014.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision. In *International Conference on World Wide Web (WWW)*, pages 805–814. ACM, 2008.

[QNG+18] Yuansong Qiao, Robert Nolani, Saul Gill, Guiming Fang, and

Brian Lee. ThingNet: A micro-service based IoT macro-programming platform over edges and cloud. In *Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 1–4. IEEE, 2018.

[Rad12] Tijs Rademakers. *Activiti in Action: Executable Business Processes in BPMN 2.0*. Manning Publications, Greenwich, CT, USA, 1st edition, 2012.

[RAF+17] João Rufino, Muhammad Alam, Joaquim Ferreira, Abdur Rehman, and Kim Fung Tsang. Orchestration of containerized microservices for IIoT using Docker. In *International Conference on Industrial Technology (ICIT)*, pages 1532–1536. IEEE, 2017.

[RCL14] Reza Rezaei, Thiam Kian Chiew, and Sai Peck Lee. An interoperability model for ultra large scale systems. *Advances in Engineering Software*, 67:22–46, 2014.

[RMBG18] T. Ramalingeswara Rao, Pabitra Mitra, Ravindara Bhatt, and A. Goswami. The big data system, components, tools, and technologies: a survey. *Knowledge and Information Systems*, pages 1–81, 2018. Advance online publication.

[RNN+16] Damian Roca, Daniel Nemirovsky, Mario Nemirovsky, Rodolfo Milito, and Mateo Valero. Emergent Behaviors in the Internet of Things: The Ultimate Ultra-Large-Scale System. *IEEE Micro*, 36(6):36–44, 2016.

[RTF06] Steve Ross-Talbot and Tony Fletcher. Web Services Choreography Description Language: Primer, 2006.

[RVC+07] Rebeca P. Diaz Redondo, Ana Fernandez Vilas, Manuel Ramos Cabrer, Jose J. Pazos Arias, and Marta Rey Lopez. Enhancing Residential Gateways: OSGi Service Composition. *IEEE Transactions on Consumer Electronics*, 53(1):87–95, 2007.

[RVHTGGMC14] Sandra Rodríguez-Valenzuela, Juan A. Holgado-Terriza, José M. Gutiérrez-Guerrero, and Jesús L. Muros-Cobos. Distributed service-based approach for sensor data fusion in IoT environments. *Sensors*, 14(10):19200–19228, 2014.

[SA11] Tariq Samad and Anuradha Annaswamy. The Impact of Control Technology: Overview, Success Stories, and Research Challenges. Technical report, IEEE Control Systems Society, 2011.

[SA16] Xiang Sun and Nirwan Ansari. EdgeIoT: Mobile Edge Computing for the Internet of Things. *IEEE Communications Magazine*, 54(12):22–29, 2016.

[SBAB19] Stelios Sotiriadis, Nik Bessis, Cristiana Amza, and Rajkumar Buyya. Elastic Load Balancing for Dynamic Virtual Machine Reconfiguration Based on Vertical and Horizontal Scaling. *IEEE Transactions on Services Computing*, 12(2):319–334, 2019.

[SBWS+17] Tomasz Szydlo, Robert Brzoza-Woch, Joanna Sendorek, Mateusz Windak, and Chris Gniady. Flow-Based Programming for IoT Leveraging Fog Computing. In *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 74–79. IEEE, 2017.

[SDSB19] Jan Seeger, Rohit A. Deshmukh, Vasil Sarafov, and Arne Bröring. Dynamic IoT Choreographies. *IEEE Pervasive Computing*, 18(1):19–27, 2019.

[SGK+10] Mirko Sonntag, Katharina Gorlach, Dimka Karastoyanova, Frank Leymann, and Michael Reiter. Process space-based scientific workflow enactment. *International Journal of Business Process Integration and Management*, 5(1):32–44, 2010.

[SHHA19] Ronny Seiger, Steffen Huber, Peter Heisig, and Uwe Aßmann. Toward a framework for self-adaptive workflows in cyber-physical systems. *Software & Systems Modeling*, 18(2):1117–1134, 2019.

[Shi07] Matthew Shields. Control- Versus Data-Driven Workflows. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 167–173. Springer, London, UK, 2007.

[SHLP05] M. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The Enterprise Service Bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, 2005.

[SHS15] Ronny Seiger, Steffen Huber, and Thomas Schlegel. PROtEUS: An Integrated System for Process Execution in Cyber-Physical Systems. In Khaled Gaaloul, Rainer Schmidt, Selmin Nurcan, Sérgio Guerreiro, and Qin Ma, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 214 of *Lecture Notes in Business Information Processing*, pages 265–280, Cham, Switzerland, 2015. Springer.

[SKG+09] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá De Souza, and Vlad Trifa. SOA-based integration of the internet of things in enterprise services. In *International Conference on Web Services (ICWS)*, pages 968–975. IEEE, 2009.

[SKNS15] Ronny Seiger, Christine Keller, Florian Niebling, and Thomas Schlegel. Modelling complex and flexible processes for smart cyber-physical environments. *Journal of Computational Science*, 10:137–148, 2015.

[SNP+15] Chayan Sarkar, Akshay Uttama Nambi S. N, R. Venkatesha Prasad, Abdur Rahim, Ricardo Neisse, and Gianmarco Baldini. DIAT: A Scalable Distributed Architecture for IoT. *IEEE Internet of Things Journal*, 2(3):230–239, 2015.

[Spa19] Spacebrew. What is Spacebrew?, 2019.

[SQV+14] Quan Sheng, Xiaoqiang Qiao, Athanasios Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014.

[SS15] Ahmed Safwat and M. B. Senousy. Addressing Challenges of Ultra Large Scale System on Requirements Engineering. In *International Conference on Communications, Management, and Information Technology (ICCMIT)*, International Conference on Communications, management, and Information technology (ICCMIT'2015), pages 442–449, 2015.

[SSC+17] Mengyu Sun, Zhensheng Shi, Shengjun Chen, Zhangbing Zhou,

and Yucong Duan. Energy-Efficient Composition of Configurable Internet of Things Services. *IEEE Access*, 5:25609–25622, 2017.

[TG18] Martin Törngren and Paul T. Grogan. How to Deal with the Complexity of Future Cyber-Physical Systems? *Designs*, 2(4):1–16, 2018.

[TJLG17] Samir Tata, Rakesh Jain, Heiko Ludwig, and Sandeep Gopisetty. Living in the Cloud or on the Edge: Opportunities and Challenges of IOT Application Architecture. In *International Conference on Services Computing (SCC)*, pages 220–224. IEEE, 2017.

[TKY+17] Yuuichi Teranishi, Takashi Kimata, Hiroaki Yamanaka, Eiji Kawai, and Hiroaki Harai. Dynamic Data Flow Processing in Edge Computing Environments. In *Annual Computer Software and Applications Conference (COMPSAC)*, pages 935–944. IEEE, 2017.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, Hoboken, NJ, USA, 1st edition, 2009.

[TSWH07] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The Triana Workflow Environment: Architecture and Applications. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 320–339. Springer, London, UK, 2007.

[TVS18] Kleanthis Thramboulidis, Danai C. Vachtsevanou, and Alexandros Solanos. Cyber-physical microservices: An IoT-based framework for manufacturing systems. In *Industrial Cyber-Physical Systems (ICPS)*, pages 232–239. IEEE, 2018.

[UGBBM+17] A. Urbieta, A. González-Beltrán, S. Ben Mokhtar, M. Anwar Hossain, and L. Capra. Adaptive and context-aware service composition for IoT-based smart cities. *Future Generation Computer Systems*, 76:262–274, 2017.

[Ven17] Yde Venema. Temporal Logic. In *The Blackwell Guide to Philosophical Logic*, pages 203–223. Wiley Publishing, Hoboken, NJ, USA, 2017.

[VGS+13] Panagiotis Vlacheas, Raffaele Giaffreda, Vera Stavroulaki, Dimitris Kelaidonis, Vassilis Foteinos, George Poulios, Panagiotis Demestichas, Andrey Somov, Abdur Rahim Biswas, and Klaus Moessner. Enabling smart cities through a cognitive management framework for the internet of things. *IEEE Communications Magazine*, 51(6):102–111, 2013.

[VN17] Asrin Vakili and Nima Jafari Navimipour. Comprehensive and systematic review of the service composition mechanisms in the cloud environments. *Journal of Network and Computer Applications*, 81:24–36, 2017.

[WAS+16] Andreas Weiß, Vasilios Andrikopoulos, Santiago Gómez Sáez, Michael Hahn, and Dimka Karastoyanova. ChorSystem: A Message-Based System for the Life Cycle Management of Choreographies. In Christophe Debruyne, Hervé Panetto, Robert Meersman, Tharam Dillon, eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences*, volume 10033 of *Lecture Notes in Computer Science*, pages 503–521, Cham, Switzerland, 2016. Springer.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–105, 1991.

[WLB09] Jian Wu, Qianhui Liang, and Elisa Bertino. Improving Scalability of Software Cloud for Composite Web Services. In *International Conference on Cloud Computing (CLOUD)*, pages 143–146. IEEE, 2009.

[WML08] Daniel Wutke, Daniel Martin, and Frank Leymann. Model and infrastructure for decentralized workflow enactment. In *ACM Symposium on Applied Computing (SAC)*, pages 90–94. ACM, 2008.

[WRS18] Danny Weyns, Gowri Sankar Ramachandran, and Ritesh Kumar Singh. Self-managing Internet of Things. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*,

volume 10706 of *Lecture Notes in Computer Science*, pages 67–84, Cham, Switzerland, 2018. Springer.

[WSJ15] Roy Want, Bill N. Schilit, and Scott Jenson. Enabling the Internet of Things. *Computer*, 48(1):28–35, 2015.

[WYG+17] Zhenyu Wen, Renyu Yang, Peter Garraghan, Tao Lin, Jie Xu, and Michael Rovatsos. Fog Orchestration for Internet of Things Services. *IEEE Internet Computing*, 21(2):16–24, 2017.

[WZY+19] Shangguang Wang, Ao Zhou, Mingzhe Yang, Lei Sun, Ching-Hsien Hsu, and Fangchun Yang. Service Composition in Cyber-Physical-Social Systems. *IEEE Transactions on Emerging Topics in Computing*, pages 1–11, 2019. Advance online publication.

[XH16] Yi Xu and Abdelsalam Helal. Scalable Cloud–Sensor Architecture for the Internet of Things. *IEEE Internet of Things Journal*, 3(3):285–298, 2016.

[XHL14] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, 2014.

[XV14] Li Da Xu and Wattana Viriyasitavat. A Novel Architecture for Requirement-Oriented Participation Decision in Service Workflows. *IEEE Transactions on Industrial Informatics*, 10(2):1478–1485, 2014.

[YLC14] Rong Yang, Bing Li, and Can Cheng. A Petri Net-Based Approach to Service Composition and Monitoring in the IOT. In *Asia-Pacific Services Computing Conference (APSCC)*, pages 16–22. IEEE, 2014.

[ZBDH06] Johannes Maria Zaha, Alistair Barros, Marlon Dumas, and Arthur ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162, Berlin, Germany, 2006. Springer.

[ZBL17] Michael Zimmermann, Uwe Breitenbücher, and Frank Leymann. A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 121–131. SciTePress, 2017.

[ZGLB10] Michele Zorzi, Alexander Gluhak, Sebastian Lange, and Alessandro Bassi. From today's INTRAnet of things to a future INTERnet of things: a wireless- and mobility-related view. *IEEE Wireless Communications*, 17(6):44–51, 2010.

[Zim17] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, 2017.

[ZWF07] Yong Zhao, Michael Wilde, and Ian Foster. Virtual Data Language: A Typed Workflow Notation for Diversely Structured Scientific Data. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 258–275. Springer, London, UK, 2007.

[ZZLH18] Zhangbing Zhou, Deng Zhao, Lu Liu, and Patrick C. K. Hung. Energy-aware composition for wireless sensor networks as a service. *Future Generation Computer Systems*, 80:299–310, 2018.

Appendix A

Permission from Publishers

This Appendix presents the permission from publishers for reusing the published papers that this thesis collects.

A.1 Permission to Reuse Content from IEEE Publications

This section shows the permission from IEEE for reusing the publications [AL17b], [AL17a], [AL19c], [AL19a] and [AL18b].

A.2 Permission to Reuse Content from Springer Publications

This section presents the permission from Springer for reusing the publication [AL18a].

SPRINGER NATURE LICENSE
TERMS AND CONDITIONS

Aug 13, 2019

This Agreement between The University of Manchester -- Damian Arellanes ("You") and Springer Nature ("Springer Nature") consists of your license details and the terms and conditions provided by Springer Nature and Copyright Clearance Center.

License Number	4647240909926
License date	Aug 13, 2019
Licensed Content Publisher	Springer Nature
Licensed Content Publication	Springer eBook
Licensed Content Title	Algebraic Service Composition for User-Centric IoT Applications
Licensed Content Author	Damian Arellanes, Kung-Kiu Lau
Licensed Content Date	Jan 1, 2018
Type of Use	Thesis/Dissertation
Requestor type	academic/university or research institute
Format	print and electronic
Portion	full article/chapter
Will you be translating?	no
Circulation/distribution	>50,000
Author of this Springer Nature content	yes
Title	PhD Student
Institution name	The University of Manchester
Expected presentation date	Sep 2019
Requestor Location	The University of Manchester Oxford Road Manchester, M139PL United Kingdom Attn: The University of Manchester
Total	0.00 GBP

Terms and Conditions

Springer Nature Customer Service Centre GmbH
Terms and Conditions

This agreement sets out the terms and conditions of the licence (the **Licence**) between you and **Springer Nature Customer Service Centre GmbH** (the **Licensor**). By clicking 'accept' and completing the

transaction for the material (**Licensed Material**), you also confirm your acceptance of these terms and conditions.

1. Grant of License

1. 1. The Licensor grants you a personal, non-exclusive, non-transferable, world-wide licence to reproduce the Licensed Material for the purpose specified in your order only. Licences are granted for the specific use requested in the order and for no other use, subject to the conditions below.

1. 2. The Licensor warrants that it has, to the best of its knowledge, the rights to license reuse of the Licensed Material. However, you should ensure that the material you are requesting is original to the Licensor and does not carry the copyright of another entity (as credited in the published version).

1. 3. If the credit line on any part of the material you have requested indicates that it was reprinted or adapted with permission from another source, then you should also seek permission from that source to reuse the material.

2. Scope of Licence

2. 1. You may only use the Licensed Content in the manner and to the extent permitted by these Ts&Cs and any applicable laws.

2. 2. A separate licence may be required for any additional use of the Licensed Material, e.g. where a licence has been purchased for print only use, separate permission must be obtained for electronic re-use. Similarly, a licence is only valid in the language selected and does not apply for editions in other languages unless additional translation rights have been granted separately in the licence. Any content owned by third parties are expressly excluded from the licence.

2. 3. Similarly, rights for additional components such as custom editions and derivatives require additional permission and may be subject to an additional fee. Please apply to Journalpermissions@springernature.com/bookpermissions@springernature.com for these rights.

2. 4. Where permission has been granted **free of charge** for material in print, permission may also be granted for any electronic version of that work, provided that the material is incidental to your work as a whole and that the electronic version is essentially equivalent to, or substitutes for, the print version.

2. 5. An alternative scope of licence may apply to signatories of the STM Permissions Guidelines, as amended from time to time.

3. Duration of Licence

3. 1. A licence for is valid from the date of purchase ('Licence Date') at the end of the relevant period in the below table:

Scope of Licence	Duration of Licence
Post on a website	12 months
Presentations	12 months
Books and journals	Lifetime of the edition in the language purchased

4. Acknowledgement

4. 1. The Licensor's permission must be acknowledged next to the Licenced Material in print. In electronic form, this acknowledgement must be visible at the same time as the figures/tables/illustrations or abstract, and must be hyperlinked to the journal/book's homepage. Our required acknowledgement format is in the Appendix below.

5. Restrictions on use

5. 1. Use of the Licensed Material may be permitted for incidental promotional use and minor editing privileges e.g. minor adaptations of single figures, changes of format, colour and/or style where the adaptation is credited as set out in Appendix 1 below. Any other changes including but not limited to, cropping, adapting, omitting material that affect the meaning, intention or moral rights of the author are strictly prohibited.

5. 2. You must not use any Licensed Material as part of any design or trademark.

5. 3. Licensed Material may be used in Open Access Publications (OAP) before publication by Springer Nature, but any Licensed Material must be removed from OAP sites prior to final publication.

6. Ownership of Rights

6. 1. Licensed Material remains the property of either Licensor or the relevant third party and any rights not explicitly granted herein are expressly reserved.

7. Warranty

IN NO EVENT SHALL LICENSOR BE LIABLE TO YOU OR ANY OTHER PARTY OR ANY OTHER PERSON OR FOR ANY SPECIAL, CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ARISING OUT OF OR IN CONNECTION WITH THE DOWNLOADING, VIEWING OR USE OF THE MATERIALS REGARDLESS OF THE FORM OF ACTION, WHETHER FOR BREACH OF CONTRACT, BREACH OF WARRANTY, TORT, NEGLIGENCE, INFRINGEMENT OR OTHERWISE (INCLUDING, WITHOUT LIMITATION, DAMAGES BASED ON LOSS OF PROFITS, DATA, FILES, USE, BUSINESS OPPORTUNITY OR CLAIMS OF THIRD PARTIES), AND
WHETHER OR NOT THE PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY PROVIDED HEREIN.

8. Limitations

8. 1. ***BOOKS ONLY:***Where **'reuse in a dissertation/thesis'** has been selected the following terms apply: Print rights of the final author's accepted manuscript (for clarity, NOT the published version) for up to 100 copies, electronic rights for use only on a personal website or institutional repository as defined by the Sherpa guideline (www.sherpa.ac.uk/romeo/).

9. Termination and Cancellation

9. 1. Licences will expire after the period shown in Clause 3 (above).

9. 2. Licensee reserves the right to terminate the Licence in the event that payment is not received in full or if there has been a breach of this agreement by you.

Appendix 1 — Acknowledgements:

For Journal Content:
Reprinted by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication)

For **Advance Online Publication papers:**
Reprinted by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication), advance online publication, day month year (doi: 10.1038/sj.[JOURNAL ACRONYM].)

For Adaptations/Translations:
Adapted/Translated by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication)

Note: For any republication from the British Journal of Cancer, the following credit line style applies:

Reprinted/adapted/translated by permission from [**the Licensor**]: on behalf of Cancer Research UK: : [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT]** (year of publication)

For **Advance Online Publication** papers:
Reprinted by permission from The [**the Licensor**]: on behalf of Cancer Research UK: [**Journal Publisher** (e.g. Nature/Springer /Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication), advance online publication, day month year (doi: 10.1038/sj.[JOURNAL ACRONYM])

For Book content:
Reprinted/adapted by permission from [**the Licensor**]: [**Book Publisher** (e.g. Palgrave Macmillan, Springer etc) [**Book Title**] by [**Book author**(s)] [**COPYRIGHT**] (year of publication)

Other Conditions:

Version 1.2