Open Research Online

The Open University's repository of research publications and other research outputs

Visual Simple Transformations: Empowering End-Users to Wire Internet of Things Objects

Journal Item

How to cite:

Akiki, Pierre A.; Bandara, Arosha K. and Yu, Yijun (2017). Visual Simple Transformations: Empowering End-Users to Wire Internet of Things Objects. Transactions on Computer-Human Interaction, 24(2) pp. 1–40.

For guidance on citations see [FAQs](#).

© [not recorded]

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Version: Accepted Manuscript

Link(s) to article on publisher's website:

<http://dx.doi.org/doi:10.1145/3057857>

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data [policy](#) on reuse of materials please consult the policies page.

oro.open.ac.uk

Visual Simple Transformations: Empowering End-Users to Wire Internet of Things Objects

PIERRE A. AKIKI, Notre Dame University – Louaize
 AROSHA K. BANDARA, The Open University
 YIJUN YU, The Open University

Empowering end-users to wire Internet of Things (IoT) objects (things and services) together would allow them to more easily conceive and realize interesting IoT solutions. A challenge lies in devising a simple end-user development approach to support the specification of transformations, which can bridge the mismatch in the data being exchanged among IoT objects. To tackle this challenge, we present Visual Simple Transformations (ViSiT) as an approach that allows end-users to use a jigsaw puzzle metaphor for specifying transformations that are automatically converted into underlying executable workflows. ViSiT is explained by presenting meta-models and an architecture for implementing a system of connected IoT objects. A tool is provided for supporting end-users in visually developing and testing transformations. Another tool is also provided for allowing software developers to modify, if they wish, a transformation's underlying implementation. This work was evaluated from a technical perspective by developing transformations and measuring ViSiT's efficiency and scalability and by constructing an example application to show ViSiT's practicality. A study was conducted to evaluate this work from an end-user perspective, and its results showed positive indications of perceived usability, learnability, and the ability to conceive real-life scenarios for ViSiT.

CCS Concepts: • **Software and its engineering** → **Visual languages; Integrated and visual development environments; Software architectures**

General Terms: Human Factors, Languages

Additional Key Words and Phrases: End-User Development, Internet of Things, Transformations

ACM Reference Format:

Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu, 2017. Visual Simple Transformations: Empowering End-Users to Wire Internet of Things Objects
ACM Transactions on Computer-Human Interaction. 24, 2, Article 10 (April 2017), 44 pages.
 DOI: <http://dx.doi.org/10.1145/3057857>

1. INTRODUCTION

The Internet of Things (IoT) is a growing paradigm that brings together a wide variety of smart objects [Kortuem et al. 2010]. The IoT will have a major impact on many aspects of the everyday-life and behavior of its potential users, in both the work and domestic environments [Atzori et al. 2010]. People could gain access to a large number and a wide variety of IoT objects (things and services) that are provided by different companies. Empowering end-users with the ability to wire

This work is supported by ERC Advanced Grant 291652.

Authors' addresses: P. A. Akiki, Department of Computer Science, Notre Dame University - Louaize, Zouk Mosbeh, Lebanon; email: pakiki@ndu.edu.lb; A. K. Bandara and Y. Yu, Computing and Communications Department, The Open University, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom; emails: {arosha.bandara, yijun.yu}@open.ac.uk.

Permission to make digital or hardcopies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credits permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1073-0516/2017/04-ART10 \$15.00

DOI: <http://dx.doi.org/10.1145/3057857>

these objects together could trigger their imagination and allow them to realize many interesting scenarios. As the number of IoT objects increases, the possible configuration combinations will also increase. Since this configuration is going to be performed by end-users, it becomes a challenge that can be classified as end-user development [Jeffrey C. F. Ho 2015]. The end-user programming paradigm is already a common form of programming, e.g., in spreadsheet applications [Burnett et al. 2004], and is particularly encouraged in an IoT setting [Burnett and Kulesza 2015].

Linking IoT objects (things and services), which could originate from different companies, requires transforming the communication data from one representation (source) to another (target). In this situation, end-users can benefit from a simple approach that allows them to link IoT objects and define the necessary transformations. In this paper, we draw on knowledge from the disciplines of software engineering and HCI to present ViSiT (Visual Simple Transformations). ViSiT allows end-users to define transformations for realizing such data conversions between communicating IoT objects. As recommended by the meta-design approach [Fischer et al. 2004], end-users are not presented with a closed IoT system, but are provided with the necessary tools that allow them to extend the system in a way that would fit their needs.

1.1 Why is ViSiT Useful?

ViSiT allows end-users to define transformations visually by using a jigsaw puzzle metaphor (based on [Danado and Paternò 2014] and [Humble et al. 2003]). This metaphor could be more usable for non-programmers than existing model transformation languages such as ATL [Jouault et al. 2006] and Henshin [Arendt et al. 2010], which usually target technical experts.

With ViSiT, end-users do not need to write code or use technical visual notations, e.g., Unified Modelling Language (UML), in order to define transformations that allow IoT objects to exchange data. In his keynote¹ at ICSE 2015, Grady Booch mentioned that in case someone was thinking of creating a new programming language, the target user-group should be non-programmers. Visual tools can in some cases reduce the learning curve for end-users with basic or no programming skills by helping them to achieve tasks that would otherwise require an advanced knowledge of coding. For example, the Yahoo Pipes² system was presented as a visual tool for transforming web information by grabbing data such as images and RSS feeds from a URL input and transforming it. In this sense, providing non-programmers with a simple tool-supported approach for creating transformations could empower them in configuring IoT objects to communicate. End-users who wish to define transformations in ViSiT are supported by a web-based visual-design tool. This tool communicates with a web-service that generates an underlying executable implementation for the jigsaw puzzle transformation. Hence, much of the complexity is hidden from the end-users.

¹ Grady Booch, **The Future of Software Engineering** at ICSE 2015:
<https://www.youtube.com/watch?v=h1TGJJ-F-fE>

² **Yahoo Pipes**: <https://pipes.yahoo.com>

Fig. 1. Basic Scenario of an IoT Refrigerator Placing an Order for its Low-Stock Items at an Online Shopping Service

1.2 An Example of Applying ViSiT in an IoT Setting

A basic example is presented in Fig. 1 to demonstrate how ViSiT can be applied in an IoT setting. This example presents a hypothetical scenario, where notifications are needed between a smart refrigerator and an online shopping service. In this scenario, the refrigerator has computational intelligence and sensors, which allow it to detect when its stock is low on certain kinds of items. Assume that the trays holding the butter and the eggs have weight-based sensors, which can detect if the quantity of an item is running low. Once low-stock items are detected, the refrigerator can place an

order at an online shopping service to replenish its stock. Hence, the owner would not have to worry about running out of butter or eggs.

The problem lies in the mismatch between the format of the data generated by the refrigerator and the one expected by the shopping service. Hence, for compatibility reasons, the refrigerator cannot transmit the data as it is to the shopping service. A transformation is needed for converting the low-stock items being sent from the refrigerator (Fig. 1 – a), to a purchase order format compliant with the shopping service (Fig. 1 – b). It is not feasible for every IoT device vendor and every service provider to create components that allow their products to communicate with all other available products. Hence, a mediation system is required for defining such communication links and transformations. There are existing general automatic mediation approaches for software components [Bennaceur and Issarny 2015]. Yet, in an IoT setting, involving end-users empowers them to realize interesting solutions of their own conception. Hence, ViSiT adds the necessary concepts that make the jigsaw puzzle metaphor applicable to the definition of transformations.

We can see in Fig. 1 – c how an end-user can wire together the refrigerator and the shopping service visually by putting three jigsaw puzzle pieces together. Then, the end-user can also add a transformation using the same visual metaphor as shown in Fig. 1 – d. This transformation is automatically converted to an underlying executable workflow implementation like the one shown in Fig. 1 – e. When executed, the transformation shown in Fig. 1 – e converts the low-stock items (Fig. 1 – a) to the purchase order items (Fig. 1 – b). End-users are not required to do any coding, but merely use visual elements to indicate how the source model should be transformed to the target model. In this example, the transformation’s visual elements shown in Fig. 1 – d include: the target collection (PurchaseOrder) and the target properties (ItemRef and Qty). The “ItemRef” property is simply assigned its “Code” counterpart from the source model. As a hypothetical scenario, assume that the end-users would like to order double the low-stock quantity. They can just multiply the low-stock-items’ “Quantity” property by two before assigning it to the “Qty” property of the purchase order. If the end-users would like to connect the refrigerator to another shopping service or vice-versa, they just need to specify a new ViSiT link and transformation to fit that purpose.

We applied ViSiT to different examples in order to demonstrate its practicality and ability to address real-life scenarios. These examples are presented in this paper and provide more details on ViSiT’s capabilities and supported features.

1.3 Structure of the Article

This work is related to both end-user development and model transformations. Hence, the strengths and shortcomings of the state-of-the-art in both areas are discussed in Section 2. ViSiT’s concepts are presented as meta-models and explained in Section 3. This section also presents and explains our proposed architecture for enabling IoT objects to communicate using ViSiT. Section 3 also discusses the reasons for choosing ViSiT’s visual jigsaw puzzle metaphor and its underlying workflow implementation. The method of creating and executing workflows that realize a transformation is explained in Section 4. The tools that support end-users and developers in defining, testing, and managing transformations are presented in Section 5. Additional examples are given in Section 6 to demonstrate the kinds of transformations that can be accomplished using ViSiT. These examples demonstrate ViSiT’s features further. ViSiT is evaluated from a technical perspective in Section 7. The efficiency and

scalability of ViSiT are demonstrated by measuring its execution times when it is applied to models with a varying size and complexity. A practical application is also presented in Section 7 to show the viability of using ViSiT for real-life applications. A usability study that we conducted to evaluate ViSiT is presented in Section 8. The threats to validity and limitations of this work are discussed in Section 9. Finally, our conclusions are given in Section 10.

2. RELATED WORK

This section highlights the strengths and shortcomings of existing approaches related to end-user development and model-transformations. These approaches are assessed from an HCI perspective, with respect to how end-users who are non-programmers might find them easy to learn and use. Our aim is to elicit the best qualities from both areas in order to devise a transformation definition approach, which would be suitable for end-users within an IoT setting. In this section, we classify the related work into two main categories: end-user development approaches and transformation approaches. The first category includes the approaches that directly target end-user development or have the potential of being used by end-users. On the other hand, the second category covers model transformation languages that usually target software developers. Each of these two categories are also divided into subcategories, which provide a more detailed classification.

2.1 End-User Development Approaches

Several approaches such as programming by example [Cypher and Halbert 1993], [Lieberman 2001] and the use of component-based technologies [Mørch et al. 2004], were proposed for supporting end-users in developing various types of applications. Many of these approaches focus on providing a visual paradigm, which helps end-users in building applications without requiring advanced technical knowledge. For example, the Cicero Designer allows end-users to customize the user interface (UI) of a museum guide by performing direct manipulation [Ghiani et al. 2009]. Some approaches are dedicated for developing IoT applications. However, these approaches do not target supporting end-users in defining transformations that allow IoT objects to communicate by exchanging data. This section provides a brief overview of the state-of-the-art, but more details can be found in the existing literature [Lieberman et al. 2006], [Ko et al. 2011].

2.1.1 End-User Development Approaches for Educational Purposes. **Alice** is an approach for making introductory programming courses easier to grasp [Dann et al. 2011]. **Scratch** was created for helping end-users, primarily children, in learning computer programming by creating real working applications [Maloney et al. 2010]. **TouchDevelop** maps the constructs offered by code-based programming languages to visual alternatives, which allow end-users to define applications by using their smartphones [Athreya et al. 2012].

These approaches are important innovations that teach non-programmers how to program, instead of restricting this activity to professional programmers who know how to use traditional programming languages.

2.1.2 Spreadsheet-Based End-User Development Approaches. Service composition is the target application of many end-user development approaches [Hang and Zhao 2015]. For example, **DashMash** promotes the use of web mashups as a technique for

end-user development [Cappiello et al. 2011]. A number of approaches attempt to support end-user service composition using spreadsheet-based UIs. **DataSheets** is a spreadsheet-based data-flow language that supports end-users with no expertise with complex transformational languages in performing data integration operations [Lagares Lemos et al. 2013]. DataSheets focuses on supporting the mapping of different web-services in a service composition environment. **AMICO:CALC** is a framework for supporting end-user spreadsheet-based service composition [Obrenović and Gašević 2008]. The authors of this framework identify and implement a set of requirements that enable spreadsheets to communicate with various services. **Mashroom** is an end-user mashup programming environment, which uses nested tables with visual mashup operators to offer a spreadsheet-like programming experience [Wang et al. 2009]. Kongdenfha et al. [2009] also present a tool for developing web mashups using a spreadsheet-like environment. This tool was implemented as a prototype that includes an Excel add-in and a backend server to execute mashups. **MashSheet** is another mashup tool that was implemented as an Excel plug-in [Hoang et al. 2010]. It uses an XML-based data model that extends the conventional spreadsheet data model to include complex data types. **Vegemite** targets end-user programming of mashups [Lin et al. 2009], by extending the CoScripter (Koala) web automation tool [Little et al. 2007]. It provides a spreadsheet-like editor in a web-browser environment.

The use of spreadsheets-like UIs in these approaches is definitely an advantage, since spreadsheets are familiar to many end-users and can support advanced features like formulas. Nonetheless, the jigsaw puzzle notation that we adopted in ViSiT could be familiar with a broader part of the end-user population and can be more suited for use on touch-screen devices.

2.1.3 IoT-Focused General Visual Programming Approaches. Several of the existing IoT-focused visual programming tools could potentially be used by non-programmers. The main aim of these approaches is providing a user-friendly way for programming IoT devices. **Node-RED**³ offers predefined blocks (nodes), which can be wired together using a browser-based environment in order to link devices and services. **NETLab Toolkit**⁴ (NTK) aims at supporting developers and other stakeholders, e.g., researchers and students, who would like to develop IoT applications. NTK offers a web-based authoring environment that can be used to wire box-shaped prebuilt components and configure them using various widgets such as: input fields, sliders, etc. There are approaches that target the development of applications for particular hardware platforms. For example, **Scratch for Android**⁵, **Ardublock**⁶, **Modkit**⁷, and **Sense** [Kortuem et al. 2013] offer desktop authoring tools that support the programming of the Arduino platform.

These approaches share two strong qualities: support for visual programming and web-based authoring tools. These qualities are valuable for end-users since visual programming could offer much needed ease-of-use, and a web-based authoring tool can be accessed from a variety of devices. However, these approaches do not offer a

³ IBM Emerging Technologies, **Node-RED**: <http://www.nodered.org>

⁴ Philip van Allen, **NETLab Toolkit**: <http://www.netlabtoolkit.org>

⁵ Citilab, **Scratch for Android** (S4A): <http://s4a.cat>

⁶ He Qichen and David Li, **Ardublock**: <http://sourceforge.net/projects/ardublock>

⁷ Modkit LLC, **Modkit**: <http://www.modkit.com>

way for end-users to easily define transformations that link different devices and services. Some approaches support the ability to program new components, which could potentially be used to perform such a link. However, doing so requires coding, e.g., JavaScript in NTK, which might not be easy for non-programmers.

2.1.4 IoT-Focused End-User Development Approaches. Several approaches target end-user development particularly in an IoT context. **Tasker**⁸ gives end-users more control over Android devices by supporting the execution of tasks based on contexts (e.g., date, time, events, etc.). Tasker provides a visual list UI for end-users to define tasks and contexts. **MakerSwarm**⁹ provides end-users with a mobile application that allows them to connect different types of IoT devices. This product has been described as a roll of duct tape for the IoT as an analogy to how it connects a variety of devices together like duct tape glues everyday objects. With MakerSwarm, end-users can wire predefined components together to realize a variety of scenarios. **Ambient Dynamix** [Carlson et al. 2015] is another approach that is similar to MakerSwarm, in the sense that it acts as a middleware framework that can connect incompatible IoT devices by using plug-ins that can be dynamically installed on the user's Android-based device. **TeC** is a framework that supports end-user development of applications for smart spaces [Sousa et al. 2011]. It provides an editor that allows end-users to associate actions that occur on certain devices with events on others. For example, when there is a fence-monitor alert make a phone call. MakerSwarm Ambient Dynamics and TeC offer a visual canvas onto which configurable nodes can be placed and connected using wires. **IFTTT**¹⁰ (If This Then That) is a service that supports the definition of connections between products and applications. IFTTT allows end-users to create chains of conditions called "Recipes", which can be of two kinds "DO" and "IF". The first type (DO) executes with the tap of a button and can perform actions like uploading photos to Facebook. The second type (IF) runs in the background and connects applications with "if this then that" statements. An example "IF" recipe is: "If I post a picture on Instagram, save it to Dropbox". IFTTT provides a web UI for browsing and choosing the "this" and "that" parts.

These approaches demonstrate the importance of allowing end-users to connect IoT objects, without having to be totally dependent on IT experts. This point is significantly related to the general idea behind ViSiT. Despite their importance, these approaches are limited by the dependence on components that are preprogrammed by software developers to serve as a communication link between particular IoT devices and services. As the IoT grows, a wide variety of devices will be available from different vendors. Hence, it will be difficult to preprogram components that satisfy all possible combinations. Additionally, they do not focus on supporting transformations between source and target data. For example, although TeC supports the definition of data streams between devices, unlike ViSiT, it does not support transforming the source data to the expected format on the target. ViSiT aims at allowing end-users to conceive a wide variety of scenarios where IoT objects can work together, link these objects, and define transformations that allow them to communicate. This approach should significantly reduce the dependency on technical

⁸ **Tasker**: <http://tasker.dinglisich.net>

⁹ MAYA Design, **MakerSwarm**: <http://www.makerswarm.com>

¹⁰ **IFTTT**: <https://ifttt.com/wtf>

experts. Hence, end-users will not be compelled to wait for software companies to release new device-linking components. With ViSiT IT experts only have to specify a description of what an IoT object can do, e.g., actions and events. This description serves as an Application Programming Interface (API) that end-users can control by visually defining links and transformations. Nonetheless, it is also possible to use preprogrammed components with ViSiT.

2.1.5 IoT-Focused End-User Development Approaches for the Home. Several works focus on end-user development in a home environment. **AppsGate** supports the composition of condition statements that are similar to those of IFTTT while making the distinction between states and events as triggers, as well as between instantaneous, extended, and sustained actions [Coutaz and Crowley 2016]. The authors assessed AppsGate by deploying it in their own home in addition to conducting an experiment in the homes of five families. **OSCAR** is an application that allows end-users to monitor, connect, and configure devices in home media networks [Newman et al. 2008]. In terms of connecting devices, OSCAR's interface allows end-users to connect a node to another one that is compatible with the provided input. For example, the output of a security camera can be displayed on a nearby screen. **Pantagruel** is a high-level visual language for programming home automation applications [Drey and Consel 2012]. This language allows end-users to place conditions on sensors and trigger an actuator when a condition is realized. For example, in case the weather is warm an alarm should be triggered.

These approaches provide interesting technological and empirical insights on end-user development in an IoT context. Yet, they do not support the specification and configuration of transformations that allow data to be exchanged among incompatible things and services.

Puzzle supports end-user development on mobile devices [Danado and Paternò 2012]. To provide end-users with an easy-to-use and familiar approach, the authors of Puzzle conducted a study that compared six different metaphors including: jigsaw, natural language, Lego, Meccano, bricks, and workflow [Danado and Paternò 2014]. End-users gave the *jigsaw* metaphor the highest ranking, with *workflows* ranking closely behind it. Puzzle focuses on supporting end-user development of applications that can control devices such as lamps. The use of the **jigsaw puzzle** metaphor to support end-user configuration of ubiquitous domestic environments was introduced in an earlier work [Humble et al. 2003]. The authors of that work used what they refer to as transformers to convert digital effects to physical ones and vice versa. They also provided an editor to support end-user composition of configuration scenarios using jigsaw puzzle pieces that represent a set of preconceived operations such as: GroceryAlarm, Reminder, SendSMS, etc.

We think that the jigsaw metaphor is quite promising for end-user development, primarily due to its familiarity and its suitability for touch-screen devices. Hence, we adopted this metaphor in ViSiT and adapted it for representing links and transformations primarily in an IoT setting.

2.2 Transformation Approaches

Model transformations are part of the backbone of model-driven software engineering. Existing model transformation languages and frameworks vary in terms of their capabilities. These approaches were not designed for end-users who are non-programmers, but target experienced software developers. This paper does not aim at

replacing these well-established approaches, but at allowing end-users to define basic transformations without requiring advanced technical knowledge. The state-of-the-art transformation approaches are assessed from this perspective.

An existing survey classifies and compares model transformation approaches [Czarnecki and Helsen 2003]. This survey uses two main classification categories: model-to-model and model-to-text (code or XML). In this section, we use a different classification that relates to whether a transformation approach is code-based or visual. Approaches falling under each of the two categories are assessed in terms of whether they could be suitable for end-users who are non-programmers.

2.2.1 Code-based Approaches. Some model transformation approaches rely on the definition of code-based rules for performing transformations.

PROGRES is an early effort for specifying graph-based transformations using an imperative code-based language [Schürr et al. 1995]. **QVT**¹¹ was created by the Object Management Group (OMG) as a standard set of languages for defining model transformations. Several transformation languages were created following the QVT standards [Gardner et al. 2003]. **ATL** is one notable example [Jouault et al. 2006], [Jouault and Kurtev 2006]. **XSLT** [Clark 1999], **XQuery** [Boag et al. 2002], and **YATL** [Cluet and Siméon 2000] are examples of query and transformation languages for XML. Some approaches target bidirectional transformations. One example is **GRoundTram** [Hidaka et al. 2011], which uses a language called **UnQL+** that is based on the graph query language UnQL [Buneman et al. 2000]. Another example is **BiFluX** [Pacheco et al. 2014], which is inspired by the functional XML update language Flux [Cheney 2008]. **SiTra** tries to simplify the definition of model transformations by using Java, which is more familiar to programmers than transformation languages [Akehurst et al. 2006], [Bordbar et al. 2007].

Many of the existing code-based model transformation approaches are quite powerful. Yet, representing even simple transformations may require a significant amount of learning time, especially when complicated syntax is involved. This could be hard for stakeholders with limited or no software development skills. Hence, the learning curve could make these languages difficult to adopt by non-programmers. In comparison, the puzzle notation adopted by ViSiT could provide end-users with an easier way for defining transformations, and could have a lower learning curve than code-based approaches like the ones described in this section.

2.2.2 Visual Approaches. Some approaches provide visual programming support to help developers in defining model transformations.

AGG is a development environment, which supports the definition of graph-based imperative model transformations [Taentzer 2004]. AGG provides a tool containing a visual editor for AGG graphs, which uses a notation similar to UML object diagrams. **GReAT** [Agrawal et al. 2006] supports the representation of transformation rules with a visual flow notation, and can be used with the Generic Modeling Environment (**GME**) [Agrawal et al. 2002]. GReAT was inspired by several previous works [Brendenfeld and Camposano 1995], [Claus et al. 1979], and [Göttler 1992]. **AToM3** uses graph grammars to achieve transformations for different situations including code generation and provides a tool for visually representing graphs [De Lara and

¹¹ OMG, **QVT** (Query /View/Transformation) 1.2: <http://www.omg.org/spec/QVT/1.2>

Vangheluwe 2002, p.3]. **FUJABA** is another tool that is dedicated for code generation [Nickel et al. 2000]. **VIATRA** is based on metamodeling and graph transformations [Csertán et al. 2002], [Bergmann et al. 2015]. It supports a UML notation using Visual and Precise Metamodeling (**VPM**) [Varró and Pataricza 2003], and the Viatra Textual Metamodeling (**VTML**) language [Balogh and Varró 2006]. **Henshin** provides a visual model transformation environment that supports the definition of rule-based Eclipse Modeling Framework (EMF) model transformations [Arendt et al. 2010]. Henshin’s visual syntax resembles UML class diagrams. MDELab’s **story diagrams** provide a notation based on UML collaboration and activity diagrams to define transformations and control flow [von Detten et al. 2012]. **eMoflon** supports the definition of transformation rules using a visual syntax that resembles UML class diagrams [Anjorin et al. 2011]. **Yahoo Pipes** and **SQL-Server Integration Services**¹² provide visual notations for expressing data transformations, specifically targeting web data transformation and SQL-server data migration respectively.

Visual design languages could provide an increase in software productivity. The visual notations offered by model transformation approaches vary, but could be in general more technically challenging than the visual notations offered of end-user programming approaches. Some notations are less technical, e.g., Yahoo Pipes, but are domain specific. The visual notations offered by model transformation approaches, are generally similar to modeling languages such as UML. For example, some notations resemble UML object diagrams, class diagrams, activity diagrams, etc. Such notations could be hard to learn by end-users, since technical (modeling) knowledge is still required. In contrast, a visual metaphor such as the jigsaw puzzle (refer to Section 2.1) could be simpler and more familiar for end-users who are non-programmers. ViSiT provides end-users with a visual metaphor through which they can use some of the functionality offered by model transformation languages. Hence, it becomes possible to define a transformation, by dragging and dropping visual constructs that are easy to understand. In a previous work, we used the Windows Workflow Foundation (WF) for adapting model-driven UIs [Akiki et al. 2013b], [Akiki et al. 2014]. The underlying workflow implementation of ViSiT is based on our previous experience with WF, but ViSiT is more general purpose and has visual constructs for specifying model transformations.

3. VISIT: CONCEPTS, ARCHITECTURE, AND IMPLEMENTATION

As its name indicates, Visual Simple Transformations (ViSiT) is an approach for expressing model transformations in a visual manner. We define simplicity in terms of learnability and usability of the approach for non-programmers. This section presents class diagrams that embody the concepts behind ViSiT and explains the architecture that ties all its components together. Here we should emphasize that end-users do not have to work with the technicalities that are presented in the class diagrams, but would just use a visual metaphor to wire IoT objects and define transformations. The class diagrams used in this paper are represented using UML. For readers who are not familiar with this modelling language, the book “UML Distilled” [Fowler 2004] may serve as a useful reference. We also provide here a brief explanation that would help in reading the class diagrams presented in this paper. Our classes are connected using three different types of relationships: association,

¹² **Microsoft, SQL Server Integration Services:** <https://msdn.microsoft.com/en-us/library/ms141026.aspx>

Fig. 2. Class Diagram Depicting the Concepts behind Wiring Things and Services Together

composition, and inheritance. An association is represented as a line, a composition is represented as a line with a black diamond at the composing edge, and an inheritance is represented as a line with a white triangle pointing towards the base class. Multiplicity values, e.g., 1, 1..*, and 0..*, are placed on relationships to indicate the degree of participation. For example, a *Thing* has one *ThingType* and a *ThingType* can be allocated to many *Things*. Hence, the multiplicity of the association connecting these two classes is 1..* on the side of the *Thing* class and 1 on the side of the *ThingType* class.

3.1 Things and Services

As we mentioned in Section 1, it is difficult to predefine all the combinations between the large number and wide variety of objects in an IoT setting. Hence, having an approach that allows end-users to wire IoT objects together can spur creativity. The class diagram presented in Fig. 2 shows part of the concepts, which allow the realization of this approach. For the sake of simplicity and clarity, we have defined our own meta-models rather than using or adapting one from the literature. Existing works such as the IOT-A project [Bassi et al. 2013] proposed architectural models

Fig. 3. Class Diagram Depicting Data Specification and Documentation for Argument and Output Data

that have some of the same concepts, but also many others that are not relevant to the challenges addressed by ViSiT.

A *Thing* in an IoT setting can be any everyday item, which has been extended with computational intelligence and the ability to transfer data over a network. For example, a *Thing* can be an electronic device, a clothing item, a piece of furniture, etc. A *Thing* has a *Type*, e.g., refrigerator, lamp, chair, shoes, etc. It also has a *Brand* indicating the company that produced it. In an IoT setting, *Things* can communicate with each other, and can also communicate with online *Services*, e.g., shopping service, weather service, etc. A *Thing* raises *Events* and performs *Actions*. An *Event* is raised once a *Thing* needs to report an occurrence of a change. For example, an IoT device (*Thing*) that monitors the humidity level of a garden's soil could raise an event called "HumidityDropped". This *Event* would report that the humidity is lower than a certain threshold, and it would provide the current humidity as output data. An *Action* denotes a kind of activity that the *Thing* can perform. For example, a robot can have *Actions* such as: move, turn, stop, etc. These *Actions* receive input data (parameters) that indicate how they are performed.

Services are traditional web-services, which have *Methods* representing the types of activities that the *Service* performs. *Services* also provide *Notifications* to a client. Examples of a *Method* and a *Notification* from an online shopping service could be "PlaceOrder" and "DiscountOnItems" respectively. A *Service* has a *Type*, e.g., shopping, weather, etc. It also has a *Provider* indicating the company that offers the service.

ObjectLinks can be defined in order to wire *Things* and *Services*. These links can connect *Events* and *ServiceNotifications* to *Actions* and *ServiceMethods*. *ObjectLinks* operate in a way that is similar to event handlers in event-driven programming. Once *Things* and *Services* raise *Events* and *ServiceNotifications* respectively, an *ObjectLink* receives output data from either an *Event* or a *ServiceNotification* and passes it through a *Call* to either an *Action* or a *ServiceMethod*. In case a *ServiceCall* is made, the data is passed as a parameter to the target *ServiceMethod*. On the other hand, if a *ThingCall* is made, the data is passed as a parameter to the desired *Action*. A *Transformation* can be used to convert the data from the source's format to the one expected by the target. It is also possible to specify a *CallCondition*, which restricts the call to specific situations. For example, a *Thing* could be reporting a temperature reading, which is being relayed to an SMS service that will report it to a recipient. An

example *CallCondition* in this case could specify that the SMS service should not be called unless the temperature is above a certain threshold. End-users can specify *ObjectLinks*, *Transformations*, and *CallConditions* using the jigsaw puzzle metaphor.

Assume that a *Call* fails due to the unavailability of a *Thing* or *Service*. End-users can decide whether or not they would like to be notified of such failures. If an end-user wishes to be notified, a *Call* (*ThingCall* or *ServiceCall*) can be associated with an *ErrorNotification*. This notification indicates whether the end-user should be notified through SMS, email, or both. The end-user receives a message stating the description of the *Thing* or *Service* that is currently unavailable. It is also possible to specify whether the end-user would like to receive a notification of success after the error is gone. If this option is specified, the end-user would receive a message the first time a successful *Call* is made after one or more errors occur due to the unavailability of an IoT object.

Consider the example that was previously presented in Fig. 1. The puzzle pieces that are placed on the sides of the *ObjectLink*, e.g., Fig. 1 – c, represent particular instances of *Things* and *Services*. For example, it is possible to have more than one refrigerator in a house. Hence, instead of labelling the puzzle piece “Refrigerator”, a more specific description would be used. The same concept applies to the shopping service in case similar services are available. End-users can simply look at a toolbox, read the descriptions of the available *Things* and *Services*, and pick the ones that they would like to use in a particular scenario. Since *Transformations* are defined for a link between a particular type and instance of a *Thing* and *Service*, it is possible to have multiple IoT objects with the same *Event* or *Method* names. For example, it is possible to have a refrigerator and cabinet both providing a “LowStock” event.

In the example presented in Fig. 1, the refrigerator is a *Thing* that raises an *Event* called “LowStock”. This event sends as output data the XML document shown in Fig. 1 – a. The shopping service is a *Service* that has a *ServiceMethod* called “PlaceOrder”. This method takes as a parameter the XML document shown in Fig. 1 – b. An *ObjectLink*, such as the one shown in Fig. 1 – c, is specified by the end-user to wire the “LowStock” event to the “PlaceOrder” method. The end-user defines a *Transformation*, as shown in Fig. 1 – d, to convert the data from the refrigerator’s format to the one expected by the shopping service.

The class diagram presented in Fig. 3 shows *Events*, *Actions*, *ServiceMethods*, and *ServiceNotifications* with attached *DataSpecification* and *Documentation* for their argument and output XML data. This information is provided by an IT expert. Having a *schema* as part of the *DataSpecification* is enough for the transformations to work. However, a *sample model* and *Documentation* provide the end-users with a lot of benefits since transformations become easier to create and test. Our system automatically generates a schema from the *sample model* (data). Additionally, end-users can test transformations on the *sample source model* and observe the extent to which the result resembles the target model.

The collection and property *descriptions* are presented to the end-users as an alternative to the technical names that make up the schema of the data being exchanged among IoT objects. For example, instead of presenting end-users with a collection called “POrder” that has a property called “ItemRef” it could be clearer to use the words “Purchase Order” and “Item Reference”. Furthermore, if *examples* are specified for properties, end-users would have a better understanding of what a property is meant to hold. As a basic example, displaying “e.g., B001” under the name of the “ItemRef” property, as shown in Fig. 1 – d, could clarify that this property is

Fig. 4. Class Diagram Depicting the Concepts behind Transformations Defined Using ViSiT

meant to hold a text-based reference. Hence, by comparing examples, the end-users would be able to match a source property to its target counterpart with more ease.

If an *explanation* is specified for collections and properties, our support tool is able to present this explanation for helping end-users in understanding the domain concepts that they are transforming.

3.2 Transformation Concepts

The class diagram shown in Fig. 4 depicts the main concepts behind transformations defined using ViSiT in order to transform a source model to a target model.

ViSiT supports two types of *Transformations*: *MToOneCollectionTransformation*, and *MToNCollectionTransformation*. The word “collection” is used here to denote a group of related data elements. For example, the low-stock items shown in Fig. 1 – a can be considered as a collection. The first type transforms data from multiple (M) collections (groups of data) to one collection, while the second type transforms data from multiple collections (M) to multiple collections (N). Both types have one or more *SourceTargetCollectionPair*, which has the name(s) of the source collection(s) and the

Fig. 5. Architecture Depicting the Communication between Things, Services, and ViSiT's Server

name of the target collection. For example, the transformation shown in Fig. 1, is an *MToOneCollectionTransformation* that converts the “LowStock” items collection to a “PurchaseOrder” items collection. The two types of transformations are implemented in a similar way with the jigsaw puzzle metaphor. The *MtoN* transformation would just have more puzzle pieces. The underlying workflow implementation can handle both types of transformations, and it provides a different construct for each one. *MToOneCollectionTransformation* can also be used as *OneToOne*, because the latter is a subset of the first.

A *Transformation* also defines *PropertyMappings*, which indicate how the properties of a source collection are mapped to those of a target collection. A *PropertyMapping* has the name of the target property and that of the target collection. It also has an *Expression*, which indicates how the source property is mapped to the target property. An *Expression* can be composed of a combination of different types of *ExpressionParts*. A *CollectionProperty* expression part indicates the name of a property in the source model that is going to be directly mapped to its counterpart in the target model or

used as part of the formula before it is mapped. An *ArithmeticOperator* is used when calculations are needed. An *Expression* can also contain fixed value parts (*Number* or *Text*). It is also possible to use *Constructs* that represent parts of a conditional structure (if, then, else, end) with *Logical* and *Relational* operators, when decisions are required before the mapping is performed. Developers can create custom *Components* to extend the supported *Expression Parts*, which are used by end-users when defining *PropertyMappings*. Such *Components* have a visual HTML-based part that is displayed in the end-user support tool. End-users will use this visual part to place the *Component* in the *Expression* and supply it with the required parameters. Developers provide support for the *Component* in the workflow through a code-based function or a visual construct. Hence, when the jigsaw puzzle transformation is converted to the underlying implementation, it is possible to map the custom puzzle *Components* to their workflow counterpart.

An example of an expression with multiple parts is shown in Fig. 1 – d. Another example can be that of an IoT device transmitting a temperature in Celsius to a device that expects a temperature in Fahrenheit. In this example, the *Expression* would include a source *CollectionProperty* representing a temperature in Celsius, a multiplication *ArithmeticOperator*, a *NumberValue* of 1.8, an addition *ArithmeticOperator*, and another *NumberValue* of 32 ($\text{TemperatureInCelsius} \times 1.8 + 32$).

3.3 Architecture

The architecture depicted in Fig. 5 shows how IoT objects communicate with ViSiT's service in order to realize the links and transformations defined by the end-users. Stakeholders who are interested in implementing a system of connected IoT objects could use this architecture as a reference.

As a first step, IoT objects (*Things* and *Services*) are defined before end-users can start creating links and transformation. The definition of these objects includes the concepts discussed in Section 3.1, and can be performed by IT experts. The data that is supplied to the server when defining IoT objects, would include either a sample model (XML) or a schema definition (XSD) for the input and output data of *Actions*, *Events*, *Service Methods*, and *Service Notifications*. This data is needed in order to specify transformations from one model to another. *Things* can also be preprogrammed to report their data automatically to the server as shown in Fig. 5 (Step 3). The definition of *Things* and *Services* provides the server with information that can act as an Application Programming Interface (API) for these objects. It is possible to obtain some of this API data through an existing initiative called Hypercat¹³, which supports the discovery of information about IoT objects over the web. After the preliminary information is entered into the system, end-users can start defining links and transformations to realize scenarios that they conceived. The differences between our approach and other approaches, e.g., MakerSwarm (refer to Section 2.1), which use preprogrammed linking components is the lower dependency on IT experts. Hence, instead of relying on IT experts to realize the large number of linking combinations between IoT objects, these experts can just define the characteristics of each object once, and the rest is left to end-users. As mentioned above, this approach could spur the imagination of end-users since they are the primary stakeholders in the IoT objects.

¹³ **Hypercat**: <http://www.hypercat.io>

When a new *Thing* is introduced, if it is not already defined, it can send its data (actions, events, etc.) to the server's *NewThingListener* component, which will store this data in a database. Currently, this component listens to requests from *Things* that wish to identify themselves to the system. Hence, it is the *Things*' responsibility to initiate the request. In the future, we can extend this component with the ability to dynamically discover new *Things* in the surrounding environment, e.g., through WiFi or Bluetooth. At that point other factors, e.g., security, which are out of the scope of this paper would be addressed. Since this component might be hosted on a remote server, it could collaborate with a component that is deployed on a mobile device within the proximity of the *Thing* in order to support automatic discovery.

The *EventListener* checks for events that are raised by a *Thing*. It also relays the output data (XML) of events to the *CallManager*, which makes the appropriate call to either a *Thing* (action) or a *Service* (method) based on what the end-user has specified in the link. Before making a call, the *CallManager* triggers the *TransformationManager* in order to transform the source data to the format expected by the target as specified by the end-user in a transformation. The *NotificationListener* listens to notifications from services and relays the data to the *CallManager*. A database is used on the server to manage *Things*, *Services*, and *Transformations*.

3.4 Implementation: Why Puzzle and Workflows?

As we previously demonstrated by the example in Fig. 1, ViSiT uses a **jigsaw puzzle** metaphor (based on [Danado and Paternò 2014] and [Humble et al. 2003]) for supporting end-user development of transformations. The underlying executable implementation of the transformations is based on the **Windows Workflow Foundation** (WF) [Bruce Bukovics 2010], which is a visual framework for expressing executable workflows (similar to BPEL [Jordan et al. 2007]).

As we previously mentioned, the **jigsaw puzzle** metaphor is used in ViSiT by end-users to define transformations and links between IoT objects. This metaphor was chosen for several reasons including its appeal to end-users as shown by a study that compared it with other metaphors (refer to Puzzle in Section 2.1). Furthermore, we use the jigsaw puzzle metaphor to represent transformations of one property to another in a way that resembles mathematical equations. Hence, as shown in Fig. 1 – d, each target collection property is represented as a left-hand side variable of an equation. An equality operator is shown on the right edge of the puzzle piece. On the right-hand side of the equation, end-users can use jigsaw puzzle pieces as building blocks to compose expressions that indicate what the value of the target property is going to be. This paradigm is similar to the Siftables, which are small smart physical blocks with a screen [Merrill et al. 2007]. The Siftables allow end-users to define solutions by aligning the small building blocks next to each other. Siftables can be used, for example, to teach math to children by allowing them to align blocks that represent numbers and arithmetic operators. Hence, this formula-like paradigm is easily understood by end-users. A main advantage of the jigsaw metaphor lies in its affordance. The principle of affordance is what allows people to know how to use an object [Sharp et al. 2007] (p. 29). With a shape that has input and output edges, the jigsaw puzzle pieces can indicate to the end-users where each piece could fit.

The **Windows Workflow Foundation** (WF) was chosen as ViSiT's underlying technology for several reasons. First, it supports the composition of workflows at runtime. Hence, it is possible to convert the transformation defined by the end-users as a jigsaw puzzle to an executable workflow. Second, WF workflows can be saved

Fig. 6. Specifying Transformations Using ViSiT

using an XML-based format that can be loaded and executed when needed. Finally, WF workflows can be extended with custom constructs, a feature that we used to define two constructs that represent *MToOne* and *MToN* collection transformations (refer to Section 3.2). Some existing approaches define model transformations using templates, which are formed of a series of statements (e.g., [Ma et al. 2012]). In ViSiT, visual constructs in the jigsaw puzzle and in the workflow act like a parameterizable template that can be obtained from a toolbox and used to compose transformations.

4. CREATING AND EXECUTING TRANSFORMATIONS USING VISIT

The web-service shown as part of the architecture in Fig. 5 comprises the process of creating and executing transformations. This web-service hides much of the complexity from the end-user, and makes it possible to compose transformations using the jigsaw puzzle metaphor.

4.1 Creating Transformations

End-users can use a dedicated tool to create transformations with the jigsaw puzzle metaphor. After an end-user saves a jigsaw puzzle transformation, the authoring tool calls the web-service to create an executable workflow transformation. The process that is performed by the web-service to create this transformation has four main steps that are shown in Fig. 6.

Fig. 7. Executing a ViSiT Transformation

First, an end-user's jigsaw puzzle specification is validated in order to check for human errors. This validation involves several steps including checking the ordering of the formula parts, and checking for target properties that were not assigned. In case errors were detected, the transformation creation process is stopped, and the errors are reported back to the tool in order to notify the end-user. If non-critical points, e.g., missing expression for a non-mandatory target property, were detected the end-users are simply notified with a warning.

Second, the workflow is generated. Either sample models or XSDs can be provided. If the input and output data (source and target models) of the IoT objects was defined using sample models, then these models are loaded from the database as XML documents and an XML Schema Definition (XSD) is automatically generated based on them. If XSDs were provided from the beginning, then this conversion is skipped. Based on the XSD, C# classes are generated and compiled on-the-fly to generate a Dynamic Link Library (DLL) containing object-oriented (OO) definitions of both the source and the target models. Then, an empty ViSiT workflow is created and the DLL is attached to it. This workflow can manipulate the objects representing the models. We should note that although C# and Windows Workflow (WF) are used, ViSiT is technology independent since the transformations can be triggered by calling a web-service from any programming language, as illustrated by the architecture in Fig. 5.

Third, after the DLL is attached to the ViSiT workflow, two variables representing the compiled source and target model classes are declared inside it. These variables can be accessed by any of the constructs that are added to the workflow in order to perform the transformation. Then, the service automatically converts the jigsaw puzzle elements, which were defined by the end-users, into transformation constructs inside the workflow. The puzzle elements are transferred to the service as an XML document. The constructs are dynamically added to the workflow using the WF libraries in C#. The conversion starts by selecting the appropriate transformation constructs and assigning the source and target collection names after analyzing the XML representation of the puzzle pieces to see if the end-user selected an *MToOne* or an *MToN* transformation. Then, conditions such as sorting and filtering are converted to their textual counterparts and assigned to their relevant fields in the workflow. Afterwards, the names of the target properties are listed in the workflow. Finally, the expressions that assign values to the target properties are parsed, converted to their textual counterparts, and assigned to their respective fields in the workflow. Upon parsing expressions represented in the puzzle transformation, the system determines which workflow functions to use, e.g., list, Get, GetFA, etc. When

Fig. 8. Tool for End-Users – Link Editor UI

the workflow is executed, its constructs transform the source model to the target model. Finally, the complete workflow definition is saved to a database.

4.2 Executing Transformations

Once a ViSiT workflow is defined and stored in the database, it can be loaded and executed against any model conforming to the schema of the source model that was initially provided when defining the transformation.

As illustrated in Fig. 7, all that is required for executing a ViSiT workflow is passing two parameters to the web-service: the identifier number (ID#) of the workflow (transformation), and the source model that needs transformation. The web-service will then invoke server-side components, which will load the appropriate workflow from the database, pass it an object-oriented (OO) representation of the source model, and execute it. Once the workflow is executed, the constructs inside it will transform the OO source model to an OO target model. Finally, the OO target model is converted to an XML representation and transmitted back to the caller. Workflows can be executed using the support tool for testing purposes, or through the *Transformation Manager* component (refer to Fig. 5) to transform a source model to a target model.

5. TOOL SUPPORT

This work primarily targets end-users; hence a tool was developed to support end-users in wiring IoT objects and defining transformations. Nonetheless, developers are supported with their own IDE-style tool that allows them to define transformations visually using workflows. This section presents and discusses both tools. The process

Fig. 9. Tool for End-Users – Transformation Editor UI

of specifying links and transformations using ViSiT's supporting tools can be viewed online in a demonstration video¹⁴.

5.1 Tool for End-Users

The primary aim of the end-users' support tool is allowing non-programmers to link IoT objects and define transformations in a simple manner. This tool is web-based and was developed using HTML, JavaScript, and CSS, to make ViSiT accessible to end-users on a wider variety of devices and platforms. The tool provides end-users with a canvas that is based on the HTML 5 canvas and the Paper.js¹⁵ open-source vector graphics scripting framework.

5.1.1 Defining Links and Transformations. Consider an example where an end-user would like to connect a refrigerator to a shopping service as previously shown in Fig. 1. First, the end-user would define a link that connects an event from the refrigerator to one of the shopping service's methods (Fig. 1 – c). As shown in Fig. 8 –

¹⁴ ViSiT's Tool Support: <http://bit.ly/ViSiT>

¹⁵ Jürg Lehni & Jonathan Puckey, **Paper.js**: <http://paperjs.org>

a, the end-user can click on the things and services to view their related events and actions, and notifications and methods respectively. Then, the puzzle pieces can be dragged onto the canvas to compose the link (Fig. 8 – b). Finally, the end-user can click on the puzzle piece representing the link to start defining the transformation that is going to convert the data sent by the event or service-notification to the data expected by the action or service-method (Fig. 8 – c).

The UI that can be used to define transformations is shown in Fig. 9. In the refrigerator and shopping service example, the target collection is called “Purchase Order” and its properties are called “ItemRef” and “Qty”. This collection and its properties would have been predefined by an IT expert, as indicated in Fig. 5, for the shopping service’s “PlaceOrder” method. Hence, as shown in Fig. 9 – f, the target collection and properties are automatically obtained by the tool from this predefined data and are displayed on the left-hand side of the canvas. The properties are shown underneath their respective collection. The target collections are given folder icons, because end-users are usually familiar with a folder being a concept that groups items under it. In case there is one target collection, a single collection puzzle piece is shown. Otherwise, several collection puzzle pieces are shown, each with its properties underneath it. Each of the puzzle pieces representing a target property has an equality sign on its right edge, thereby making each property assignment resemble an equation. This sign could provide better affordance to the end-users by indicating that other puzzle pieces have to be placed on the right hand-side of the equation. Both source and target properties show an example under the property name. These examples are retrieved from the documentation that can be specified for *Things* and *Services* (refer to Section 3.1 and Fig. 3). Showing such examples could make the purpose of each property clearer for end-users who might not be domain experts.

To assign a value to a target property, end-users can compose an expression on the right-hand side of each equation. This composition can be done by dragging expression parts, represented as puzzle pieces, from the toolbox shown in Fig. 9 – j and dropping them onto the canvas. Expression parts are presented in the same way in the toolbox and on the canvas to help end-users in identifying what part they are dragging. Once a puzzle piece is dragged and placed next to another, it automatically snaps into place if the puzzle pieces fit otherwise the new piece is placed apart.

The expression parts (puzzle pieces) that are shown in Fig. 9 – j are automatically presented to the end-users who are not expected to extend these parts but simply use them. Expression parts that represent constructs and operators are predefined in the system. On the other hand, the expression parts that represent properties are generated based on the data specification of the events and actions of things and the methods and notifications of services (refer to Fig. 3). Data specifications are made by IT experts when defining things and services (refer to Fig. 5 – Step 1). The tool determines which things and services are involved based on the link that was specified by the end-user prior to specifying the transformation as shown in Fig. 8.

5.1.2 Validating and Testing Transformations. The tool supports direct validation when the end-user is composing expressions. If the pieces fit but there is an illogical ordering of pieces, the new piece is placed apart and a message is displayed to explain the reason. For example, as shown in Fig. 11, the tool prevents end-users from placing an “if” condition expression part directly after another and issues a warning that this operation is not possible. The end-user can then close the message and continue composing the expression. Following this simple puzzle composition

Fig. 10. Showing Help Upon Request

Fig. 11. Validation on the Canvas of the End-Users' Tool

(a) Standard Result Display**Source Data**

	Code	Description	Quantity
Edit	B00YT6QD70	Eggs	6
Edit	B00B04G0JU	Butter	1

Result (data after transformation)

ItemRef	Qty
B00YT6QD70	12
B00B04G0JU	2

** This result is based on sample data.*

(b) Custom Result Display**Source Data****Temperature from Sensor**

40 °C

Result (data after transformation)**Send Email through Service**

Subject: Wiring Closet Overheating

Body: The wiring closet temperature reached 40 °C.

(c) Editing Sample Data in Standard Result Display**Source Data**

	Code	Description	Quantity
Update Cancel	<input type="text" value="B00YT6QD70"/>	<input type="text" value="Eggs"/>	<input type="text" value="6"/>
Edit	B00B04G0JU	Butter	1

Fig. 12. Visualizing the Result After Testing the Transformation on a Sample Model

style helps in making transformation development, an activity usually performed by programmers, within the reach of end-users. Once the end-users are done working, they can save the transaction to the server (Fig. 9 – a). Additional validation is conducted upon saving, and a warning message is shown to the end-user indicating detected problems that need to be fixed before the transformation is saved. For example, when there is an “if/else” statement that will always yield the same value the end-user is notified to modify the expression. The tool will show a success message once the transformation is saved and there are no more validation errors. We should note that our system does not entirely prevent end-users from making mistakes, but it can react to these mistakes using validation as previously described.

The end-users can test a transformation by running it (Fig. 9 – b) on the sample source model in case it was provided (refer to Section 3.1 and Fig. 3). Upon running

the transformation, the source data and the result are shown in a default visualization. This visualization, shown in Fig. 12 – a, simply displays the data in grids that allow the end-users to visually compare the source to the result. It is also possible for the end-user to edit the source data, as shown in Fig. 12 – c, in order to try the transformation with different inputs. Software developers can create their own visualizations, such as the ones shown in Fig. 12 – b, using HTML, CSS, and JavaScript, and deploy them alongside the data specifications that they define (refer to Fig. 3). These custom visualizations could provide end-users with a more tailored way of observing the transformation results.

5.1.3 Other Features. The tool allows zooming the canvas to support a better view on different screen sizes (Fig. 9 – d). In case the collections and properties do not fit on the screen, it is possible to drag the canvas in order to scroll to the hidden parts. Dragging can be done by clicking on the canvas and moving it around (Fig. 9 – h). Adopting dragging instead of a scrollbar could make it easier to use the tool on touchscreen devices.

The expression parts in the toolbox can be filtered by category (Fig. 9 – i). When an expression part is added to the canvas, it can be removed by clicking on the close (“X”) icon in its top right corner (Fig. 9 – g). As shown in Fig. 9, expression parts are color-coded to make their type easier to identify. In Fig. 9, the canvas shows gridlines that allow the end-users to easily position the jigsaw puzzle pieces. It is possible to hide these gridlines by clicking on the button shown in Fig. 9 – c.

The end-users can request an explanation of a puzzle piece by clicking on the help button shown in Fig. 9 – e and then clicking on the puzzle piece itself. A callout that contains help information, such as the one shown in Fig. 10, will then be displayed above the puzzle piece. This type of contextual help could provide end-users with a quick way of learning about ViSiT, while using the tool.

5.1.4 Cognitive Dimensions. Recommendations from the “cognitive dimensions” framework [Green and Petre 1996] were taken into consideration when thinking about using the jigsaw puzzle metaphor as ViSiT’s visual notation. The way these recommendations have been taken into consideration is summarized as follows. Composing transformations with ViSiT is **consistent** due to the nature of the jigsaw puzzle metaphor. Hence, once the end-users learn the basics about composing expressions, the rest should not be hard to infer. Concerning **diffuseness**, when composing expressions each meaning is denoted by one puzzle piece that has a color, icon, and description. ViSiT’s visual notation is **terse** (compact) enough to effectively represent the mapping of several properties on the screen. The **visibility** of the transformations can be improved by making the puzzle pieces smaller in order to make a larger portion of a transformation visible on the screen. The size of each puzzle piece can be reduced to a size that is comparable to that of constructs of visual languages such as Scratch [Maloney et al. 2010] and TouchDevelop [Athreya et al. 2012]. However, we tried to maintain certain proportions that will make a puzzle piece easy to drag on hand-held devices. Nonetheless, a transformation can be zoomed out to make a bigger part of it fit on the screen. It is possible to use the end-users’ support tool in order to perform a **progressive evaluation** by running the transformation even if it is not fully complete. Hence, end-users would be able to evaluate their own progress at frequent intervals. The only constraint in this case is to complete the expressions that have already been started, but it is not mandatory

to compose all expressions before testing. We can consider that the adopted visual notation has a low *viscosity*, because little effort is required to change an expression. Expression parts can be simply deleted by clicking on the close (“X”) icon in their top right corner, and then these parts can be replaced by dragging other parts from the toolbox. In terms of the *abstraction gradient*, we can say that ViSiT is *abstraction-tolerant* because expressions can be composed using the provided expression parts such as constructs and operators. Nonetheless, additional components can be added to extend these expression parts with new abstractions (*Component* in Fig. 4). The end-users do not have to make a *premature commitment* in terms of the target property for which they start composing an expression. The target properties are listed underneath each other on the left-hand side of the canvas. However, the end-users can start composing expressions for any property they wish. Furthermore, since the tool divides the canvas into blocks with the target properties placed on one side, as seen in Fig. 9, the end-users do not have to look ahead in order to avoid possible “visual spaghetti” as might happen with line and box notations. The way the jigsaw puzzle notation is used in ViSiT does not impose *hard mental operations* on the end-user. This notation does not use complex conditionals that are connected together by lines. Each target property has one expression that maps one or more source properties to it. Hence, this eliminates complex search paths that could cause the end-users to move their fingers over the screen to try and follow the logic. *Hidden dependencies* are not a problem in ViSiT’s notation as it does not have hidden formulas that link components such as the ones present in spreadsheet applications or GOTO statements that are usually present in code-based programming languages. All the expression parts that form a transformation are visually presented to the end-user. The visual jigsaw puzzle notation used by ViSiT makes it less *error prone* than code-based languages that have a complex syntax with delimiters and separators that can cause end-user error slips. This notation also provides the desired *closeness of mapping*, in the sense that the end-users do not have to do a lot of “programming games” in order to define a transformation. Each expression simply has the target property on the left-hand side with an equality sign that indicates to the end-users that they can compose an expression on the right-hand side out of puzzle pieces. In terms of *role expressiveness*, answering what a certain puzzle piece is for can be done by selecting it and clicking on a help button in order to get a descriptive messages such as the one shown in Fig. 10. Although no *secondary notation* is provided for adding comments, the help messages alongside the examples that are placed under the names of properties can aid the end-users in knowing what each part of the transformation is meant to do. Another point that could improve the end-users’ understanding is that the notation follows the same style of expression composition, whereby each source property is mapped to a target property using an inline expression.

5.2 Tool for Developers

Developers can also define and execute transformations using their own tool, which provides an IDE-style UI that they are usually familiar with. The tool we created for developers supports the design of transformations using workflows that are based on the Windows Workflow Foundation, which is the underlying implementation of ViSiT.

This tool was added to our IDE Cedar Studio [Akiki et al. 2013a] and is shown in Fig. 13. It hosts the Windows Workflow Foundation design component, and offers a

Fig. 13. Tool for Developers is Part of the Cedar Studio IDE

toolbox with a set of basic programming constructs, e.g., if conditions, loops, etc., in addition to transformation-specific constructs such as the one illustrated in Fig. 1 – e.

In order to define a new transformation in Cedar Studio, developers can click on the “New Model Transformation” menu item. Upon doing so, they will be prompted to select sample source and target models represented as XML. These models do not have to be the final ones, but they merely serve as a template for generating an object-oriented (OO) representation. It is possible to supply XSD files instead. ViSiT uses an OO representation of the source and target models, because it can be simpler to work with than an XML document. For example, the developer could simply reference a collection by using its name, e.g., “PurchaseOrder”. On the other hand, with XML, a language like XPath is needed to select all the nodes of a certain type.

An example ViSiT workflow is illustrated inside Cedar Studio in Fig. 13. We can observe the toolbox in Fig. 13 – a, containing the visual constructs that are used to compose a model transformation. Constructs are dragged from the toolbox and dropped onto the canvas shown in Fig. 13 – b. The properties, e.g., parameters, of these constructs are edited either directly on the canvas, or through a property-box (Fig. 13 – c). Eventually, the workflow is saved to a database.

6. WHAT KINDS OF TRANSFORMATIONS CAN BE REALIZED USING VISIT?

We used ViSiT to develop transformations based on examples that were realized by existing model transformation languages. Although these examples are not related to an IoT setting, our aim in this section is to highlight further the possibility of using ViSiT for realizing a variety of scenarios while providing the simplicity required by end-users. The specification is shown in both the jigsaw puzzle notation that is used

(a) Source Model Represented using XML

```

<Families>
  <Family lastName="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
    <sons firstName="David"/>
    <sons firstName="Dylan"/>
    <daughters firstName="Kelly"/>
  </Family>
</Families>

```

(b) Target Model Represented using XML

```

<!--Note that the target model is based on a
different meta-model than the source model.-->
<Persons>
  <Male fullName="Jim March"/>
  <Male fullName="Brandon March"/>
  <Male fullName="Peter Sailor"/>
  <Male fullName="David Sailor"/>
  <Male fullName="Dylan Sailor"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Brenda March"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Kelly Sailor"/>
</Persons>

```

(c) ViSiT Jigsaw Puzzle Transformation **(d) ViSiT M To N Workflow Transformation Construct**

Transform Families to Persons		
SourceCollection	TargetCollection	Condition
mother	Female	
daughters	Female	
father	Male	
sons	Male	
Property	Values	
fullName	Get("firstName") + " " + GetFA("lastName")	

Fig. 14. Families to Persons Example

by end-users and the underlying workflow notation that is automatically generated (can be also used by programmers if they wish).

The examples were chosen from the ATL Zoo¹⁶ due to their implementation in ATL, which is a well-established transformation language that is popular in the literature. We should restate that we are not claiming that ViSiT is a replacement for well-established model transformation languages. Hence, the demonstration of a ViSiT solution for these examples is not a direct comparison to transformation

¹⁶ ATL Zoo: http://www.eclipse.org/atl/documentation/basicExamples_Patterns

languages that are code-based such as ATL or to those that offer a technical visual notation such as Henshin. It is rather a way of showing that ViSiT offers some of the capabilities of these languages, which can be leveraged by end-users through a visual notation that is simple for them to use. Please note that the names of the examples and those of the elements on the source and target models were used as they were defined by the ATL Zoo, in order to make it easier for the readers to refer back to that source if necessary.

6.1 Families to Persons Example

One of the ATL Zoo examples that we implemented is demonstrated in Fig. 14. This example requires the transformation of a group of families into a list of persons. The source and target models are shown in Fig. 14 – a, and Fig. 14 – b respectively. We can see that the source model has a number of families each having one father, one mother, and several daughters and sons. However, the target model is based on a different meta-model, which has a list of males and females. The father and sons of each family in the source model should be transformed to males on the target model, while the mother and daughters should be transformed to females. This is an example of an *MToNCollectionTransformation* (refer to Section 3.2).

As shown on ATL Zoo, the sought-after transformation can be defined by writing 42 lines of ATL code. On the other hand, if we use ViSiT, end-users can achieve the same result by using the visual puzzle pieces shown in Fig. 14 – c. We can see that in this example there are two target collections: “Female” and “Male”. Each of these target collections has two source collections. Hence, “Female” and “Male” are placed on the left-hand side of “mother” and “daughters” and “father” and “sons” respectively. Underneath the collections the “firstName” and “lastName” source properties are mapped to the “fullName” target property with a space separating them. It is possible in this case to add the target collections directly above each other, because the target properties in both collections are the same. If this was not the case, then the collections would have been placed separately. This is an example of an *MToOneCollectionTransformation* (refer to Section 3.2).

The jigsaw puzzle representation is converted to the workflow transformation constructs shown in Fig. 14 – d. We can see that this construct has a data-grid for mapping the source and target collections. The source collections “mother”, “daughters”, “father”, and “sons” are mapped to the target collections “Male” and “Female”. This construct also has a data-grid containing the names of the properties in the target collection and their values. The field for assigning the properties’ values supports expressions. For example, the value for the “fullName” property is the “firstName” from the family member tag and the “familyName” from the family tag separated by a space. Two methods can be used to retrieve the values. The first method is called “Get”, and can retrieve a value directly from the object being transformed such as: father, mother, son, and daughter. The second method is called “GetFA”, which stands for “get from ancestor”. This method retrieves a value from an ancestor object of the element being transformed, which in this example is “family”. It searches first in the immediate ancestor and moves upwards in case the property was not found. The power of the “GetFA” method lies in eliminating the need to write a separate helper, which retrieves the “familyName” from the family tag. In other cases where more values have to be retrieved from ancestor tags, it is possible to simply call “GetFA” with a different parameter.

(a) Source Model Represented using XML

```

<Node name="T3h main node">
  <children type="Node" name="another node">
    <children type="Leaf" name="SUB MEDIUM azdzere"
      size="medium"/>
    <children type="Leaf" name="SUB SMALL fdfsdf"/>
    <children type="Leaf" name="SUB SMALL aaaaaa"/>
  </children>
  <children type="Leaf" name="MEDIUM a leaf in main node"
    size="medium"/>
  <children type="Leaf" name="BIG another leaf in main node"
    size="big"/>
  <children type="Leaf" name="SMALL leaf in main node"/>
  <children type="Node" name="last node">
    <children type="Leaf" name="SUB2 SMALL2"/>
    <children type="Leaf" name="SUB2 MEDIUM" size="medium"/>
    <children type="Leaf" name="SUB2 SMALL"/>
  </children>
</Node>

```

(b) Target Model Represented using XML

```

<!--Note that the target model is based on a
different meta-model than the source model.-->
<elmlst name="T3h main node">
  <elements name="BIG another leaf in main node"/>
  <elements name="SUB MEDIUM azdzere"/>
  <elements name="MEDIUM a leaf in main node"/>
  <elements name="SUB2 MEDIUM"/>
  <elements name="SUB SMALL fdfsdf"/>
  <elements name="SUB SMALL aaaaaa"/>
  <elements name="SMALL leaf in main node"/>
  <elements name="SUB2 SMALL2"/>
  <elements name="SUB2 SMALL"/>
</elmlst>

```

(c) ViSiT Jigsaw Puzzle Transformation **(d) ViSiT M To One Workflow Transformation Construct**

Transform Tree to List	
Source Collections	children
Target Collection	elements
Condition	type = "Leaf"
Order By	size list("big", "medium", "small")
Property	Values
name	Get("name")

Fig. 15. Tree to List Example

6.2 Tree to List Example

In this example, the source model shown in Fig. 15 – a, has a tree structure that should be transformed to a target model that has a list structure as shown in Fig. 15 – b.

Each child inside the tree structure of the source model can be either a “node” that has children or a “leaf” with no children. Each child has a name and can have a size, e.g., big, medium, etc. The required transformation should convert the leaf children of the source model’s tree structure to elements inside the list structure of the target model. The names have to be mapped as they are. However, the elements inside the target model must be sorted by the values of the tree children’s size property.

The visual construct illustrated in Fig. 15 – d realizes this transformation, which takes 35 lines of code to define with ATL. With ViSiT, end-users can achieve the same result by adding the visual puzzle pieces shown in Fig. 15 – c. In this example, mapping the source and target properties is relatively straight-forward. We only need to set the name of a list element to the name of its relevant tree child. Since there is only one source collection, it is not necessary to specify it next to the target collection as was done in the Families to Persons example (refer to Section 6.1). The tree-to-list example portrays the use of two additional constructs, namely Filter and Sort. The puzzle pieces representing these constructs are placed above the collection and property mappings. The “Filter” construct is used to filter-out the source collection elements that should be transformed to the target collection. In this case, the filter has a condition on the “type” field to indicate that only the “Leaf” elements should be transformed. Hence, the tree children of type “Node” will not be converted to elements on the target model, but will be simply used to get their children that are of type “leaf”. The “Sort” construct can order the outcome of the transformation by one or more properties. In this case, the result is sorted by the “size” property from “large” to “small”. End-users can specify the field(s) on which the sorting should be done and follow it with values, which can be entered using the *TextValue* construct. ViSiT supports three types of sorting on one or more properties: ascending (asc), descending (desc), and list. The type “list” indicates that the collection should be sorted based on the order of values in a predefined list. In this example, the list has the possible values of the “size” property of a tree child including: “big”, “medium” and “small”. Sorting by specific properties in ascending and descending order can be specified by placing puzzle pieces that represent “asc” and “desc” after the relevant properties.

In the tree-to-list example, the jigsaw puzzle transformation is converted to an M to One workflow transformation construct, because there is only one target collection. The source collection is specified to be “children”. The internal implementation of the construct will automatically work recursively to get the nested collections of children. The target collection is specified to be the “elements” in the target model. Hence, for each child in the source model an element will be created in the target model. We can see that the condition “type=Leaf” is specified in the condition field. The sorting expression is specified in the “Order By” field. As was done with the previous example, the property/values data-grid is used to indicate the mapping of the source properties to their target counterparts. The value of the “name” target property is Get(“name”), which indicates that the value should be obtained directly from the “name” source property of the tree children.

7. TECHNICAL EVALUATION

In this evaluation, we defined transformations and measured ViSiT’s efficiency and scalability. We also created an application to demonstrate ViSiT’s viability using a real-world scenario.

7.1 Efficiency and Scalability Evaluation

We tested ViSiT’s efficiency and scalability using the transformations that were developed to realize three examples. Two of these examples, namely families-to-persons and tree-to-list, were discussed in Section 6. The third example, purchase-order-to-sales-order, was developed to test whether ViSiT can be applied outside the IoT domain. We did not include the details of the third example, because it is outside

Table I. Information on the Source Models Used for the Efficiency and Scalability Test

Example	Model Size and Description		
	Small	Medium	Large
Families to Persons	95 KB (3000 XML Tags) 500 Families (4-5 members each)	283 KB (9000 XML Tags) 1,500 Families (4-5 members each)	849 KB (27,000 XML Tags) 4,500 Families (4-5 members each)
Tree to List	29 KB (500 XML Tags) 500 Children (80 nodes, 420 leafs)	87 KB (1500 XML Tags) 1,500 Children (240 nodes, 1,260 leafs)	259 KB (4500 XML Tags) 4,500 Children (720 nodes, 3,780 leafs)
Purchase Order to Sales Order	441 KB (23,003 XML Tags) 500 Purchase Orders	1,320 KB (69,009 XML Tags) 1500 Purchase Orders	3,960 KB (207,027 XML Tags) 4500 Purchase Orders

Each model fitting under: “small”, “medium”, or “large”, is 3 times bigger than the one before it. We also used two other model sizes, one is between small and medium, and the other is between medium and large. These models are 1.5 times bigger than the ones before them.

Fig. 16. Results of Efficiency and Scalability Test

the scope of this paper. The tests were done using models varying in size and complexity, and the results demonstrated that ViSiT is efficient and scalable.

The tests were conducted on a computer with an Intel Core i5 2.5GHz CPU, and 4 GB of RAM. Information describing the source models, which were used in these tests, is provided in Table I. We used five different input models for each of the tested examples. To make these different input models comparable on one graph, we used the number of domain objects as a size benchmark. The smallest model has 500 domain objects such as: family, tree child, and purchase order. In some examples these objects also have children such as: family members, purchase order items, etc. Hence, the number of XML tags and file size of each model are also stated in Table I to give a clearer indication about the size of the models that were used. The largest model has 4500 domain objects. The main model sizes are: small (500), medium (1500), and large (4500), with intermediate sizes of 750 and 2250 domain objects between small and medium, and medium and large respectively. The information on the two models containing 750 and 2250 domain objects is not shown in Table I in order to make it more legible. The use of these different sizes allowed us to show that ViSiT is not only efficient but is also scalable.

The results of the tests are illustrated by the chart in Fig. 16. The fitting curves show that the execution times are polynomial of the 3rd order with R^2 ranging between 0.997 and 0.999. Each transformation took between 2 and 12 seconds to be executed. ViSiT is not aimed at transforming large-scale data sources such as databases, all at once, but it is more intended for converting small models one at a time. Hence, we deem its efficiency to be acceptable considering the models in reality will be smaller in size than the “small” models, which were used in this test. So every individual transformation, e.g., refrigerator low-stock items to shopping service purchase order (refer to Fig. 1), would generally not take more than a few milliseconds. Nonetheless, in the future, we aim at improving ViSiT’s efficiency by tuning our algorithms further, in order to obtain an execution time that does not exceed a few milliseconds with the “small” models used in this test.

7.2 Application

This paper presented examples on how IoT devices communicate in the background after an end-user defines a link and a transformation using ViSiT. One example was related to a refrigerator that automatically orders low-stock items from a shopping service (refer to Section 1.2). It is also possible to use ViSiT for applications that require more direct control by the end-user. In this section, we present an example application that involves controlling a Lego Mindstorms robot with an Xbox controller.

We constructed the robot shown in Fig. 17 – a using the Lego Mindstorms robotics kit. The basic capabilities of this robot include moving around and shooting plastic pellets with the help of three motors. It has left and right motors for movement and a third motor for shooting. The Xbox controller shown in Fig. 17 – c was not initially intended for controlling this particular robot and can transmit a movement pattern (forward, backward, left, and right) and a set of actions (buttons A, B, X, and Y). The Xbox controller can transmit data to a PC using the Bluetooth adapter shown in Fig. 17 – b. The Lego Mindstorms robot can also be controlled by deploying a software to it or by passing it commands through a Bluetooth connection.

This example application is particular, because the devices (controller and robot) cannot communicate with web-services themselves. Hence, intermediary applications that communicate with the devices through Bluetooth were used for communicating with ViSiT’s web-service and relaying commands between the controller and the robot. We can see in Fig. 17 that a desktop application that receives commands from the Xbox controller can report an event to the ViSiT web-service. Another application, which can be desktop or mobile based, receives a notification from the web-service and reports the commands to the robot. Different configurations are also possible. Assume that certain IoT objects were capable of communicating with web-services, e.g., through a Raspberry Pi, there would be no need for the intermediary applications. Also, in this example, if the web-service was hosted on a machine that had Bluetooth access to the devices, e.g., in a home environment, the communication with the devices could happen directly through that machine.

We can see in Fig. 18 the ViSiT transformation that is needed to convert some of the commands supplied by the Xbox controller into motor power values, which will make the Lego Mindstorms robot function. This transformation is executed by the service to make the data sent by the controller compatible with the data expected by the robot. We can see in Fig. 18 that this example introduces a custom component (refer to Section 3.2) for converting the movement data sent by the Xbox controller to

* In case the IoT objects have the ability to communicate with services, e.g., through a Raspberry Pi, there will be no need for intermediary applications like the ones represented in this figure.

Fig. 17. Lego Mindstorms Robot Controlled by an Xbox Controller

Fig. 18. ViSiT Transformation for Connecting an Xbox Controller to a Lego Mindstorms Robot

left and right motor powers that the robot expects. Primitive constructs, such as “if, then, else”, could have also been used in this case instead of the custom component.

This example demonstrates a practical application of how different objects, which were not intended to work together can be linked using ViSiT. The ever-growing diversity and abundance of IoT objects coupled with end-user imagination and the support of ViSiT could make many of these kinds of applications easily realizable.

Fig. 19. Demographic Information of the Usability Study Participants

8. USABILITY EVALUATION

This study focuses on evaluating the usability of ViSiT's jigsaw puzzle notation and its end-user support tool with participants who have no prior software development experience. We recruited 30 participants through Amazon Mechanical Turk (AMT)¹⁷. The mean time spent by the participants on this study is 15 minutes and 27 seconds. A summary of their demographic information is shown in Fig. 19. This information was collected by asking the participants to answer a questionnaire before starting the study. They were asked about their computer literacy level using a set of eight questions, which were taken from an existing computer literacy test [Kay 1993]. The participants were also asked to state whether or not they had any prior software development experience either using a programming language such as C# or Java, or using an end-user development environment like Scratch. This information allowed us to restrict our participants to those who do not have any significant prior software development experience. Before starting the study, the participants were given an explanation about its purpose. This explanation demonstrated a few examples that showed how to compose transformations related to IoT scenarios using ViSiT's jigsaw puzzle constructs.

¹⁷ Amazon Mechanical Turk: <https://www.mturk.com>

a) Trashcan Orders from Shopping Service

Consider that your trashcan is smart and can report items that are thrown in it. You would like to connect this trashcan to an online shopping service, so that when an item is thrown a new one is ordered automatically. The service expects an “ItemRef” and a “Qty”, and the trashcan sends a “Code” and a “Quantity”.

b) Armband Controls Drone

Consider that you would like to control a drone with an electronic armband. The drone expects a “Speed” and a “Direction” as inputs, and the armband provides the same parameters as an output. However, you have to multiply by 2 the speed that the armband sends before passing it to the drone.

c) Blood Pressure Monitor Changes Color of a Lightbulb

Consider that you would like to link a blood pressure monitor to a lightbulb. If your blood pressure increases more than 1.5 (120/80) the light would turn red otherwise its color would be white. This behavior is achieved by checking the “Blood Pressure” reported by the monitor before assigning a “Color” to the lightbulb.

Fig. 20. Scenarios for which the Participants were asked to Define Transformations in the User Study, and the ViSiT Transformations that the Participants were expected to Define for Each Scenario

8.1 Study Design

The participants were presented with the three scenarios described in Fig. 20. They were asked to realize these scenarios by defining transformations using ViSiT’s end-user support tool (refer to Section 5.1). After completing the tasks, the participants were asked to offer their perception of ViSiT’s usability and learnability by answering some rating questions. They were also asked to propose, if possible, scenarios from their everyday lives where they think that ViSiT could be useful. Such scenarios give

an indication about the creativity that end-users may exhibit if they were empowered by ViSiT to wire IoT objects together.

The transformations that the participants defined were recorded in addition to the time taken to define each one. Recording this information allowed us to check the extent to which the participants were actually able to use ViSiT and compare that to the feedback that they gave by answering the questionnaire. We observed the percentage of the scenarios the participants were able to complete and the amount of time it took them to finish.

For measuring the participants' perceived usability, we used the System Usability Scale (SUS) [Brooke 1996] and the Product Reaction Cards (PRCs) [Benedek and Miner 2002]. By asking participants to answer the SUS questionnaire, we can quantitatively measure their perceived usability. SUS is complemented by the PRCs that allow participants to select from a predefined set of terms the ones they think are best fit to describe the system. The participants were given from the original 118 terms a subset that included the following 12 positive and 12 negative terms:

–*Positive*: Clear, Creative, Easy to use, Empowering, Effective, Familiar, Friendly, Fun, Stimulating, Straight Forward, Useful, Valuable.

–*Negative*: Annoying, Complex, Confusing, Difficult, Dull, Fragile, Frustrating, Hard to Use, Ineffective, Not Valuable, Overwhelming, Too Technical

Six learning barriers in end-user programming systems were identified by prior work [Ko et al. 2004]. The participants of our study were asked questions to see to what extent they thought these barriers existed in ViSiT and its supporting tool. The answers provide an indication of whether the participants perceived ViSiT and its supporting tool to be learnable or not.

8.2 Measures to Ensure Feedback Quality

Since our end-user support tool is web-based, we chose to use Amazon Mechanical Turk (AMT), because it offers tools for successfully completing an online research study and can provide access to a wide variety of participants at a relatively low cost. AMT can be used for collecting data, which is as reliable as those obtained via traditional methods [Buhrmester et al. 2011]. The existing literature has examples, which indicate the viability of using AMT for conducting research studies [Paolacci et al. 2010]. Nonetheless, we took some measures to ensure feedback quality.

Several measures were taken to ensure the recruitment of serious participants in order to obtain high quality feedback. One measure is setting filters that are provided by AMT. These filters restrict the participants, who qualify to take part in this study. We specified that a qualified participant must have previously completed more than 5000 tasks (hits) on AMT with an accuracy of over 95%. The accuracy indicates whether a person did the assigned task properly or arbitrarily, and it is specified by the person who requests the task. AMT workers with a high accuracy are serious about selecting tasks that they can complete accurately, in order not to negatively affect their rating. Hence, they usually provide reliable answers.

Gold standard awareness questions are used in crowdsourcing to check the seriousness of participants (workers) [Eickhoff and de Vries 2013], and are one of the measures that we used in this study. These questions are usually very easy to answer by serious participants. Those who do not properly answer these questions would have most likely answered arbitrarily; hence their input can be excluded. One of the gold standard questions that we included in this study asked the participants about the number of scenarios (namely three) for which they were required to

Fig. 21. Time Taken by the Participants to Complete Each of the Three Scenarios in the Study

Table II. Summary Statistics of the Scenario Completion Times

	Time in Seconds		
	Mean	Median	Standard Error (SE)
Scenario 1	151.26	128	16.43
Scenario 2	94.8	73	15.88
Scenario 3	111.8	110	9.70

*80% of the participants were able to correctly define transformations for all three scenarios

compose a transformation. Two gold standard questions were also placed among the demographic information questions on computer literacy. The first question asked the participants to select from a list of devices the one (namely a mouse), which is not used for storing data. The second question asked the participants to rate their ability to use an operating system. This question was presented in two different wordings, which were placed apart to check whether the participants are carefully reading each question and are providing consistent answers.

As a final measure, we also logged the amounts of time taken by each participant to complete the whole study, define each transformation, and answer the feedback questions. If certain participants finish the study in an unreasonable amount of time that is very short in comparison to the other participants, it could mean that they provided arbitrary answers.

8.3 Results

The results of this study generally offered positive indications about the usability and learnability of ViSiT. These results also showed some interesting real-life scenarios that the participants conceived for using ViSiT.

The box-plot shown in Fig. 21 presents the time it took the participants to define a transformation for each of the scenarios previously presented in Fig. 20. Most of the participants (80%) were able to correctly define transformations for all three scenarios. We can see that although the first scenario was the least complicated the participants required on average more time to complete it, with the mean being 151.26 seconds. Scenarios 2 and 3 required the addition of more puzzle elements than Scenario 1. Yet, it took the participants a mean time of 94.8 and 111.8 seconds to complete Scenarios 1 and 2 respectively. These completion times indicate that the

*Each participant was asked to select five product reaction cards that mostly describe ViSiT and its supporting tool

Fig. 22. Perceived Usability Expressed by SUS Scores (top) and Product Reaction Cards (bottom)

participants familiarized themselves with how to use the tool in the first scenario and were able to construct the second and third scenarios more efficiently.

The results of the participants' perceived usability are presented in Fig. 22 using SUS scores (top) and Product Reaction Cards (bottom). The mean SUS score was 76.1 and the selected product reaction cards were mostly positive. Some of the selected cards, e.g., creative, fun and stimulating, show emotional satisfaction. Other cards, e.g., effective, easy-to-use and straight-forward, show the participants' satisfaction with using the system's functionality. Some choices, e.g., useful, valuable, and empowering, show a positive end-user perception of the system's utility.

One group of six participants chose both the "complex" and "confusing" product reaction cards, and an additional participant also chose "complex". Nonetheless, we can still say that the overall result provides a positive indication due to several reasons. The participants of the study are non-programmers who generally have average computer usage skills. Nonetheless, they were able to use ViSiT for developing basic transformations. This type of development task is usually restricted to programmers who have expertise in using traditional model transformation languages. We think that the participants could find our system easier to use if they get a training session or are offered a more extensive tutorial. In this study, they were simply offered a basic explanation of what the system is for and how to use it. The participants gave positive feedback about the system's learnability as shown by their answers to the questions presented in Fig. 23. As explained previously, the participants' perception of learnability was elicited by asking six questions about learning barriers defined in prior research [Ko et al. 2004]. A general question about

*The questions on the six learning barriers are negatively written, while the one on the system's overall learnability is positively written.

*The questions on the six learning barriers are negatively written, while the one on the system's overall learnability is positively written.

Fig. 23. Participants' Perceived Learnability Expressed as Mean Scores on Questions about Learning Barriers and a Question about the Overall Ease of Learning the System (1=Strongly Disagree; 5=Strongly Agree)

the system's overall learnability was also asked. We can see that the participants considered the effect of the learning barriers to be low, and perceived the system to be generally easy to learn.

When asked to suggest scenarios where they thought ViSiT could be used in real-life, 75% of the participants provided reasonable answers. These answers indicate that end-users are capable of envisioning useful real-life IoT scenarios, if they were empowered with an implementation technology such as ViSiT. Some answers were general and were restricted to categories of applications, e.g., home appliances, while others were more specific and explicitly described a scenario. What follows are some examples of the different types of scenarios that were suggested by the participants.

The following are examples of some general answers:

- “I think it would be good to use on all kinds of electronics, like sensors and monitors.” (P1)
- “It is suitable for home products.” (P2)
- “Electrical appliances, ordering groceries, paying bills, etc.” (P3)

Some more specific answers include the following:

- “When a thermometer shows that a child's temperature is above 102 it automatically notifies me that the child needs a fever tablet.” (P4)
- “Medicine cabinet that automatically orders more supplements” (P5)
- “Connecting the refrigerator with your smart phone, so you know what you need to buy” (P6)
- “Connecting different toys or game consoles together. When I connected my Xbox one with a Kinect it made things more interesting.” (P7)
- “A laundry washer that is connected to the supply of detergent. When the amount of detergent remaining passes a preset limit, it orders more.” (P8)

One participant answered this question by expressing how excited he was about ViSiT and wrote the following statement:

– *“This is an amazing system and I wish I could learn it a bit more and implement it in real life. Can you imagine? If temp > 55 then open door else turn on heat. Or something like that. I’d love to learn it.”* (P9)

9. THREATS TO VALIDITY AND LIMITATIONS

In this section, we explain the threats to validity in the user study and the technical limitations of ViSiT in comparison to existing model-transformation languages.

The scenarios that the participants of the user study were asked to develop cover a subset of the visual constructs that are supported by ViSiT. Our aim was to obtain an indication about the usability of ViSiT and its supporting tool. Since the other constructs, e.g., filtering and sorting, follow the same jigsaw puzzle paradigm the results of the study could be indicative of their usability as well. Yet, the end-users may require some more time to learn and understand all the constructs and how to apply them in a practical scenario. We think that this could be facilitated by providing video tutorials that demonstrate the creation of real-life IoT scenarios. Also, further in-situ evaluation can be conducted in the future in order to obtain additional insights on how end-users would handle more complex scenarios and what typical mistakes and misunderstandings might arise.

As we previously mentioned, ViSiT is not intended to replace existing, mature model transformation languages. Instead, it is meant to empower end-users with some of the capabilities of those languages, so that they can wire IoT objects without needing programming skills. Hence, ViSiT does not support certain features, e.g., bidirectional transformations, which are present in existing model transformation languages. In terms of efficiency, some existing model transformation languages could also be few seconds faster than ViSiT’s underlying implementation. However, as it is currently implemented ViSiT is enough for realizing many IoT scenarios since it will mostly be used to convert small models at a time. Nonetheless, in the future, we aim to improve the efficiency of ViSiT’s underlying implementation. Additionally, we aim to extend the jigsaw puzzle notation to support more features that are currently realizable using ViSiT’s underlying workflow implementation.

10. CONCLUSIONS

Our aim in this work is to empower end-users to conceive and realize useful scenarios that involve wiring together a wide variety of IoT objects (things and services). The main challenge that this work aims to address is allowing end-users who have no programming experience to define transformations, which can convert the data being sent from one IoT object to the data expected by another. We attempted to address this challenge by presenting ViSiT (Visual Simple Transformations). With ViSiT, end-users can realize scenarios such as wiring a smart refrigerator to an online shopping service in order to automatically replenish low-stock items.

ViSiT allows end-users who are non-programmers to define transformations using a jigsaw puzzle notation. The meta-models that embody the concept behind ViSiT were presented and explained. This paper also presented an architecture that can serve as a reference for developing a network of connected IoT objects. These objects can leverage ViSiT to transform their data in a way that enables meaningful communication between them. We presented a web-based tool for supporting end-

users in defining and testing transformations using ViSiT's jigsaw puzzle notation. The transformations that are defined by end-users using jigsaw puzzle pieces are automatically converted into an underlying executable workflow. Although our approach primarily targets end-users who are non-programmers, software developers could also use it to define transformations. Since software developers may be familiar with a workflow metaphor, e.g., for defining business rules, we also presented a tool that allows them to leverage this metaphor for defining transformations.

We developed example transformations to demonstrate ViSiT's efficiency and scalability. We also developed an example of controlling a Lego Mindstorms robot with an Xbox controller to show ViSiT's practicality. ViSiT was also evaluated from an end-user perspective by conducting a study that showed positive indications about our system's usability and learnability, in addition to the end-users' ability to conceive real-life scenarios where ViSiT could be useful.

As the IoT becomes more widespread, solutions like ViSiT will play an important role in involving end-users in activities that were traditionally restricted to technical stakeholders such as software developers. End-users could, in many cases, assume responsibility of many activities that might be too expensive to assign to software developers. In the future, it is possible to extend ViSiT with additional functionality that would allow it to empower end-users even more. More functionality can be added to ViSiT by extending its jigsaw puzzle notation to support more of the procedural logic constructs that are supported by its underlying workflow technology. It could also be useful to extend our end-user support tool to enable end-users to collaborate on defining transformations through a form of crowdsourcing. The tool could enable end-users to define, share, and rate transformations for a wide variety of IoT objects.

ACKNOWLEDGEMENTS

This work is supported by ERC Advanced Grant 291652.

REFERENCES

- Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. 2006. The Design of a Language for Model Transformations. *Software & Systems Modeling* 5, 3 (2006), 261–288.
- Aditya Agrawal, Tihamer Leventovszky, Jon Sprinkle, Feng Shi, and Gabor Karsai. 2002. Generative Programming via Graph Transformations in the Model-Driven Architecture. In *Workshop on Generative Techniques in the Context of Model Driven Architecture, OOPSLA*.
- David H. Akehurst, Behzad Bordbar, Michael J. Evans, W. Gareth J. Howells, and Klaus D. McDonald-Maier. 2006. SiTra: Simple Transformations in Java. In *Model Driven Engineering Languages and Systems*. Springer, 351–364.
- Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. 2013a. Cedar Studio: An IDE Supporting Adaptive Model-Driven User Interfaces for Enterprise Applications. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. London, UK: ACM, 139–144. DOI:<https://doi.org/10.1145/2494603.2480332>
- Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. 2014. Integrating Adaptive User Interface Capabilities in Enterprise Applications. In *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India: IEEE/ACM, 712–723.
- Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. 2013b. RBUI: Simplifying Enterprise Application User Interfaces through Engineering Role-Based Adaptive Behavior. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. London, UK: ACM, 3–12. DOI:<https://doi.org/10.1145/2494603.2480297>
- Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. 2011. eMoflon: Leveraging EMF and Professional CASE Tools. *Informatik* (2011), 281.
- Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems*. Springer, 121–135.
- Balaji Athreya, Faezeh Bahmani, Alex Diede, and Chris Scaffidi. 2012. End-user programmers on the loose: A study of programming on the phone for the phone. In *Visual Languages and Human-Centric*

- Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 75–82.
- Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A Survey. *Computer networks* 54, 15 (2010), 2787–2805.
- András Balogh and Dániel Varró. 2006. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proceedings of the 2006 ACM Symposium on Applied Computing*. SAC '06. New York, NY, USA: ACM, 1280–1287. DOI:https://doi.org/10.1145/1141277.1141575
- Alessandro Bassi et al. eds. 2013. *Enabling Things to Talk - Designing IoT Solutions With the IoT Architectural Reference Model*. Springer.
- Joey Benedek and Trish Miner. 2002. Measuring Desirability: New Methods for Evaluating Desirability in a Usability Lab Setting. *Proceedings of Usability Professionals Association 2003* (2002), 8–12.
- Amel Bennaceur and Valérie Issarny. 2015. Automated Synthesis of Mediators to Support Component Interoperability. *Software Engineering, IEEE Transactions on* 41, 3 (2015), 221–240.
- Gábor Bergmann et al. 2015. VIATRA 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*. Springer, 101–110.
- Scott Boag et al. 2002. *XQuery 1.0: An XML Query Language*, W3C.
- Behzad Bordbar, Gareth Howells, Michael Evans, and Athanasios Staikopoulos. 2007. Model Transformation from OWL-S to BPEL via SiTra. In *Model Driven Architecture-Foundations and Applications*. Springer, 43–58.
- Ansgar Bredendfeld and Raul Camposano. 1995. Tool Integration and Construction Using Generated Graph-based Design Representations. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*. San Francisco, California, USA: ACM, 94–99.
- J. Brooke. 1996. SUS: A Quick and Dirty Usability Scale. In P. W. Jordan, B. Weerdmeester, A. Thomas, & I. L. McLelland, eds. *Usability Evaluation in Industry*. London, UK: Taylor and Francis.
- Bruce Bukovics. 2010. *Pro WF: Windows Workflow in .NET 4*, Apress.
- Michael Buhrmester, Tracy Kwang, and Samuel D. Gosling. 2011. Amazon's Mechanical Turk: A New Source of Inexpensive, Yet High-Quality, Data? *Perspectives on Psychological Science* 6, 1 (2011), 3–5. DOI:https://doi.org/10.1177/1745691610393980
- Peter Buneman, Mary Fernandez, and Dan Suciu. 2000. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *The VLDB Journal—The International Journal on Very Large Data Bases* 9, 1 (2000), 76–110.
- Margaret Burnett, Curtis Cook, and Gregg Rothermel. 2004. End-User Software Engineering. *Communications of the ACM* 47, 9 (2004), 53–58.
- Margaret Burnett and Todd Kulesza. 2015. End-User Development in Internet of Things: We the People. In *Proceedings of the Workshop on End User Development in the Internet of Things Era*. Seoul, Korea: ACM.
- Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, and Chiara Francalanci. 2011. DashMash: a Mashup Environment for End User Development. In *Web engineering*. Springer, 152–166.
- Darren Carlson, Matthias Mögerle, Max Pagel, Shivam Verma, and David S. Rosenblum. 2015. A Visual Design Toolset for Drag-and-drop Smart Space Configuration. In *Proceedings of the 5th International Conference on Internet of Things*. Seoul, Korea.
- James Cheney. 2008. FLUX: functional updates for XML. In *ACM Sigplan Notices*. ACM, 3–14.
- James Clark. 1999. *XSL Transformations (XSLT)*, W3C.
- Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. 1979. *Graph-grammars and Their Application to Computer Science and Biology: International Workshop*, Springer.
- Sophie Cluet and Jérôme Siméon. 2000. YATL: a Functional and Declarative Language for XML. (2000).
- Joelle Coutaz and James L. Crowley. 2016. A First-Person Experience with End-User Development for Smart Homes. *IEEE Pervasive Computing* 15, 2 (2016), 26–39.
- György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. 2002. VIATRA-Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*. Edinburgh, United Kingdom: IEEE, 267–270.
- Allen Cypher and Daniel Conrad Halbert. 1993. *Watch What I Do: Programming by Demonstration*, MIT press.
- Krzysztof Czarnecki and Simon Helsen. 2003. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. USA, 1–17.
- Jose Danado and Fabio Paternò. 2014. Puzzle: A mobile application development environment using a jigsaw metaphor. *Journal of Visual Languages & Computing* 25, 4 (2014), 297–315.
- Jose Danado and Fabio Paternò. 2012. Puzzle: A Visual-Based Environment for End User Development in Touch-Based Mobile Phones. In *Human-centered software engineering*. Springer, 199–216.
- Wanda P. Dann, Stephen Cooper, and Randy Pausch. 2011. *Learning to Program with Alice*, Pearson.

- Juan De Lara and Hans Vangheluwe. 2002. ATOM3: A Tool for Multi-formalism and Meta-modelling. In *Fundamental Approaches to Software Engineering*. Springer, 174–188.
- Markus von Detten, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Dietrich Travkin, and Stephan Hildebrandt. 2012. Story Diagrams—Syntax and Semantics. *Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Tech. Rep. tr-ri-12-324* (2012).
- Zoé Drey and Charles Consel. 2012. Taxonomy-Driven Prototyping of Home Automation Applications: A Novice-Programmer Visual Language and its Evaluation. *Journal of Visual Languages & Computing* 23, 6 (2012), 311–326. DOI:<https://doi.org/http://dx.doi.org/10.1016/j.jvlc.2012.07.002>
- Carsten Eickhoff and Arjen P. de Vries. 2013. Increasing Cheat Robustness of Crowdsourcing Tasks. *Information retrieval* 16, 2 (2013), 121–137.
- G. Fischer, E. Giaccardi, Y. Ye, A.G. Sutcliffe, and N. Mehandjiev. 2004. Meta-design: A Manifesto for End-user Development. *Commun. ACM* 47, 9 (September 2004), 33–37. DOI:<https://doi.org/10.1145/1015864.1015884>
- Martin Fowler. 2004. *UML Distilled: A Brief Guide to the Standard Object Modeling Language* 3rd ed., Addison-Wesley Professional.
- Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. 2003. A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard. In *MetaModelling for MDA Workshop*. Citeseer, 41.
- Giuseppe Ghiani, Fabio Paternò, and Lucio Davide Spano. 2009. Cicero Designer: An Environment for End-User Development of Multi-Device Museum Guides. In *End-User Development*. Springer, 265–274.
- Herbert Götter. 1992. Diagram editors= graphs+ attributes+ graph grammars. *International Journal of Man-Machine Studies* 37, 4 (1992), 481–502.
- Thomas R.G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A “Cognitive Dimensions” Framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.
- F. Hang and L. Zhao. 2015. Supporting End-User Service Composition: A Systematic Review of Current Activities and Tools. In *Web Services (ICWS), 2015 IEEE International Conference on*. 479–486. DOI:<https://doi.org/10.1109/ICWS.2015.70>
- Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. 2011. GRoundTram: An Integrated Framework for Developing Well-behaved Bidirectional Model Transformations. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 480–483.
- Dat Dac Hoang, Hye-Young Paik, and Anne H.H. Ngu. 2010. Spreadsheet as a Generic Purpose Mashup Development Environment. In Paul P. Maglio, Mathias Weske, Jian Yang, & Marcelo Fantinato, eds. *Service-Oriented Computing: 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 273–287.
- Jan Humble et al. 2003. “Playing with the Bits” User-Configuration of Ubiquitous Domestic Environments. In Anind K. Dey, Albrecht Schmidt, & Joseph F. McCarthy, eds. *UbiComp 2003: Ubiquitous Computing: 5th International Conference, Seattle, WA, USA, October 12-15, 2003. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 256–263.
- Jeffrey C. F. Ho. 2015. Configuring Devices as End-User Programming in the Era of Internet of Things. In *Proceedings of the Workshop on End User Development in the Internet of Things Era*. Seoul, Korea: ACM.
- Diane Jordan et al. 2007. Web services business process execution language version 2.0. *OASIS standard* 11, 120 (2007), 5.
- Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. 2006. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 719–720.
- Frédéric Jouault and Ivan Kurtev. 2006. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*. Montego Bay, Jamaica: Springer, 128–138.
- Robin H. Kay. 1993. A Practical Research Tool for Assessing Ability to Use Computers: The Computer Ability Survey (CAS). *Journal of Research on Computing in Education* 26, 1 (1993), 16–27.
- Andrew J. Ko et al. 2011. The State of the Art in End-User Software Engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 21.
- Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 199–206.
- Woralak Kongdenfha, Boualem Benatallah, Julien Vayssière, Régis Saint-Paul, and Fabio Casati. 2009. Rapid Development of Spreadsheet-based Web Mashups. In *Proceedings of the 18th International Conference on World Wide Web. WWW '09*. New York, NY, USA: ACM, 851–860. DOI:<https://doi.org/10.1145/1526709.1526824>
- Gerd Kortuem, Arosha K. Bandara, Nadia Smith, Mike Richards, and Marian Petre. 2013. Educating the Internet-of-Things generation. *Computer* 46, 2 (2013), 53–61.

- Gerd Kortuem, Fahim Kawsar, Daniel Fitton, and Vasughi Sundramoorthy. 2010. Smart Objects as Building Blocks for the Internet of Things. *Internet Computing, IEEE* 14, 1 (2010), 44–51.
- Angel Lagares Lemos, Moshe Chai Barukh, and Boualem Benatallah. 2013. DataSheets: A Spreadsheet-Based Data-Flow Language. In Samik Basu, Cesare Pautasso, Liang Zhang, & Xiang Fu, eds. *Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 616–623.
- Henry Lieberman. 2001. *Your Wish is My Command: Programming By Example*, Morgan Kaufmann.
- Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-User Development: An Emerging Paradigm. In *End User Development*. Springer, 1–8.
- James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces*. IUI '09. New York, NY, USA: ACM, 97–106. DOI:<https://doi.org/10.1145/1502650.1502667>
- Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. New York, NY, USA: ACM, 943–946. DOI:<https://doi.org/10.1145/1240624.1240767>
- Kun Ma, Bo Yang, and Ajith Abraham. 2012. A Template-based Model Transformation Approach for Deriving Multi-Tenant SAAS Applications. *Acta Polytechnica Hungarica* 9, 2 (2012), 25–41.
- John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 16.
- David Merrill, Jeevan Kalanithi, and Pattie Maes. 2007. Siftables: Towards Sensor Network User Interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*. ACM, 75–78.
- Anders I. Mørch, Gunnar Stevens, Markus Won, Markus Klann, Yvonne Dittrich, and Volker Wulf. 2004. Component-Based Technologies for End-User Development. *Communications of the ACM* 47, 9 (2004), 59–62.
- Mark W. Newman, Ame Elliott, and Trevor F. Smith. 2008. Providing an Integrated User Experience of Networked Media, Devices, and Services Through End-User Composition. In *Proceedings of the 6th International Conference on Pervasive Computing*. Pervasive '08. Berlin, Heidelberg: Springer-Verlag, 213–227. DOI:https://doi.org/10.1007/978-3-540-79576-6_13
- Ulrich Nickel, Jörg Niere, and Albert Zündorf. 2000. The FUJABA Environment. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM, 742–745.
- Željko Obrenović and Dragan Gašević. 2008. End-User Service Computing: Spreadsheets as a Service Composition Tool. *IEEE Transactions on Services Computing* 1, 4 (2008), 229–242. DOI:<https://doi.org/http://doi.ieeecomputersociety.org/10.1109/TSC.2008.16>
- Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014. Biflux: A Bidirectional Functional Update Language for XML. In *Proceedings of 16th International Symposium on Principles and Practice of Declarative Programming*. Canterbury, United Kingdom: ACM, 147–158.
- Gabriele Paolacci, Jesse Chandler, and Panagiotis G. Ipeirotis. 2010. Running Experiments on Amazon Mechanical Turk. *Judgment and Decision Making* 5, 5 (2010), 411–419.
- Andy Schürr, Andreas J. Winter, and Albert Zündorf. 1995. Graph Grammar Engineering with PROGRES. In *Software Engineering — ESEC '95*. Springer, 219–234.
- Helen Sharp, Yvonne Rogers, and Jenny Preece. 2007. *Interaction Design: Beyond Human-Computer Interaction* 2nd ed., Wiley.
- João P. Sousa et al. 2011. TeC: end-user development of software systems for smart spaces. *International Journal of Space-Based and Situated Computing* 1, 4 (2011), 257–269.
- Gabriele Taentzer. 2004. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance*. Springer, 446–453.
- Dániel Varró and András Pataricza. 2003. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Software and Systems Modeling* 2, 3 (2003), 187–210.
- Guiling Wang, Shaohua Yang, and Yanbo Han. 2009. Mashroom: End-user Mashup Programming Using Nested Tables. In *Proceedings of the 18th International Conference on World Wide Web*. WWW '09. New York, NY, USA: ACM, 861–870. DOI:<https://doi.org/10.1145/1526709.1526825>