IoTLink: An Internet of Things Prototyping Toolkit

Ferry Pramudianto[1], Carlos Alberto Kamienski[2], Eduardo Souto[3], Fabrizio Borelli[4], Lucas L. Gomes[5], Djamel Sadok[6], Matthias Jarke[7]
[1]Fraunhofer FIT, Schloss Birlinghoven, St. Augustin, Germany
[2,4]Federal University of ABC (UFABC), Santo André, Brazil
[3]Federal University of Amazonas (UFAM), Manaus, Brazil
[5,6]Federal University of Pernambuco (UFPE), Recife, Brazil
[7]RWTH-Aachen, Ahornstr. 55, 52056 Aachen Germany
e-mail: [1]ferry.ferry@fit,fraunhofer.de, [2]cak@ufabc.edu.br, [3]esouto@icomp.ufam.edu.br, [4]fabrizio.borelli@ufabc.edu.br, [5]lucas.gomes@gprt.ufpe.br, [6]jamel@gprt.ufpe.br, [7]jarke@informatik.rwth-aachen.de

***Abstract*— The Internet of Things (IoT) application development is a complex task that requires a wide range of expertise. Currently, the IoT community lacks a development toolkit that enables inexperienced developers to develop IoT prototypes rapidly. Filling this gap, we propose a development toolkit based on a model-driven approach, called IoTLink. IoTLink allows inexperienced developers to compose mashup applications through a graphical domain-specific language that can be easily configured and wired together to create an IoT application. Through visual components, IoTLink encapsulates the complexity of communicating with devices and services on the internet and abstracts them as virtual objects that are accessible through different communication technologies. Consequently, it solves interoperability between heterogeneous IoT components. Based on the visual model, IoTLink is able to generate a complete Java project including an extendable Java code. In a controlled experiment, IoTLink was 42% faster than using a Java library and able to outperformed the Java library's user satisfactions.**

Keywords—model driven development, mashup, Internet of Things, code generation, development tool

I. INTRODUCTION

While the number connected things increases rapidly, the process of developing IoT system prototypes is still a complex task. It requires expertise in various fields, as developers have to deal with various technological challenges such as noise of various sensor components, network protocols and data format interoperability, storing and analyzing a huge amount of data. Additionally, the existing development platform used in IoT development are designed to support specific group of developers such as embedded development or enterprise application development. Consequently, to create a simple IoT prototype, developers are required to combine different disintegrated tools and programming platform. For instance, embedded C and model-driven development are often used for embedded system development [1, 2] since they could work very efficiently on devices that have very limited computing resources. Meanwhile integrating IoT to enterprise applications which often can be run on a powerful server or PCs often is done through middleware with newer programming languages such as Java and C# since they offer features that simplify developers' tasks, easier to maintain, and more forgiving since they utilize garbage collectors. In addition, developers are required to understand heterogeneous communication paradigms and protocols that are used by embedded devices, as well as enterprise applications.

This paper proposes a development toolkit, called IoTLink, based on model-driven development (MDD) approach that suggest system development should be done by defining the computation independent model (CIM) which is refined to Platform-Independent Model (PIM), and detailed in a platform-specific model (PSM) [3]. By utilizing a high-level model, IoTLink allows inexperienced developers to compose distributed devices and services into mashup [4] and visually define how the components are combined to represent "Things". The model can be transformed into a Java code, which can be extended by more experienced developers in a further phase of the development. The generated codes may also be run as a standalone application that exposes the domain objects through different protocols and serialization formats, which can be chosen during the modeling phase.

The remainder of the paper is organized as follows. Section 2 elaborates the related works including Internet of things definition and mashup development. Section 3 describes the implementation of IoTLink. Section 4 describes the evaluation using a case study and formal study, in section 5 we conclude our work and describe our plans for IoTLink in the future.

II. RELATED WORKS

A. Internet of things definition and context

The Internet of Things (IoT) is a concept in which devices and physical objects are connected and able to cooperate to achieve some goals. Initially, the term 'Internet of Things' was coined by Kevin Ashton when he worked on P&G's supply chain [1]. Since then IoT definitions have been conveyed from different

perspectives, inspired by diverse visions [2]. First, the "Things" vision focuses on enabling interaction between physical objects and users. Second, the network-oriented vision deals with various communication methods among devices, systems and their users. Third, the semantic oriented vision focuses on retrieving useful information from massive and inconsistent data generated by sensors and another kind of data providers to support the users.

The Internet of Things European Research Cluster (IERC) defines IoT as "A dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network" [3]. This definition highlights that IoT is not only concerned with the communication between physical and virtual world but also demands that the physical objects become smarter to accomplish autonomous systems that require very little to zero maintenance.

In terms of application development, high-level architectures and IoT middleware have been proposed which has been summarized extensively here [4, 5]. Among these approaches, Service Oriented Architecture (SOA) has become popular to ensure horizontal and vertical integrations among applications and devices.

Recently, cloud-based providers known as the Web of Things (WoT) such as Xively (www.xively.com), OpenSense (open.sen.se), and ThingWorx (www.thingworx.com) become quite popular as a platform to collect, aggregate, and visualize a large amount of sensor data. They provide an API store sensor data in the cloud, perform data analytics and visualize relevant information (e.g., geographical data). Some providers offer a mashup development tool for process and visualize sensor streams rapidly. However, only a few of them provide a support for integrating heterogeneous devices into the platform e.g. Xively provides libraries in several languages to consume their API from devices directly.

B. Mashup Development

Mashup development is a way of building web applications rapidly by aggregating different data sources on the web using a graphical development interface [6]. Yahoo! Pipes (Figure 1), and DERI Pipes [7] are examples of mashup development platforms that allow end-user developers to compose services by linking components with a high level of abstraction. Thus, mashup development is less flexible than conventional programming language since it reveals less technical details to ensure the simplicity of the development [6]. Because of this reason, mashup development generally targets end-users with minimal development experiences instead of expert developers. Usually, mashup development is supported by a visual tool such as shown in Figure 10.

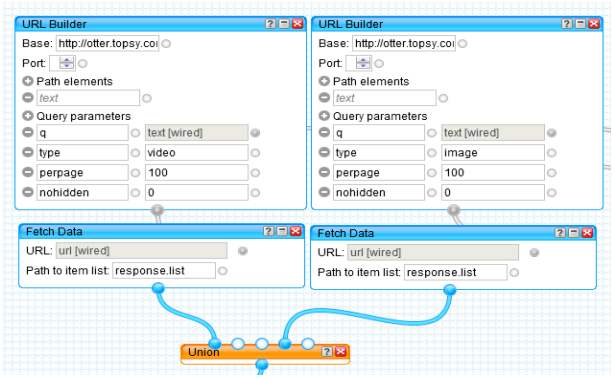

Figure 1. An example of a mashup service for querying a web service, developed on Yahoo! Pipes.

Figure 1 presents an example of a mashup development environment of the Yahoo! Pipes. It follows a Flow-Based Programming (FBP) [8] approach for composing the interaction between predefined modules. The outputs of one module can be connected to the input of other modules as long as their types are compatible. Otherwise, data transformation components must be introduced. A study evaluated the acceptance of Yahoo! Pipes' among eight students revealed a good acceptance and a fast learning curve [7]. Moreover, mashup development has been investigated to involve business users, who do not have extensive programming experience, to create and share customized business applications. This approach is proposed to reduce the bottleneck on the IT department, which has to implement different business requirements within a limited period [8, 9]. Allowing non-experts to participate in creating business applications may help overcoming the time to market demands, which has been increasingly becoming shorter.

An open source project from IBM called Node-RED (nodered.org) adopts a similar approach to develop IoT mashup that can be deployed on a PC as well as smaller platforms such as Raspberry PI (www.raspberrypi.org). Their approach relies purely on data flow abstractions unlike our approach that keeps the abstraction resembling physical objects, which is more natural to interact with from the application development perspective.

III. IOTLINK CONCEPT & IMPLEMENTATION

There has been some efforts to define IoT metamodel which suggest how physical objects could be represented by software services e.g., Ebbits (www.ebbits.eu) an European research project aims at developing IoT platform for business applications, IoT-A, a European research project aim at standardizing IoT architecture. IoT-A has investigated the different IoT architectures and concludes them as an IoT Architecture Reference Model (ARM) [9]. Figure 2 illustrates a simplified IoT-A metamodel. It shows that a physical object must be uniquely identifiable, has physical qualities that partly can be observed by sensors, and has some capabilities or services that could affect the environment. Physical

objects can be represented by virtual objects, which act as their proxy allowing applications to retrieve their states and consume their services.

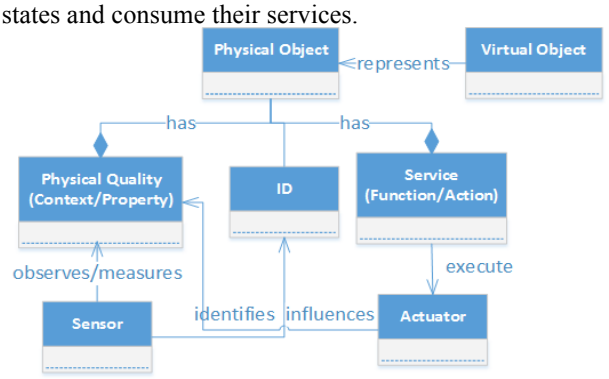

Figure 2. IoT Metamodel based on the IoT-A project [10]

Based on this conception, we designed IoTLink that allows developers to compose software representation of physical objects through a model-driven approach. We identify that IoTLink's platform-independent metamodel comprises four abstraction layers. The first layer abstracts the heterogeneous connections to physical sensors. This includes providing specific communication technologies and providing uniform interface for the component in other layers

The second layer is responsible for processing sensor data to determine the actual status of the physical objects. This layer is required since sensor hardware has physical limitations and may contain measurement noise. Thus, sometimes several types of sensor must be combined for sensing physical events. For instance, to measure the stress level, several bio-readings such as respiration rate, heart rates, skin conductance may be collected and through intelligent algorithms, the system could conclude the stress level [11]. Thus, this layer must provide sensor fusion modules, which can be used to pre-process and fuse sensor readings before these values can represent the actual state of a physical object.

The third layer is responsible for abstracting the domain objects that represent the "Things" and their attributes. We follow the object-oriented paradigm since most developers are already familiar with the concept. It requires the domain objects to be abstracted through classes.

The fourth layer is responsible for exposing the domain objects to the application logic, distributed applications, as well as persistence storage. Thus, it must provide a network interface and a specific data format that can be accessed and processed by distributed applications.

IoTLink allows developers to define the applications in a platform-independent model through visual notations, which then can be transformed into a platform-specific model. We decided to use Java to implement the platform-specific model since Java offers extensive open source components for software developments that ease the required efforts to implement IoTLink. Moreover, Java allows us to implement artifacts that can be directly compiled and used as proxies for the "Things".

Due to these considerations we define IoTLink's metamodel as a groundwork for the development of IoTLink (has been evaluated in [12]). The metamodel comprises the aforementioned abstraction layers and specific implementations of each layer. As depicted in Figure 3, several concrete connections are implemented and derived from the connections class. The link between connections and properties could go through concrete sensor fusion components. The virtual objects can be serialized through output components. The output components also allow external applications to interact with the virtual objects i.e. by consuming their services.

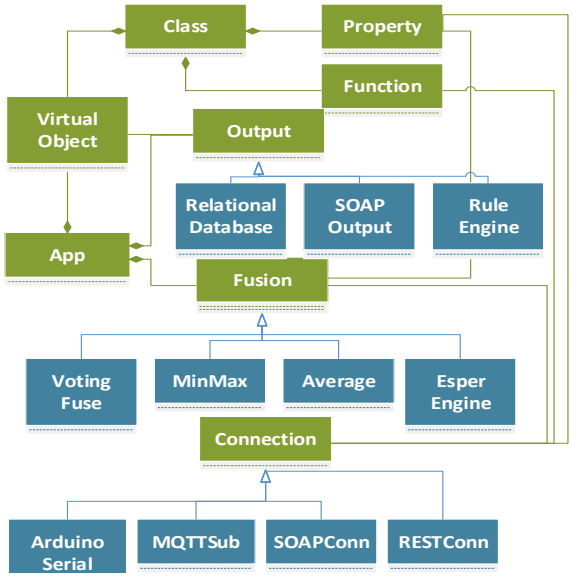

Figure 3. Logical view of the EMF meta-model

Based on this metamodel, we choose to provide our users a visual editor for defining concrete application models since the visual notations could enable inexperienced developers as our investigation section two shows. Using the visual editor, the users may choose each concrete components in the four layers for their applications and link them together. Then the model can be transformed into a Java implementation that can be extended by developers that are more experienced.

A. Implementation

IoTLink was developed by following a human-centered approach. First, a low-fidelity wireframe was developed using Balsamiq (www.balsamiq.com) and validated by 8 users using a cognitive walkthrough [13] approach to evaluate the metamodel as well as identify the usability problems of the whole approach. Based on this initial feedback, we improved the metamodel and the user interface design (e.g., sensor fusion is not necessarily required. Thus, the virtual object should be able to be linked with connections). After the metamodel and wireframe design was quite mature, a high fidelity

prototype of IoTLink was built, evaluated, and the result has been published in [12].

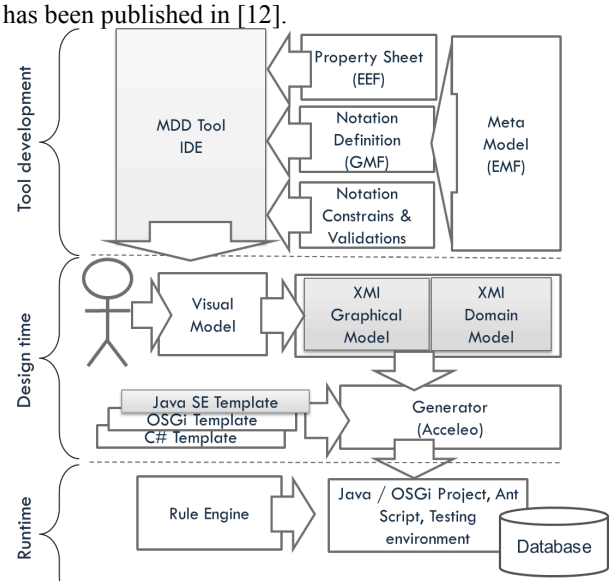

Figure 4. High-level architecture of the IoTLink

We chose to implement the IoTLink as an eclipse plugin since Eclipse already offers many features required to support the productivity of the system development that are required for extending the generated code. The components used to develop IoTLink are shown in Figure 4. The IoTLink's development extensively explored the Eclipse Modeling Project (www.eclipse.org/modeling/) which already provide frameworks for developing a customized modeling language, a model transformation, and a code generator. After a careful investigation, the following plugins were selected for developing IoTLink

- Eclipse Modeling Framework (EMF) to define the metamodel of the modeling language
- Eclipse Graphical Modeling Framework (GMF) to create a graphical editor
- Extended Editing Framework (EEF) to create a property editor for the EMF elements
- Acceleo to create a model transformation from the EMF objects into Java code.

The metamodel shown in Figure 3 is implemented using a simplified UML called EMFCore (ECore) which is a standard model required by EMF. Then, as depicted in Figure 4 the ECore model is derived by the GMF to define the Graphical definition model, called "gmf-graph". It determines the visual elements to be displayed on the main canvas, the relationships, and constraints between diagrams, as well as other behaviors. Further, GMF creates a Tooling definition model, called "gmftool", which defines the notations to be displayed on the palette menu. The gmfgraph and gmftools are then mapped in a mapping configuration, called "gmfmap" which is used by GMF to decide on what notation should be shown on the main canvas when an item from the palette menu is dragged and dropped to the main canvas. To create a more visually attractive property sheet for each diagram, we use the EEF plugin. EEF derives the Metamodel to generate an EEF model. An EEF model defines the widgets used in the property sheet of each notation. As shown in Figure 5, IoTLink's user interface maps the proposed metamodel.

B. The connection components

Currently, we have implemented several components that are necessary to enable IoT prototyping as well as to support interoperability with services within the enterprise environment. They include:

- *ArduinoSerial* enables communication with Arduino (www.arduino.cc) boards. Arduino has been widely used for rapid hardware prototyping.
- *SOAPInput* enables connection to a SOAP-based web service, which are widely used among various enterprise applications and recently has been proposed for devices (DPWS). The SOAPInput uses an XPath (www.w3.org/TR/xpath/) expression to parse the incoming soap objects.
- *RESTInput* provides a simple and lightweight

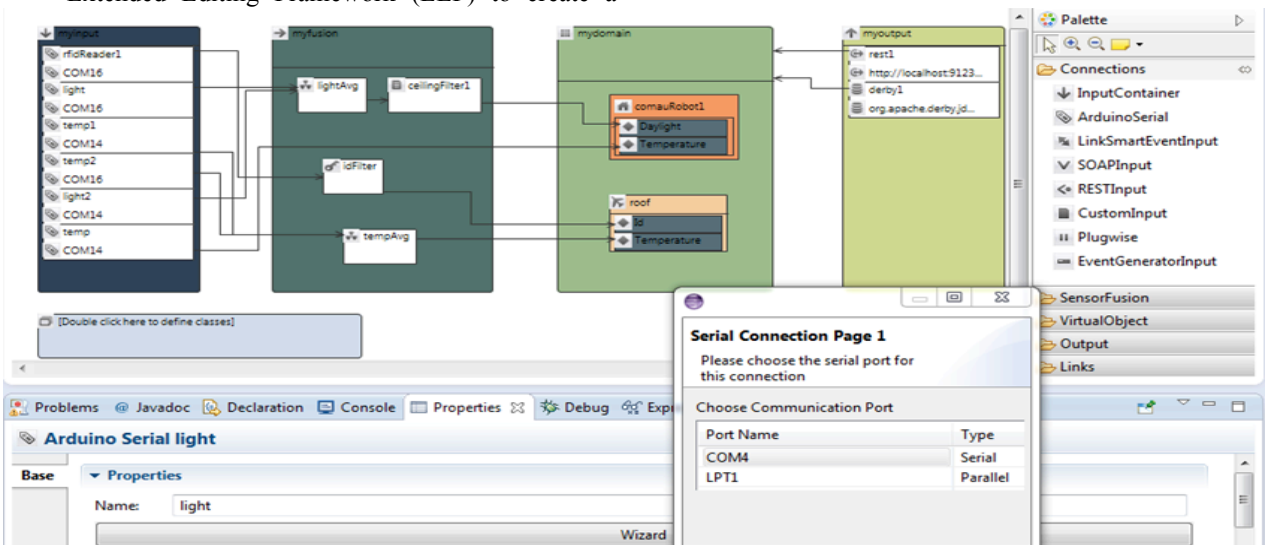

Figure 5. Latest iteration of the IoTLink

alternative to SOAP-based web service. RESTInput allows the users to poll a resource on a specific URL. It also uses XPath and JSONPath to parse the incoming XML and JSON respectively.

- *OPCClient* enables the communication to industrial devices through an OPC middleware, which is widely used in the industrial environment. The OPCClient component can be configured to poll an OPC variable by providing the tag of the variable.
- MQTTInput, this connection receives data from an MQTT broker[14]. MQTT is an emerging communication standard for IoT that adopts publish-subscribe paradigm. MQTT features a small footprint and three level of QoS, which makes it ideal to run on devices with limited resources and unreliable network with low bandwidths.

C. *Defining Complex Event Processing*

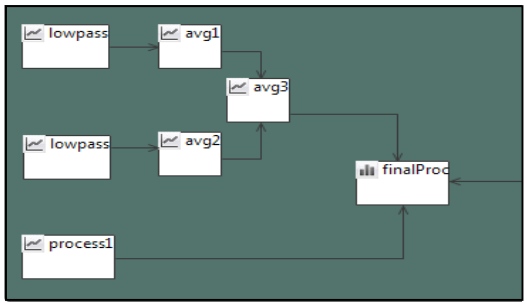

Figure 6. A network of sensor fusion modules

For processing and combining sensor data, IoTLink includes a complex event processing (CEP) engine called Esper (esper.codehaus.org). We choose Esper since it is able to process data stream efficiently. Esper allows the users to find event patterns or aggregate events using a query language called Event Processing Language (EPL). ESPER also allows aggregation and grouping using "group by" and "having" clause, which is useful to perform calculations of values based on particular group.

To enable parallel processing of event streams, IoTLink allows sensor fusion modules to be combined as a network of processes that are run in separate threads as depicted in Figure 6. This allows data to be processed through a network of modular algorithms until the desired information is obtained.

D. *Defining Virtual Object Component*

In the virtual object container, developers are able to define the representation of the physical objects belonging to two different types. First, StaticObject represents stationary relation between physical objects and the sensors and actuators that observe them e.g. a room that has a temperature sensor attached on the wall of the room can be represented by a static object. Secondly, objects that only have temporary relations to the sensors e.g. occupants who move from one room to another can be observed by the sensors located nearby.

Similar to object-oriented programming, each virtual object must have a class that defines its structure. The structure of a class includes properties and functions. Properties may have a type of primitive data types such as int, float, double, string, boolean, byte or a type of another class. The latter ones are called Complex Properties. These classes are defined in the TemplateContainer, which opens a separate diagram when the users double click on it. When the classes are defined, on the main canvas, the users could add concrete objects and assign a class to them. When a class is assigned, the structure of the class is applied to the object. This is useful for maintaining structural changes to a lot of objects.

When a sensor is used to observe a specific property of a physical object, the developer can model this using IoTLink by linking the relevant input component to the property of the virtual object. This mapping is used by the code generator to route the values of the corresponding sensor to the object being observed. When several sensors are required to determine a specific property of a physical object, it can be modelled by linking the respective input components to a sensor fusion component, which then linked to a virtual object. The objects may also contain functions that can be mapped to actuators, which are used by the generated code to forward the function calls to the relevant actuators.

E. *Output Components*

The output components define how the virtual objects should be exposed to the external applications. We have implemented several components including storing the states of the objects to a relational database, exposing the objects as SOAP, or resources through REST, publishing the objects to MQTT broker, and sending the objects to a Drools rule engine. As these components work differently, each output results in different behavior, for instance, the SOAP-based Web service provides a method to get different objects based on their classes. E.g., if there is a static object with a name of "object1" and has a type of "Class1", the output component will generate a Web Service method named getClass1(String Id). To retrieve object1, users could invoke getClass1("object1"). The REST component represents the virtual objects as web resources that can be retrieved through specific URLs. For instance, given an object with an id of "object1" and class of "Class1" and the application is run on the local host, the REST component generates the following URLs:

http://localhost/virtualobject/ class1/object1.

Moreover, the REST component generates parameterized URLs to invoke the functions of the virtual objects e.g.:

http://localhost/virtualobject/class1/object1?setOn =true.

The database component uses EclipseLink (www.eclipse.org/eclipselink/), the implementation of Java Persistence API (JPA) to interact with a database

engine. The generated classes are annotated and automatically mapped to tables by EclipseLink. When the state of the object has changed, the snapshot is stored in the history table. The MQTTOutput component provides an event publisher to publish the state of the virtual objects through an MQTTevent broker [15]. The publisher could be configured to publish events with the two topic formats. First, a flat structure topic, which only includes the class of the object, the object id, and the property as follows:

baseTopic/virtualobject/[ObjectClass]/[Obj.Id]/[PropName]

The topic structure allows developers to subscribe all events based on the class of the virtual objects using wildcard topic. The second format follows a hierarchical structure containing the objects id as shown by the following example:

baseTopic/virtualobject/[Obj.Id1]/[Obj.Id2]/../[PropName],
where the subsequent object is a child
object of the prior object.

The second topic pattern allows the application to subscribe to all events belong to an object and its children. IoTLink is also able to generate a connection to a Drools[16] rule engine. This enables developers to define rules to act based on the state of the virtual objects. The Drools component can be configured to poll the rules from a central repository called Guvnor[16] Database. This allows developers to deploy and change rules at runtime, which saves the re-deployment time.

F. Generated Application

IoTLink generates Java artifacts based on the platform-independent model. For each data source, sensor fusion and output component a Java class is generated. These classes are used by the controller class named MainApp, which initializes the concrete objects. The MainApp holds the link between domain objects, data sources, sensor fusion modules, and output. When data source objects receive data from physical objects, they are pushed to the sensor fusion modules, to which they are connected. The data could go through several levels of fusion depending on how the sensor fusion components are modeled. Once the sensor data is processed, it is pushed to the MainApp. If the sensor data does not need to be processed through sensor fusion modules, the data is pushed directly to the MainApp. Since the MainApp maintains the link between modules, it is able to assign these data to the corresponding virtual objects.

When the virtual objects are updated, the output components are notified so that they can push the data if necessary e.g. the Database could persist the changes, MQTT broker could notify the subscribers, and the Drools could update the objects in its knowledge base. This is however not required by the output components that must be pulled e.g. SOAP- and RESTOutput.

IV. EVALUATION

IoTLink was evaluated through two methods. First, to measure the usefulness of IoTLink in a real-world development, a case study is used. However, the result of the case study is hard to be generalized [17]. Therefore, a formal study measuring the usability of IoTLink was also conducted.

A. Case study

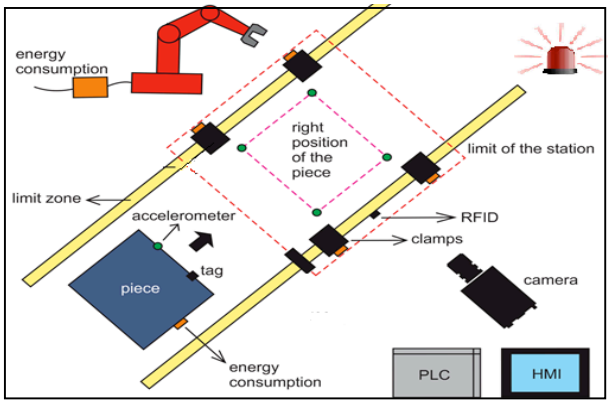

Figure 7. Manufacturing test bed set up at COMAU's site

In the case study, a model-driven approach and IoTLink was applied for integrating the data from manufacturing stations into a monitoring application that runs on an iPad. The main challenge in this case study is integrating different technologies including industrial automation devices that can be accessed through OPC protocol [18], wireless sensor network using 6LowPAN, and iPad which supports a Wi-Fi network.

The showcase was part of an EU-Brazil research project to demonstrate energy efficient manufacturing. Thus, the iPad App was essential to monitor the amount of energy that the station requires welding a rooftop of a sedan. As Figure 7 illustrates, the station contains a robot with a welding gun, a conveyor system. Each device is equipped with a power sensor that are accessible through an OPC server. In addition to the real sensors, event generators are used for simulating four further stations and four robots per stations. To monitor the energy consumptions in each station and robot, the application must retrieve data from each sensor, and sum all the energy values per robot and then per station. The aggregated values must be stored in a database and shown on an iPad App to the line manager.

For creating the prototype of the aforementioned application, the class diagram was created using IoTLink as depicted in Figure 8A. We modelled classes for Manufacturing Line, Station, Robot, and Device, which have a "Power" property for containing the energy consumptions of the devices attached to them. After the classes are modeled, the virtual objects must be instantiated in the main canvas and linked with necessary the input, output, and sensor fusion components. Fortunately, IoTLink is able to generate the concrete objects based on the cardinality defined in the class

diagram. In the input compartment, OPC Inputs for subscribing retrieving the sensor data were used. In the sensor fusion compartment, several fusion components are created first to aggregate the axles power consumption into the robot power consumption, secondly to aggregate the robots' consumptions into the overall station's consumptions, and finally from stations' consumption into the overall line's consumptions. The model uses Esper's complex event processing engine, which can be configured with a domain specific language (EPL) to accumulate the consumption events within a time interval.

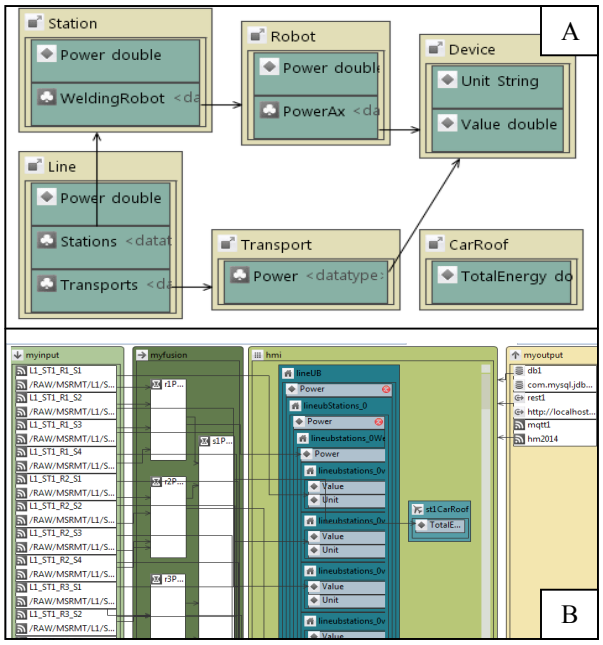

Figure 8. Classes used to represent the entities in the domain (A) and the concrete implementation model where concrete instances are linked to data sources (B)

In addition, another Virtual Object in each station is required to represent the car roof being processed. Each of this car roof has a "TotalEnergy" property, which is linked to all energy sensors in the station when it enters the station (Figure 8B). This allows the roof to accumulate the energy data from one station to another station providing an overview how much energy is required to produce the roof of a car. In the output area, three output components including the DatabaseOutput, RestOutput and MQTTEventOutput are instantiated. The DatabaseOutput generates the necessary Java Persistence API (JPA) annotations, which are used by the EclipseLink (www.eclipse.org/eclipselink/) to generate the database schema and map the objects into the entries in the database tables.

Applying IoTLink to the case study was able to solve the following challenges to the development:

- It solves the interoperability issues between different components since it supports different communication technologies for communicating with physical devices as well as for exposing the virtual objects.
- It keeps the code consistent based on a more abstract model, which is easier to maintain.
- It was able to accelerate the development time by generating the required source code to perform monitoring and reduce mistakes that were usually caused by copy-pasting chunk of codes.
- It was able to facilitate communication between stakeholders with different background, i.e., the abstract model was able to be understood by the electrical engineers and the project manager easily.

B. Empirical study

The empirical study is designed to identify IoTLink's efficiency and effectiveness as well as the users' satisfaction when using it in the IoT software development. These are three factors that resemble a usability metric as described by ISO 9241-11[19]. We compared the time required for developing a program that monitor the temperature and light intensities in two rooms using IoTLink and Java libraries. The study was done using within-group design, which requires the participants to perform the same tasks twice, using IoTLink and Java libraries. The Java library was designed to have a similar abstraction level to the IoTLink components. The order of the tool was alternated for every different participant to cancel out the learning and fatigue effects. The monitoring program was decomposed into five smaller tasks. They include (1) defining domain model including the class and objects. (2) Subscribing to four MQTT events and update the domain objects based on the values. (3) Perform an average of the light intensity values before they are assigned to a property of the rooms. (4) Publish the objects through REST-based service. (5) Publish the objects as MQTT events when the objects are updated. After the five tasks were done with IoTLink or Java libraries, the participants are asked to fill a Post-Study System Usability Questionnaire (PSSUQ) [20]. It comprise four 19 questions to evaluate four aspects of the tool including the overall satisfaction score (OVERALL), the system usefulness (SYSUSE), the information quality (INFOQUAL), and the interface quality (INTERQUAL). The efficiency was measured by the time required by the participants to perform a task, and the effectiveness was measured by the errors done by the participants. The evaluation was performed on a Dell latitude E6230 with core i7, 16GB Memory, 256 SSD. The 12 male participants were randomly chosen and have object oriented experiences between 2-17 years with median of 7.5, UML experiences between 1-12 with a median of 5 years, and IoT experience between 0-6 with a median of 1.5 years.

Performing an analysis on the time required to complete the tasks of using paired T-Test shows that the total time to solve the tasks using IoTLink (M=29; SD=15) was 42% faster than Java (M=49; SD=20) ($T(11)=3.3$, $p<.05$). Using IoTLink (M=7.8; SD=3.5) to

link MQTT events to virtual object (task2) was 52% faster than using Java library (M=16.8; SD=11.7) [T(11)=2.4, p<.05.]. IoTLink (M=4.6; SD=4.5) was 59.5% faster to perform task 4 compared to the Java Library (M=10.5; SD=9) [T(11)=3.4, p<.05].

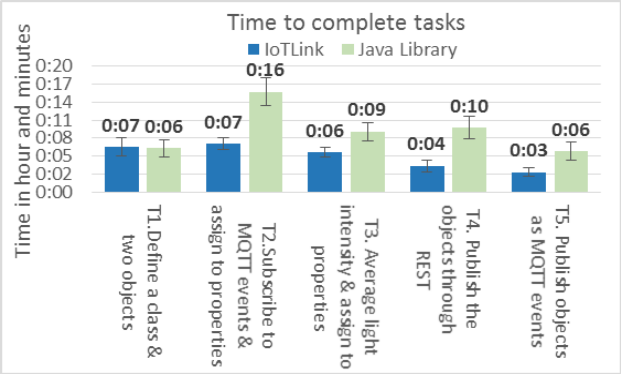

Figure 9. Time to complete each task.

IoTLink was 34.4% and 51.3% faster than Java library for performing tasks 3 and 4 respectively. However, paired T-Test analyses show no significant differences. IoTLink was 2.7% slower than Java for defining classes and objects, but a T-Test analysis shows no significant differences.

We could see a pattern from these tasks where IoTLink is faster for linking components than using the Java library. For instance, task 2 requires the users to select the components and drop them in the input container. Then they had to link the input components to the domain objects, which only requires the user to draw lines from the input components to the property of each object. In opposite, using Java, the users must instantiate the objects of the connection component, create a listener, link the listener to each input object, then they must set the properties of the domain objects based on the data received by the listeners.

Using IoTLink to perform task 4 was also significantly faster since the users were only required to select the components drag them to the corresponding containers, and draw lines from the output components to the domain model container. In opposite, using Java to expose the objects through REST requires the users to annotate the Java Beans and create a service class providing methods to be called when the REST service is accessed. The process required by IoTLink can be simplified much further since it is able to generate the necessary service classes required by the REST library. In task 4, the IoTLink was also able to generate the necessary code for publishing events to an MQTT broker much further than what a library could provide. Therefore, although Java library and the IoTLink component may provide the same abstractions, code generation bring further advantages that could simplify the development.

In task 3 and 5, although the average time of IoTLink shows an improvement over Java, however it does not show significant differences. This was caused by the more experienced developers were able to reuse their java code from the previous tasks. They copied some code and modified them. Although some users were able to finish these tasks nearly as fast as using IoTLink, they also made more mistakes when using copy and paste since they were not able to change the code consistently.

There was no significant time different between IoTLink and Java for solving task 1. Since IoTLink did not optimize how objects and classes should be defined. It seems that some participants had difficulties on clicking the field to define the names, therefore the time required by IoTLink was slightly higher than using Java where some users used the eclipse wizard to generate the class, and some more experienced users were able to write code quicker than interacting with visual user interface.

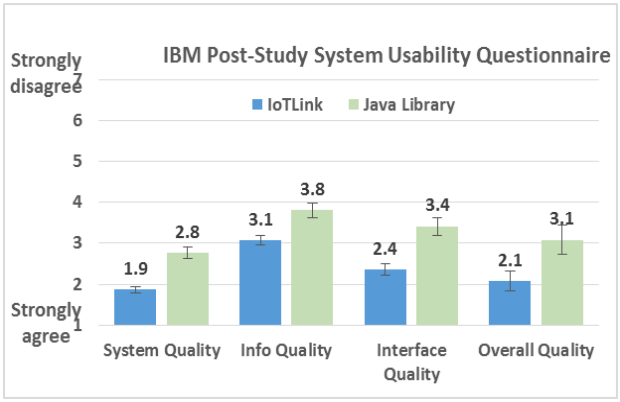

Figure 10. Participants' satisfaction to IoTLink compared to Java in a rapid prototyping.

The results of the post-study questionnaires for both IoTLink and Java are categorized according to the type of the questions as explained by Lewis et al. [21]. The analysis using paired T-Test while assuming unequal variances shows that there are significant differences between IoTLink and Java on the system quality, information quality, interface quality, and overall quality from the participants' perceptions. As illustrated in Figure 10, the users' perception on the quality of the IoTLink is superior in all categories. However, many participants pointed out that the documentation should be improved since some of they were not too familiar with IoT terms. The documentation could also be improved by providing a quick-start and an example of creating a prototype from scratch until finish. Two participants complained because the documentation did not explain in detail the logic of the components, which made them difficult to understand what happened behind the scenes.

V. CONCLUSION

Applying IoTLink to build a prototype of a European-Brazil research project has given us an insight that the tool is definitely able to support a rapid prototyping development. When different components must be tested, developers could rapidly compose the components visually and generate a Java code based on the model. This feature is able to save a lot of time

compared to using conventional programming language where developers are required to learn the documentation extensively.

A controlled experiment comparing IoTLink with Java development revealed that for almost every task IoTLink requires a less time than the conventional Java library since it was able to encapsulate the technical details and automate some implementation tasks. In addition, visual cues seem to add more confidence to the developers when dealing with unfamiliar components. IoTLink only presents the necessary options for the developers, which makes it faster for the developers to decide the necessary actions required to complete the tasks. In contrast, Java programming provides extensive possibilities, which could overwhelm the inexperienced developers. The participants also claimed that a visual representation of the data flow could provide them with a better overview of their solution. However, we have not investigated how far this would affect the developers' comprehension of the solution. Therefore, a formal study should be done to measure this. Another advantage of using MDD approach is that it is able to automate some programming task and generate the necessary code, which must be typed manually when using Java programming.

In the case study, we learned that IoTLink could be improved by utilizing some kind of iterations for handling a large number of identical objects. Moreover, the diagram should be partitioned to increase the understandability as well as the performance.

As future work, we plan to evaluate the tool with a different group of users such as experienced versus inexperienced developers in IoT. In addition, we will evaluate other IoTLink features such as the connection to the Drools rule engine. We also intend to add features that allow developers to simulate their solutions in order to check the completeness and correctness of the model before they generate the Java code.

VI. ACKNOWLEDGMENT

This work was co-funded by the European Commission and the CNPq through joint research project IMPReSS (FP7-ICT-2013-EU-Brazil, GA No. 614100) and EBBITS (FP7-ICT-2009.1.3, GA No. 257852).

VII. REFERENCES

[1] http://www.rfidjournal.com/article/print/4986, accessed June 20, 2014

[2] Atzori, L., Iera, A., and Morabito, G.: 'The internet of things: A survey', Computer Networks, 2010, 54, (15), pp. 2787-2805

[3] Vermesan, O., and Friess, P.: 'Internet of things: converging technologies for smart environments and integrated ecosystems' (River Publishers, 2013. 2013)

[4] Bandyopadhyay, S., Sengupta, M., Maiti, S., and Dutta, S.: 'A survey of middleware for internet of things': 'Recent Trends in Wireless and Mobile Networks' (Springer, 2011), pp. 288-296

[5] Bandyopadhyay, S., Sengupta, M., Maiti, S., and Dutta, S.: 'Role of middleware for internet of things: A study', International Journal of Computer Science and Engineering Survey, 2011, 2, (3), pp. 94-105

[6] Grammel, L., and Storey, M.-A.: 'An end user perspective on mashup makers', University of Victoria Technical Report DCS-324-IR, 2008

[7] Le-Phuoc, D., Polleres, A., Tummarello, G., and Morbidoni, C.: 'DERI pipes: visual tool for wiring web data sources', in Editor (Ed.)^(Eds.): 'Book DERI pipes: visual tool for wiring web data sources' (2008, edn.), pp.

[8] Morrison, J.P.: 'Flow-Based Programming: A new approach to application development' (CreateSpace, 2010. 2010)

[9] Bauer, M., Bui, N., De Loof, J., Magerkurth, C., Nettsträter, A., Stefa, J., and Walewski, J.W.: 'IoT Reference Model': 'Enabling Things to Talk' (Springer, 2013), pp. 113-162

[10] IoT-A: 'Deliverable D1.3 – Updated reference model for IoT ', in Editor (Ed.)^(Eds.): 'Book Deliverable D1.3 – Updated reference model for IoT ' (2012, v1.5 edn.), pp.

[11] Haag, A., Goronzy, S., Schaich, P., and Williams, J.: 'Emotion recognition using bio-sensors: First steps towards an automatic system': 'Affective dialogue systems' (Springer, 2004), pp. 36-48

[12] Pramudianto, F., Indra, I.R., and Jarke, M.: 'Model Driven Development for Internet of Things Application Prototyping', in Editor (Ed.)^(Eds.): 'Book Model Driven Development for Internet of Things Application Prototyping' (Knowledge Systems Institute Graduate School, 2013, edn.), pp.

[13] Spencer, R.: 'The streamlined cognitive walkthrough method, working around social constraints encountered in a software development company', in Editor (Ed.)^(Eds.): 'Book The streamlined cognitive walkthrough method, working around social constraints encountered in a software development company' (ACM, 2000, edn.), pp. 353-359

[14] Locke, D.: 'MQ Telemetry Transport (MQTT) V3. 1 Protocol Specification', IBM developerWorks Technical Library], available at http://www. ibm. com/developerworks/webservices/library/ws-mqtt/index. html, 2010

[15] Hunkeler, U., Truong, H.L., and Stanford-Clark, A.: 'MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks', in Editor (Ed.)^(Eds.): 'Book MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks' (IEEE, 2008, edn.), pp. 791-798

[16] Bali, M.: 'Drools JBoss Rules 5.0 Developer's Guide' (Packt Publishing Ltd, 2009. 2009)

[17] Kitchenham, B., Pickard, L., and Pfleeger, S.L.: 'Case studies for method and tool evaluation', IEEE software, 1995, 12, (4), pp. 52-62

[18] Zheng, L., and Nakagawa, H.: 'OPC (OLE for process control) specification and its developments', in Editor (Ed.)^(Eds.): 'Book OPC (OLE for process control) specification and its developments' (IEEE, 2002, edn.), pp. 917-920

[19] ISO: 'ISO 9241-11: Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs): Part 11: Guidance on Usability', in Editor (Ed.)^(Eds.): 'Book ISO 9241-11: Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs): Part 11: Guidance on Usability' (1998, edn.), pp.

[20] Lewis, J.R.: 'Psychometric evaluation of the post-study system usability questionnaire: The PSSUQ', in Editor (Ed.)^(Eds.): 'Book Psychometric evaluation of the post-study system usability questionnaire: The PSSUQ' (SAGE Publications, 1992, edn.), pp. 1259-1260

[21] Lewis, J.R.: ' IBM computer usability satisfaction questionnaires: psychometric evaluation and instructions for use ' , International Journal of Human‐Computer Interaction, 1995, 7, (1), pp. 57-78