Internet of Things: From Small-to Large-Scale Orchestration

Charles Consel, Milan Kabáč

Internet of Things: From Small- to Large-Scale Orchestration

Charles Consel
Bordeaux Institute of Technology / Inria
Bordeaux, France
email: charles.consel@inria.fr

Milan Kabáč
Imperial College
London, UK
email: m.kabac@imperial.ac.uk

***Abstract*—The domain of Internet of Things (IoT) is rapidly expanding beyond research, and becoming a major industrial market with such stakeholders as major manufacturers of chips and connected entities (*i.e.,* things), and fast-growing operators of wide-area networks. Importantly, this emerging domain is driven by applications that leverage an IoT infrastructure to provide users with innovative, high-value services. IoT infrastructures range from small scale (*e.g.,* homes and personal health) to large scale (*e.g.,* cities and transportation systems).**

In this paper, we argue that there is a continuum between orchestrating connected entities in the small and in the large. We propose a unified approach to application development, which covers this spectrum. To do so, we examine the requirements for orchestrating connected entities and address them with domain-specific design concepts. We then show how to map these design concepts into dedicated programming patterns and runtime mechanisms.

Our work revolves around domain-specific concepts and notations, integrated into a tool-based design methodology and dedicated to develop IoT applications. We have applied our work across a spectrum of infrastructure sizes, ranging from an automated pilot in avionics, to an assisted living platform for the home of seniors, to a parking management system in a smart city.

***Keywords*-Internet of things, domain-specific languages, programming frameworks, MapReduce, orchestration**

I. Introduction

Internet of Things (IoT) is a rapidly emerging domain, spanning a wide range of application areas, including homes, cities, environment, energy systems, retail, logistics, industry, agriculture, and health [1]. This domain is supported by a new breed of infrastructure operators (*e.g.,* Sigfox [2], Lora [3]) that provide a wide area network and application-oriented services. Such IoT infrastructures have been leveraged by a host of companies that have developed economically viable applications to manage parking lots city-wide [4], to supervise wide-area transportation systems [5], to monitor offshore oil production platforms [6], *etc.* These applications have in common that they process information produced by a range of entities, connected to a large-scale, global network infrastructure. In a typical IoT infrastructure, entities have a unique identity, as well as network, computing and storage capabilities. Depending on their purpose, they also offer specific sensing and actuating functionalities (*e.g.,* pollution, motion, luminosity, humidity).

A key challenge of IoT is its crosscutting nature, spanning such areas as embedded systems, networking, distributed systems, security, pervasive computing, and software engineering. As a result, developing applications requires to gather many dimensions of expertise, in addition to coping with the various scales of entities that need to be *orchestrated* (*i.e.,* structured, organized and managed) to turn received data into actionable information.

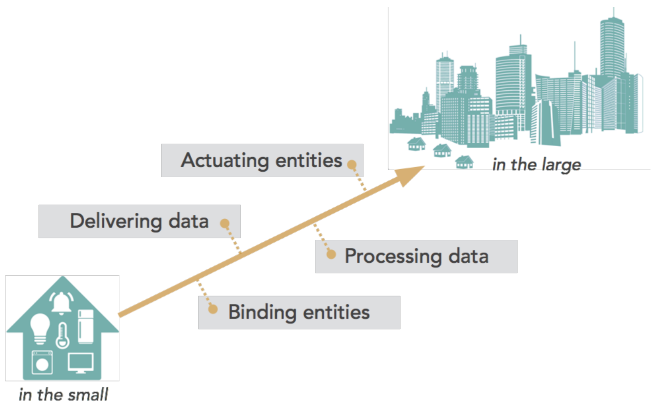

Figure 1: The continuum between small and large-scale IoT orchestration.

In this paper, we argue that there is a continuum between orchestrating IoT entities in the small and in the large (Figure 1), and propose a unified approach to developing IoT applications. We examine the requirements for orchestrating IoT entities and address them with domain-specific design concepts and notations. We then show how to map these design concepts into dedicated programming patterns and runtime mechanisms. To achieve this mapping, we introduce a Domain-Specific Language dedicated to the design of IoT applications. An IoT design is processed by a compiler that produces a customized programming framework in a host (mainstream) programming language, bridging the gap between designing in a domain-specific language and programming in a mainstream language. The generated programming framework is then used to guide and support the implementation of the design. As such, this generative programming approach allows to factorize the many dimensions of expertise at the compilation level, thus raising the level of abstraction of the IoT application development process.

More specifically, we introduce a paradigm that captures common application design patterns of the IoT domain (Section II). We propose declarative constructs that abstract over the heterogeneity of IoT entities (Section III). Then, we identify four key domain-specific activities of an application orchestrating IoT entities: binding entities, delivering data, processing data, and actuating entities (Section IV). We present design declarations that capture these four activities. Finally, we show how the proposed domain-specific design language can be integrated into a mainstream programming language (Section V).

Our domain-specific language approach has been implemented and takes the form of a tool-based design methodology, dedicated to develop IoT applications [7], [8]. It has been applied across a spectrum of infrastructure sizes and application areas, ranging from an automated pilot in avionics [9], to an assisted living platform for the home of seniors [10], to a parking management system in a smart city [11].

II. An IoT-Specific Design Paradigm

Conceptually, most IoT applications follow the *Sense-Compute-Control* (SCC) paradigm, promoted by Taylor *et al.* [12], and consist of an iterative process, as depicted in Figure 2: 1) an environment (*e.g.,* physical) is *sensed* to collect some data; 2) these data are processed to *compute* actionable information; and 3) this information is used to issue actions, impacting and *controlling* the environment. Following the domain-specific approach, we introduce a design language, named *DiaSpec* [13], [8], which decomposes the SCC paradigm into SCC-specific constructs. Let us, examine the concepts included in DiaSpec that match the SCC model (Figure 2). To sense the environment, *devices* are declared and include a source facet for collecting data. (Note that devices may either correspond to hardware entities or services.) Sensed data are passed to *context* software components to be processed and eventually turned into actionable information. This information is passed to *controller* software components when the environment needs to be impacted; controllers compute effects and issue corresponding actions to devices (*i.e.,* their action facet).

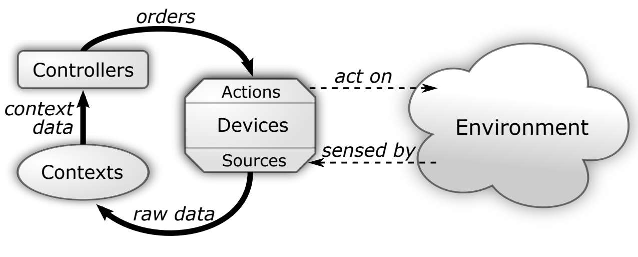

Figure 2: The Sense-Compute-Control paradigm.

We illustrate this IoT-specific design paradigm with two applications, used throughout this paper: a cooker monitoring application and a parking management application.

Cooker monitoring. This small-scale orchestrating application ensures the home safety for older adults by detecting when the cooker stays on beyond a time threshold and notifies the user. If this situation occurs, the user may decide to turn off the cooker remotely through a dedicated TV prompter. For the sake of simplicity, this application has a rudimentary behavior and only considers a small number of sensors and actuators; the graphical view of its SCC design is depicted in Figure 3. Conceptually, this view decomposes an application into components (devices, contexts, controllers) and defines how they form a functional chain from device sources to device actions.

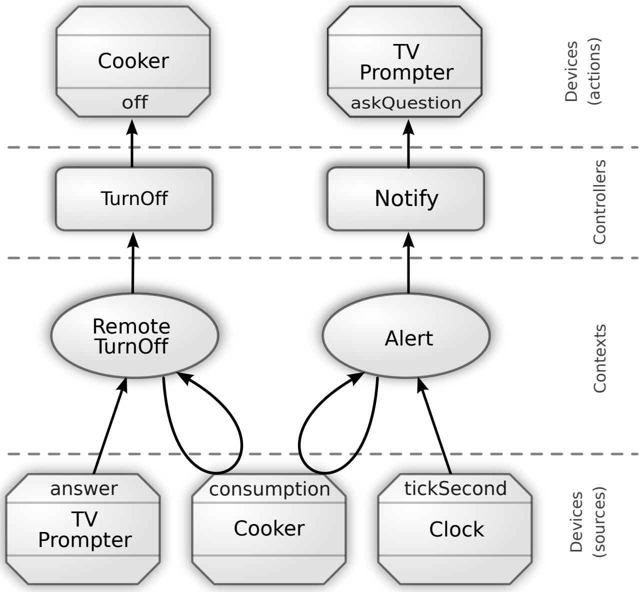

Figure 3: Graphical view of the design of the cooker monitoring application.

Let us illustrate this view with the first functional chain of the cooker monitoring application on the right-hand side of Figure 3; the `Clock` device periodically triggers the `Alert` context, which is subscribed to this event (noted by a straight arrow). When triggered, this context then queries the `consumption` source of the `Cooker` device (noted by a loop arrow). If the cooker has been on for too long, the `Alert` context invokes the `Notify` controller, which computes a notification to be issued to the `TVPrompter` device via the `askQuestion` operation. On the left-hand side of Figure 3, the source facet of the `TVPrompter` triggers the `RemoteTurnOff` context when the user has supplied a response to the notification. In this situation, the `RemoteTurnOff` context queries the current consumption level from the `Cooker` to ensure that the cooker is still on before turning it off, if the user's response instructed such action. The `TurnOff` controller is then invoked and issues an `off` action to the `Cooker` device.

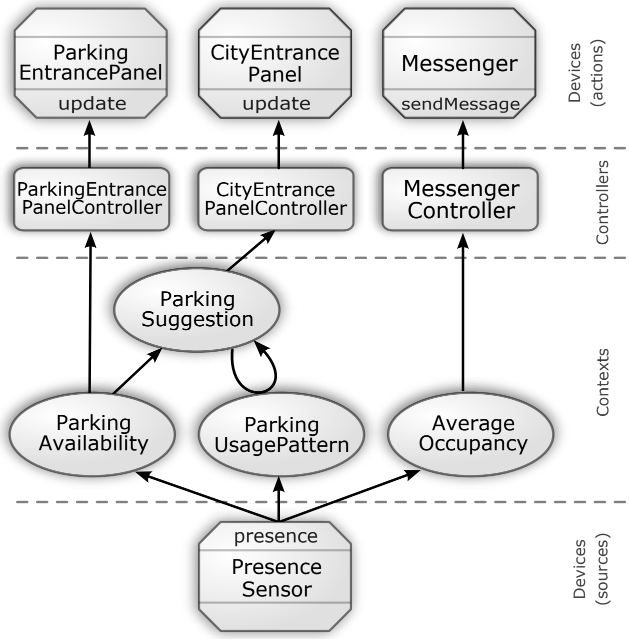

Figure 4: Graphical view of the design of the parking management application.

Parking management. The SCC design of this large-scale orchestrating application is graphically represented in Figure 4. Each parking space is equipped with a `PresenceSensor` device, which produces occupancy statuses via its `presence` source to subscribed context components, namely `ParkingAvailability`, `ParkingUsagePattern` and `AverageOccupancy`. The `ParkingAvailability` context keeps track of the number of available parking spaces in parking lots and, in turn, publishes these values periodically to the `ParkingEntrancePanel` controller. This controller is in charge of refreshing the number of available parking spaces displayed on the entrance screens via the `update` action. The `ParkingSuggestion` context combines information from `ParkingAvailability` with usage patterns of parking lots supplied by `ParkingUsagePattern`. This combination of information is processed to produce parking suggestions, passed to `CityEntrancePanelController` in charge of displaying information to drivers on screens spread across the city. The `AverageOccupancy` context computes presence values for each parking lot, averaged over a period of 24 hours for management purposes. This information is passed to the `MessengerControler` and to the `sendMessage` action of the `Messenger` device.

III. Abstracting Over Heterogeneous Entities

To cope with the heterogeneity of entities, DiaSpec provides the `device` construct to declare their functionalities, abstracting over their implementation or hardware specificities. This is illustrated in Figure 5 by the device declarations of the cooker monitoring application. These declarations consist of `source` and `action` facets depending of the functionalities to be described. A source can be *indexed* to distinguish between values. For example, the `answer` source in the `Prompter` device is indexed to allow to match a response, obtained from the `answer` source, with a question, issued by the `askQuestion` action.

```
device Clock {
  source tickSecond as Integer;
  source tickMinute as Integer;
  source tickHour as Integer;
}

device Cooker {
  source consumption as Float;
  action On;
  action Off;
}

device Prompter {
  source answer as String indexed by questionId as String;
  action askQuestion;
}
```

Figure 5: Device declarations for the cooker monitoring application.

Figure 6 shows the device declarations for the parking management application. To account for the location of presence sensors, their declaration includes an attribute defining their respective parking lot; this attribute is introduced via the `attribute` construct. When the IoT infrastructure is actually deployed, the location of the sensors are registered using this attribute. Also, as can be noticed for the declaration of display panel variations (`DisplayPanel`, `ParkingEntrancePanel`, `CityEntrancePanel`), DiaSpec entities can be declared hierarchically to inherit attributes and/or operations, as is done in object-oriented languages.

A concrete entity (*e.g.,* a device) needs to conform to the interface and implement the sources and action operations. This task can be viewed as implementing a device driver. As discussed in the next section, a concrete device is required to implement three data delivery modes to match the range of context usages of applications.

Also, note that device declarations are factorized and form a taxonomy dedicated to a given area, used across applications. For example, we created a taxonomy of entities for the domain of assisted living. More details can be found elsewere [7], [8].

Finally, note that the device declarations can serve as a vehicle to express non-functional information about an entity. We illustrated this approach by introducing annotations in

```
device PresenceSensor  {
   attribute parkingLot as ParkingLotEnum;
   source presence as Boolean;
}

device DisplayPanel  {
   action update(status as String);
}

device ParkingEntrancePanel extends DisplayPanel {
   attribute location as ParkingLotEnum;
}

device CityEntrancePanel extends DisplayPanel {
   attribute location as CityEntranceEnum;
}

device Messenger {
   action sendMessage(message as String);
}

enumeration ParkingLotEnum {
  A22, B16, D6,...
}

enumeration CityEntranceEnum {
  NORTH_EAST_14Y, SOUTH_EAST_1A,...
}
```

Figure 6: Device declarations for the parking management application.

declarations to describe potential errors [14] or quality of service constraints [15] at the level of devices and related contexts and controllers. This extended approach to software design was applied to the avionics domain [9].

IV. IoT-Specific Activities

Based on the literature and practical experience, we have identified four fundamental activities that uniformly capture entity orchestration across the orchestration scale: 1) binding entities, 2) delivering data, 3) processing data, and 4) actuating entities. Our goal is to map these four activities into dedicated language concepts. We first present these activities and then exemplify them with our two IoT applications.

Binding entities. This activity is concerned with how entities are bound to the environment. For example, when sensors are deployed in a house or in a parking lot, each sensor needs to be registered and attribute values defined (*e.g.,* setting a room for a sensor location). Another aspect of entity binding regards how entities are bound to an application. For example, an application controlling a room temperature requires to be bound to a temperature sensor. Depending on the area and orchestration scale, entity binding can occur at configuration time, deployment time, launch time, or runtime.

Delivering data. This activity is concerned with how data are delivered to an application. We propose three data delivery models, inspired by the domain of wireless sensor networks [16]: periodic, even driven and query driven. These three models cover most practical cases from small-scale to large-scale orchestration. As shown in Figures 5 and 6, a device declaration does not restrict client context components to use any of the three models. In fact, an implementation of a device is required to implement the three data delivery modes, providing flexibility to client applications.

Processing data. In general, small-scale orchestration is not concerned with high-volume of data. However, large-scale orchestration may involve masses of sensors, gathering large amounts of data. This situation may require efficient processing strategies that can leverage our design approach to expose parallelism.

Actuating entities. This activity is the dual of data delivery. We assume that entities are issued actions by the application with a simple call mechanism.

Let us now illustrate these four fundamental activities with our working examples for small and large-scale IoT orchestration.

1) Small-Scale Orchestration: Consider the DiaSpec design of the cooker monitoring application, shown in Figure 7. This application design consists of the two functional chains, strictly mimicking the graphical view displayed in Figure 3. The `Alert` context is defined as subscribing, via the `when provided clause`, to the `tickSecond` source of the `Clock` device; this corresponds the event-driven data delivery model. Once triggered, this context queries the electric consumption of the cooker; it uses the query-driven delivery model. The design of the `Alert` context specifies that this component may publish a result, if indeed, the cooker has remained turned on for too long. Otherwise, this context component does not produce any value, hence the `maybe publish` clause. The rest of the design follows the presentation of this application given earlier. Notice that contexts can invoke other contexts or controllers, but controllers cannot invoke context components in conformance with the SCC paradigm. Finally, controller declarations use the `do` construct to specify that the controller implementation performs one of more operations on the action facet of a device. For example, the `Notify` controller sends a notification to the user.

Let us examine the four activities illustrated by this IoT application. As is usually done in pervasive computing, we assume that entities are bound to the application at runtime, enabling much flexibility. As a result, the entity discovery operation is invoked in the implementation of the context and controller components (see Section V), as opposed to statically in the design. Data from entities are delivered using two models: event and query. Note that the `tickSecond` source could have also been delivered using a periodic model. For small-scale IoT applications, the volume of data to be processed is modest and does not require specific computing or networking capabilities beyond what is available in a standard home. Actuating entities does not require any specific treatment either.

```
context Alert as Integer {
  when provided tickSecond from Clock
  get currentElectricConsumption from Cooker
  maybe publish;
}

controller Notify {
  when provided Alert
  do askQuestion on TvPrompter;
}

context RemoteTurnOff as Boolean {
  when provided answer from TvPrompter
  get currentElectricConsumption from Cooker
  maybe publish;
}

controller TurnOff {
  when provided RemoteTurnOff
  do off on Cooker;
}
```

Figure 7: Design of the cooker monitoring application.

```
1  context ParkingAvailability as Availability[] {
2    when periodic presence from PresenceSensor <10 min>
3    grouped by parkingLot
4    with map as Boolean reduce as Integer
5    always publish;
6  }
7
8  context ParkingUsagePattern as UsagePattern[] {
9    when periodic presence from PresenceSensor <1 hr>
10   grouped by parkingLot
11   no publish;
12
13   when required;
14 }
15
16 context AverageOccupancy as ParkingOccupancy[] {
17   when periodic presence from PresenceSensor <10 min>
18   grouped by parkingLot every <24 hr>
19   always publish;
20 }
21
22 context ParkingSuggestion as ParkingLotEnum[] {
23   when provided ParkingAvailability
24   get ParkingUsagePattern
25   always publish;
26 }
27
28 controller ParkingEntrancePanelController {
29   when provided ParkingAvailability
30   do udpate on ParkingEntrancePanel;
31 }
32
33 controller CityEntrancePanelController {
34   when provided ParkingSuggestion
35   do update on CityEntrancePanel;
36 }
37
38 controller MessengerController {
39   when provided AverageOccupancy
40   do sendMessage on Messenger;
41 }
42
43 structure Availability {
44   parkingLot as ParkingLotEnum;
45   count as Integer;
46 }
47
48 structure UsagePattern {
49   parkingLot as ParkingLotEnum;
50   level as UsagePatternEnum;
51 }
52
53 structure ParkingOccupancy {
54   parkingLot as ParkingLotEnum;
55   occupancy as Float;
56 }
57
58 enumeration UsagePatternEnum { HIGH, MODERATE, LOW }
```

Figure 8: Design of the parking management application.

2) Large-Scale Orchestration: We now turn to the presentation of DiaSpec for the design of large-scale IoT applications, leveraging our work from the DiaSwarm project [17]. We first present how our design language is evolved to bind (or discover) masses of sensors. Our approach provides the IoT developer with design constructs to declare how sensor data should be gathered and presented to the application. These constructs involve parameters specific to the data delivery models (*e.g.,* time for the periodic model) and attributes specific to the application (*e.g.,* grouping sensors by parking lots). To illustrate this approach consider the design of the parking management application in Figure 8. The `ParkingAvailability` context specifies in line 2 that presence statuses should be delivered periodically (every 10 minutes) and line 3 requires these values to be grouped by an application-specific attribute, namely, parking lots. Specifically, every 10 minutes, all presence sensor statuses of all parking lots are delivered to the `ParkingAvailability` context. To ease the processing, the `grouped by` construct (line 3) requires these statuses to be split into (or grouped by) parking lots.

Large-scale IoT applications often require to cope with large amounts of data, calling for efficient processing strategies. To do so, our IoT design-driven approach can be leveraged in two ways: first, design declarations are used by the compiler to generate the customized programming framework (see Section V); second, declarations can be supplemented with information to expose parallelism and allow efficient processing of large datasets. This last approach is used to introduce the MapReduce programming model [18] in the design of large-scale IoT applications. Specifically, we leverage the `grouped by` construct because it partitions a large set of gathered data. This partitioning can then be applied to the MapReduce programming model by splitting data processing into a Map and a Reduce phase. To express these two phases at the design level, the `grouped by` construct is extended with an optional clause that specifies what types of values are produced by both the Map and Reduce phases. An example of the resulting construct is displayed in lines 3 and 4 in Figure 8. The `ParkingAvailability` context includes a `grouped by` construct that declares a Map phase that processes `Boolean` values and a Reduce phase that produces an `Integer` value. The next section examines how these declarations are exploited to guide the developer to implement the context and controller components against a generated programming framework that parallelizes the Map and Reduce phases.

In the parking management application, as in most applications, actuators are handled as they are in small scale IoT applications: they are discovered with respect to attributes (*e.g.,* locations) and are invoked to perform specific operations. In constrast with sensors, there is no need for processing large datasets.

As shown in this section, a design approach can be general enough to cover the continuum from small to large-scale IoT applications. Let us now examine how designing in an IoT-specific language can be put in synergy with programming in a mainstream language.

V. Mainstream Programming Language Integration

Programming IoT applications is time consuming and requires a range of expertise. This task involves much boilerplate code needed for discovering devices, establishing network communications, managing the state data of underlying subsystems, *etc.* Writing this boilerplate code is error prone and requires intimate knowledge of the underlying layers, without losing track of the application requirements.

To ease the programming of orchestrating applications, our approach provides the developer with a design compiler that generates an application framework tailored to a given application design. Because the generated programming frameworks employ the *inversion of control* [19], implementing a design is devoted to implementing the declared contexts and controllers of an application, which are then called as required by the runtime system.

The current implementation of our approach generates programming frameworks in Java. However, it can be applied to any mainstream programming language regardless of the language paradigm, as shown by Van der Walt *et al.* [20]. The generated framework ensures conformance between design and programming, and allows the developer to concentrate exclusively on the application logic. Indeed, this generative approach greatly improves productivity as the amount of generated code may represent up to 80% of the resulting application code [8].

Let us now examine the code support generated from an application design using our two working examples. Programming frameworks generated from application design provide domain-specific functionalities, including entity binding, data gathering and component interaction. The code generator provides the programmer with an abstract class for each declared application component (*i.e.,* context, controller) or device. The generated abstract class needs to

```
1 public class Alert extends AbstractAlert {
2   @Override
3   public AlertValuePublishable onTickSecondFromClock(
4               TickSecondFromClock tickSecondFromClock,
5               DiscoverForTickSecondFromClock discover)  {
6         // TODO Auto-generated method stub}
7   }
8 }
```

Figure 9: An implementation of the `Alert` context of the cooker monitoring application.

be subclassed to implement the logic of a component. In this paper, we provide details on the code generated for context and controller components. Support for the implementation of devices is examined elsewhere [8].

A. Small-Scale Orchestration Support

Figure 9 presents the subclassing of an abstract class generated for the `Alert` context in the cooker monitoring application. The declaration of the `Alert` context (Figure 7, line 2 to 4) is mapped into the `onTickSecondFromClock` callback method, which is triggered every time the `Clock` device publishes a value (*i.e.,* every second according to the design declaration). The developer has to implement the generated method to introduce the application logic. The `tickSecondFromClock` parameter (line 4) provides information about the `Clock` device, including its attributes and the published source value. The `discover` parameter (line 5) exposes a specialized interface to querying the current consumption of the cooker. Finally, the resulting context value has to be wrapped using the generated method return type (*i.e.,* `AlertValuePublishable`) and may be passed to the `Notify` controller as specified by the `maybe publish` clause (Figure 7, line 4).

B. Large-Scale Orchestration Support

Figure 10 presents the implementation of the `ParkingAvailability` context for the parking management application. The generated code is colored in gray as in Figure 9; while the user-supplied code is displayed with a white background. The context keeps track of the number of available parking spaces in parking lots. As in the previous example, the context implementation is done by subclassing the generated abstract class. In addition, the `ParkingAvailability` context relies on the MapReduce model (Figure 8, line 4) for processing large datasets, which requires the developer to implement the `MapReduce` interface (line 3) provided by the generated programming framework.

In conformance with the MapReduce programming model, the Map function is passed a key and a value, which correspond to the parking lot identifier (defined by the attribute of the `grouped by` directive) and a reading from the corresponding presence sensor. If there is no

```java
1  public class ParkingAvailability
2          extends AbstractParkingAvailability
3          implements MapReduce<ParkingLotEnum, Boolean,
4                               ParkingLotEnum, Boolean,
5                               ParkingLotEnum, Integer> {
6    @Override
7    public void map(ParkingLotEnum parkingLot,
8                            Boolean presence,
9         MapCollector<ParkingLotEnum, Boolean> collector) {
10
11     if(!presence)
12       collector.emitMap(parkingLot, true);
13   }
14
15   @Override
16   public void reduce(ParkingLotEnum parkingLot,
17                          List<Boolean> values,
18      ReduceCollector<ParkingLotEnum, Integer> collector) {
19
20     int sum = values.size();
21     collector.emitReduce(parkingLot, sum);
22   }
23
24   @Override
25   protected List<Availability> onPeriodicPresence(
26       Map<ParkingLotEnum, Integer> presenceByParkingLot) {
27
28     List<Availability> availabilityList =
29                          new ArrayList<Availability>();
30     for(Entry<ParkingLotEnum, Integer> parkingLot :
31                    presenceByParkingLot.entrySet()) {
32       Availability availability = new Availability(
33               parkingLot.getKey(), parkingLot.getValue());
34       availabilityList.add(availability);
35     }
36     return availabilityList;
37   }
38 }
```

Figure 10: An implementation of the ParkingAvailability context with MapReduce for the parking manager application.

presence detected, the `emitMap` method produces a key/-value result. Intermediate results from the Map phase are grouped into a list by the generated framework, and passed to the Reduce phase via the `values` parameter (line 17). The Reduce phase sums up the set of values associated with a given intermediate key (*i.e.,* parking lot) and emits the total availability per parking lot via the `emitReduce` method. Data resulting from MapReduce computations are passed to the `onPeriodicPresence` method through a map (line 26). Finally, the `onPeriodicPresence` method wraps the refined data into a list of values of type `Availability`, as specified by the context declaration. This list is consequently returned to subscribed components (*i.e.,* `ParkingEntrancePanelController`, `ParkingSuggestion`). As can be noticed, the generated programming framework exposes an interface that prevents the specificities of a target MapReduce implementation to percolate to the application logic. This separation simplifies the programming of applications and promotes reusability. Details on how an orchestrating application is combined with an actual implementation of MapReduce can be found elsewhere [11].

```java
public class ParkingEntrancePanelController extends
                   AbstractParkingEntrancePanelController {
  @Override
  protected void onParkingAvailability(Discover discover,
           ParkingAvailabilityValue parkingAvailability) {
    for(Availability availability :
                        parkingAvailability.getValue()) {
      String status = getStatus(availability);
      discover.parkingEntrancePanels().whereLocation(
              availability.getParkingLot()).update(status);
    }
  }
}
```

Figure 11: An implementation of the ParkingEntrancePanel controller.

Figure 11 presents the implementation of the `ParkingEntrancePanel` controller. Similar to a context, a controller is implemented by subclassing the generated abstract class. The generated abstract class ensures that the controller receives data from subscribed contexts in conformance with design declarations. The controller receives data from the `ParkingAvailability` context via the `onParkingAvailability` callback method. The role of this component is to display the availability of each parking lot on dedicated display panels (*i.e.,* `ParkingEntrancePanel`). This is done using the `discover` object, which comprises a set of proxies for invoking remote devices without the need for managing distributed systems details.

VI. Conclusion

The IoT domain encompasses a wide spectrum of areas, devices, requirements, and scales. To harness the possibilities of this domain, the many dimensions involved in developing IoT applications need to be factorized. To do so, we propose to leverage a domain-specific language approach where IoT-specific concepts, mechanisms and notations are abstracted from these dimensions. We have shown that a design language can capture the development of IoT applications along a continuum from small to large-scale infrastructures.

Our approach is instantiated with DiaSpec, an IoT-specific design language. We have shown that DiaSpec design declarations cover the four activities of IoT applications (binding entities, delivering data, processing data, and actuating entities) from small to large-scale infrastructures. We have argued that a range of implementation issues can be encapsulated in the compiler that processes design declarations and generates customized programming frameworks. Not only can this generative programming approach allow to introduce parallelism to handle large datasets from sensors, but it also guides and supports the implementation of the declared software components in a mainstream language such as Java.

This design language approach can be a vehicle to explore a number of challenges in the IoT domain. Can design declarations be used to match the requirements of an application with the resources of an infrastructure? The application requirements could be extracted (or estimated) from the design declarations; they could include devices, network bandwidth, and processing capability. What non-functional dimensions should be added to the design declarations to account for constraints of the IoT infrastructure? We illustrated this idea with parallel processing that was introduced at the design level. Other dimensions could address quality of service or device failure.

Acknowledgements

This paper revisits, aggregates and extends the work conducted in the Phoenix research group by a number of PhD students, postdocs, and collaborators: Emilie Balland, Benjamin Bertran, Julien Bruneau, Damien Cassou, Zoé Drey, Quentin Enard, Stéphanie Gatti, Hongyu, Guan, Henner Jakob, Wilfried Jouve, Julia Lawall, Nicolas Loriant, Julien Mercadal, Young-Joo Moon, and Paul van der Walt.

We also thank Nic Volanschi and Bernard Serpette for providing feedback on earlier versions of this paper.

References

[1] A. Bahga and V. Madisetti, *Internet of Things: A Hands-On Approach*. VPT, 2014.

[2] Sigfox, "Global cellular connectivity for IoT," Online, Accessed 31/3/2017, http://www.sigfox.com.

[3] LoRa, "Low Power Wide Area Network," Online, Accessed 31/3/2017, http://www.lora-alliance.org.

[4] Libelium, "Smart City project in Santander to monitor Parking Free Slots," Online, Accessed 31/3/2017, http://www.libelium.com/smart_santander_parking_smart_city.

[5] Y. Mizuno and N. Odake, "Current Status of Smart Systems and Case Studies of Privacy Protection Platform for Smart City in Japan," in *2015 Portland International Conference on Management of Engineering and Technology (PICMET)*, Aug 2015, pp. 612–624.

[6] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, "The internet of things: Mapping the value beyond the hype. mckinsey global institute," 2015.

[7] B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Balland, and C. Consel, "Diasuite: A tool suite to develop Sense/Compute/Control applications," *Science of Computer Programming*, vol. 79, pp. 39–51, 2014.

[8] D. Cassou, J. Bruneau, C. Consel, and E. Balland, "Toward a tool-based development methodology for pervasive computing applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1445–1463, 2012.

[9] Q. Enard, S. Gatti, J. Bruneau, Y.-J. Moon, E. Balland, and C. Consel, "Design-driven development of dependable applications: A case study in avionics," in *PECCS-3rd International Conference on Pervasive and Embedded Computing and Communication Systems*. SciTePress, 2013.

[10] C. Consel, L. Dupuy, and H. Sauzéon, "HomeAssist: An assisted living platform for aging in place based on an interdisciplinary approach," in *Proceedings of the 8th International Conference on Applied Human Factors and Ergonomics (AHFE 2017)*. Springer, 2017, (To appear).

[11] M. Kabáč and C. Consel, "Designing parallel data processing for large-scale sensor orchestration," in *13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC 2016)*, 2016.

[12] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[13] D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging software architectures to guide and verify the development of Sense/Compute/Control applications," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 431–440.

[14] J. Mercadal, Q. Enard, C. Consel, and N. Loriant, "A domain-specific approach to architecturing error handling in pervasive computing," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 47–61, 2010, (OOPSLA'10).

[15] S. Gatti, E. Balland, and C. Consel, "A step-wise approach for integrating QoS throughout software development," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2011, pp. 217–231.

[16] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, "A taxonomy of wireless micro-sensor network models," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, no. 2, pp. 28–36, 2002.

[17] M. Kabáč and C. Consel, "Orchestrating Masses of Sensors: A Design-Driven Development Approach," in *14th International Conference on Generative Programming: Concepts & Experience (GPCE'15)*, Pittsburgh, Pennsylvania, United States, Oct. 2015.

[18] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[19] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Communications of the ACM*, vol. 40, no. 10, pp. 32–38, 1997.

[20] P. Walt, C. Consel, and E. Balland, "Frameworks compiled from declarations: a language-independent approach," *Software: Practice and Experience*, 2016.