Yogurt: a programming language for the Internet of Things (IoT)

submitted in partial fulfilment for the degree of master of science

Ivan H. Gorbanov
12403555

master information studies
Information Systems
faculty of science
university of amsterdam

2019-07-19

	UvA Examiner	Academic Supervisor	Industry Supervisor	Industry Supervisor
Title, Name	Dr Frank Nack	Dr Pablo Cesar	Jack Jansen	Steven Pemberton
Affiliation	University of Amsterdam	CWI & Delft University of Technology	CWI	CWI
Email	nack@uva.nl	p.s.cesar@cwi.nl	jack.jansen@cwi.nl	steven.pemberton@cwi.nl

Universiteit van Amsterdam

Yogurt: a programming language for the Internet of Things (IoT)

Ivan Gorbanov
Universiteit van Amsterdam
Information Systems
Centrum Wiskunde Informatica
Distributed and Interactive Systems
ivan.gorbanov@student.uva.nl

ABSTRACT

As the Internet of things moves from the hands of professionals and academics into the those of the general consumer, it becomes increasingly important to provide the appropriate tools for interaction for that particular target audience. The available solutions appear to be either powerful but too complex, or easy to use, but lacking substantial expressiveness. This paper presents a programming model and subsequent language which aim to address this disparity by providing the right level of abstraction analogous with the real world for simplicity. In addition, they incorporate several concepts from established high-level programming paradigms in order to accommodate for a large number of use cases in this heterogeneous domain. The proposed designed is evaluated through user testing which shows that the presented solution is expressive enough to facilitate the common use cases, which have become expected by IoT users. The results also show that the language does not present a steep learning curve as the distance between the problem which the programmer is trying to solve and how to solve is kept very small.

KEYWORDS

Internet of Things (IoT), Programming Language Design, User Interaction

1 INTRODUCTION

Since its beginnings, the Internet of things (IoT) has slowly moved from the realm of science fiction into the mainstream. Today there are over 8 billion linked, smart devices and that number is predicted to exceed 20 billion by 2025 [20]. This steep increase could be partly attributed to the increasing adoption of the technology by the general public. Today, consumers have access to a plethora of small, interconnected devices which allow them to automate both their living and working environments for better convenience and efficiency [23]. As this technology moves into the hands of the consumer, it becomes increasingly important to provide the average user with the right tools to interact with it in order to best fulfill his or her needs.

Traditionally these devices are programmed through a variety of high-level languages which allow the programmer full control over their system. This, however, requires an in-depth knowledge of computer programming which often takes years of training and practice to develop and understand completely. In addition, the decentralized nature of the development of these technologies has led to the incorporation of a large variety of communication protocols, formal languages and environments [35]. Therefore, managing a system which consists of just a few devices from various manufacturers, often requires knowledge of a mixture of programming environments and paradigms. This in turn further increases the barrier to adoption due to the steep learning curve associated with this knowledge.

Both the consumer industry and academia have tried to address this problem by introducing systems which act as a central control point for the customer's entire IoT ecosystem [2, 35]. Through these, the user can manage the variety of devices from one environment without worrying about any discrepancy in protocols or operating systems between them. One such system is *Igor* [13]. Developed by Jansen and Pemberton, Igor is an architecture designed to fulfil the role of a digital butler for the smart home or office.

The latest version of Igor[1] already offers some powerful capabilities such as access control, hiding of data-format differences, integration of data into homogeneous collections and automatic updates of both devices and data store [13]. However, the mechanism used to configure the system is an interim solution consisting of a basic web interface leaving much room for improvement in both usability and expressiveness. This paper aims to extend the work of Jansen and Pemberton [13] by presenting a new programming model and a subsequent programming language called Yogurt, which facilitate the users' interaction with Igor.

Moreover, the research seeks to address the disparity between expressiveness and usability that often exists in the programming facilities of IoT systems. As these programming environments are usually optimised for the latter, they often lack the expressive power of traditional programming languages and therefore limit the user's capabilities in order to reduce the possibility of errors in the program [24]. On the other hand, high-level languages don't allow the programmer to think about the problem at hand in the way it appears in the real world. Rather, they require translating it into algorithms logical to a CPU. This gap in the cognition, significantly increases the complexity of the task. Consequently this paper tries to answer the following research question:

To what extent can a new declarative programming language, designed for the Internet of Things, achieve high expressiveness while keeping the gap small between what the programmer is trying to achieve and how to achieve it?

The following sub questions help break down the main research question into more manageable portions:

- Do programs written in this new language have a simple connection between cause and effect?
- Is the programming language easy to learn?
- Does the program comply with the criteria for best practices in programming language design?

In order to find an answer, a programming language designed following the standard Software Development Life Cycle methodology [6], since a programming language is a piece of software in

[1] Latest Igor Version: 0.99.1

itself. [8]. The proposed design is then evaluated through performing user tests, following the Discount Method for programming language evaluation methodology [17].

While the evaluation reveals aspects of the design which can benefit from some improvement, the suggested model achieves simplicity by reducing the gap between what the programmer aims to achieve and the steps required to reach this goal. It accomplishes this by providing the right level of abstraction which is analogous with the real world in order to make it easier for the end-user to conceptualize and therefore understand [8]. In addition, the language described in this paper, uses encapsulation and inheritance mechanisms in order to allow even complex systems to be represented with the proposed model by thinking about the subsystems and components that make them up, therefore providing a high degree of expressive power. While this language has been developed for Igor, a large portion of the solution is architecture independent and therefore could be applied to a wide variety of such systems with minimum customization.

The rest of this paper is structured as follows. Section 2 provides a more detailed overview of the Igor architecture and why it has been chosen as the basis for the development of this new language. In addition the need for a new programming mechanism is identified. The related work section then examines the current solutions available for this need. Section 4 presents the elicitation process and subsequent requirements for the development of the language. The designed model is then explained in section 5 and a possible language is presented in section 6. The work is then evaluated in section 7 and conclusions are drawn.

2 IGOR

The following section gives a more detailed description of Igor; this is important for this research as it will reveal certain requirements and constraints for the design of the language. Throughout this and the subsequent sections, the requirements IDs (Section 4.2.1) will be used as references to their motivation. As introduced above, Igor is an architecture designed to give unified access to IoT devices regardless of their varying interfaces or data formats. It achieves this by placing a functional layer of plug-ins around an XML repository which stores all of the data generated by the devices. The database is updated bidirectionally, meaning that every time a device's state changes, the corresponding values will change in the data store. Vice versa, if the values in the database are altered, the device associated with those values will also change its state to reflect the new information in the repository. (See requirement R3.2) This is achieved by the plug-ins in the functional layer, which are responsible for contacting each individual device, listening for any state changes (R1.6), converting the device data to and from XML and updating the device if there is a change in the database [13].

This system has several advantages, other than the ones already mentioned, which make it a good candidate for this research. Firstly, Igor is state-based (R1.5, R3.1), which allows for better abstraction of information in comparison to event-based systems such as the ones described in the Related Work section [37]. In addition, it utilizes widely adopted web standards such as REST, XML and DOM which allow the framework specific portion of the proposed programming model to be more easily adaptable to a larger number of different architectures without the need to be rewritten completely [13]. Finally, the project is open source and therefore all of the code is easily accessible for implementation and testing purposes.

Currently Igor is configured through a basic text-based web interface, which as mentioned above is an interim solution used more for proof of concept than being designed with the programmer in mind. The user can specify the device's behavior by creating "Actions". These consist of selecting a trigger, which in most cases is a change in the value at the specified path in the data store, an optional condition and an action. The latter of which consists of a REST method (GET/PUT/POST/DELETE), the URL for the entry that is to be changed and the value that it should be changed to [13]. While the current implementation allows the user to tackle a large variety of use cases, programming complex "Actions" involving several triggers and actions quickly becomes difficult to manage. In addition, the dependencies of the different programs are not made very clear by the interface and it can be difficult to spot any conflicts. These are all problems that the new programming model has to tackle, without losing any of the functionality of the old system.

3 RELATED WORK

Before starting on the design of the new model, current solutions available to users for programming their IoT devices, are analysed. The goal of this literature review is to discover to what extent an existing solution is appropriate for Igor. The analysis is subdivided into the three areas where these solutions come from in order to allow for easier comparison as concepts from the same field will have many common features.

3.1 Traditional Programming Languages

High-level programming languages have been the default way in which people have bent technology to their will since their introduction in the 1950s. In the last seven decades, much research has gone into the area resulting in countless models and notations fitting a variety of paradigms for programming computers and machines [36]. Due to the high number of languages and the time limitations associated with this project, it would be unrealistic to evaluate every single one of them for the purposes of this research, therefore only the more popular languages which suit the programming paradigm of the problem domain will be examined. While there are other popular paradigms than the once listed here, such as functional programming, they have been omitted from this paper, as this research failed to identify any significant applications in the IoT domain.

3.1.1 Imperative Programming. Imperative programming refers to writing programs based on updating variables in storage. It is one of the oldest paradigms and it still retains massive popularity today [36]. This is partially attributed to inertia, as programmers have been using the paradigm for so long that they are familiar with it, but it has also proven itself to be a good way to model real world systems. Programming languages Pascal [8], Ada [10] and Python [21] are all good examples of the imperative style of programming. They are powerful enough to tackle most computation tasks. Furthermore, in the case of the first two in particular, they run very fast as they model machine architectures quite closely and therefore can be implemented very efficiently [36]. However, when writing imperative programs, developers have to think in terms of computer algorithms rather than real world applications

as imperative programs require a very precise list of instructions of how to achieve the task at hand [8]. This is therefore very dependent on the architecture where the program is being implemented and can be difficult to generalize for a motley collection of devices such as the Internet of Things [5]. This therefore introduces a big barrier to adoption which is not appropriate for a technology being assimilated by the general public. A sub-paradigm of imperative programming is object-oriented programming. While the two are related, the latter is examined in the next subsection for clearer separation of concepts.

3.1.2 Object-Oriented Programming. One of the major issues with programming ecosystems with a lot of these devices is complexity. As this technology finds its way into an increasing number of appliances, writing programs that incorporate multiple instances can quickly become difficult to follow. Object-oriented programming is a paradigm created to deal with these issues by offering an inheritance mechanism [8]. What this means is that a complex system can be represented as a collection of simpler ones. These are desired qualities as users may have to deal with such use cases. One particular example is programming an elevator which can be seen as a collection of buttons for the floors, scale that checks the load on the elevator, lights inside it, doors, door sensors and a motor that makes the elevator move between floors. Programming all of these together can easily result in code which is difficult to read, debug and more prone to mistakes. However, if each one of those systems is programmed separately and then the elevator object encapsulates them and therefore inherits their properties, the program becomes a lot more manageable while producing the desired result. Furthermore, encapsulation allows to hide some the data of some objects from other. This quality is also highly desirable as the user may want to restrict the access to some more sensitive devices such as their fire alarm.

Smalltalk [11], C++ [30] and the already mentioned Python are all languages which fit this paradigm. However, they are procedural in nature, requiring the programmer to understand memory allocation and management and every step of the process involved in executing the program. As stated above this is not a desirable quality and therefore such languages are inappropriate for Igor [8].

3.1.3 Declarative Programming. Declarative programming refers to a style of programming where the programmer specifies the intended outcome of the program without necessarily stating each step required to accomplish it, in contrast to imperative programming [19]. This programming style seems very appropriate to IoT applications as users should not be burdened with the details of memory management and control flow but rather focus on the desired behavior of their devices. As a result declarative programs are easier to understand making them more appropriate for users without a background in computer science [1]. This feature of the paradigm also allows for context-independence, which means that the language is less dependant on any framework or architecture as the result of the program should be the same regardless, while the step by step details are left to each individual system to decide for itself. Finally, declarative programs are easier to analyze for correctness as the logic can be examined simply and statically without having to compile the program first [1].

Prolog [4], Godel [12] and SQL [16] are all notable examples of high-level programming languages which adhere to the principles of the declarative paradigm. Prolog and Godel can be further classified as logic programming languages, where the program is a set of rules which the system needs to follow [1]. This is a way in which IoT systems can also be represented. For example, a program that turns the lights inside the house on, when a user is present can be written as the rule: `lights-on ← user-home` . However, complex programs with many rules become harder to follow and therefore are more prone to misinterpretation. (R2.3) In addition the arithmetic operations are cumbersome and they might be required to convert units in order to match sensor outputs to actuator inputs [1]. (R1.9)

SQL (Structured Query Language) is a domain specific language designed to be used in the creation and management of relational databases. One of its main advantages over other languages made for the same purpose comes from its declarative nature. SQL queries do not need to contain specific instructions on how to reach a particular record through indexing or other pointer mechanisms. One simply has to specify what record they need [16]. In addition, the language uses very readable keywords as commands, making it easy to follow and understand. While these features are both desirable for the new model proposed in this paper, SQL comes short in some of the arithmetic and control flow operations, therefore making it unsuitable for the problem domain as these will most likely be required [7].

3.2 Event-Based Frameworks

Recent years have seen the rise of event-based trigger-action frameworks designed to allow users to create automated tasks [18]. These frameworks have gained popularity within the research community due to their simplicity [24]. The models allow the specification of the device's behavior as a set of if-this-then-that rules. This approach is very much declarative in nature as the users only have to concern themselves with the logic followed by the devices rather than control [19]. This further supports the conclusion from the previous subsection, that the declarative paradigm is the preferred choice for the language presented in this paper.

Two of the more popular representatives of this approach are IFTTT [2] and Zapier [3] . Both platforms allow users to automate tasks across multiple devices and online services. This is achieved by specifying an event as a trigger (ex. alarm clock goes off) and a desired action (ex. lights come on). Each pair of trigger and action is known as a recipe or a zap in IFTTT and Zapier respectively [24]. While both frameworks have proven very effective for simple tasks, they are both limited to one trigger per recipe/zap and while Zapier allows for the inclusion of conditions in the form of filters as well as several actions per zap, both frameworks fail to allow their users to specify more complex tasks with multiple triggers and actions [24].

Architectures such as homeBlox [35] aim to address these issues by providing a selection of common logical operators (and, or) and removing any limits in regard to number of actions or triggers. However, homeBlox's programming model does not provide a facility to deal with different users/home occupants and have the system react depending on who is present in its environment [35].

One of the main advantages of these frameworks collectively is that they are all presented to the user in a GUI which helps

[2] https://ifttt.com/
[3] https://zapier.com/

eliminate some of the intimidation that textual representations might pose to the novice user. It also eliminates the possibility of bugs in the programs caused by typing errors [34]. The visual programming approach has been utilized by other IoT programming tools such as NodeRed [4]. The JavaScript based framework is much more powerful than the other solutions examined in this section, however the abstractions are at a much lower level [26]. Therefore, the distance between implementation and desired goal is too great for the approach to be easily adoptable. Nevertheless the advantages of graphical representations make a GUI for the model a worth while implementation to explore.

3.3 Commercial Platforms

As mentioned in the introduction, there are a variety of commercial solutions available, which like Igor, act as control centers for IoT ecosystems, eliminating any concern regarding the heterogeneous nature of IoT devices. Prime examples of such systems are Alexa [5], WinkHub [6] and Homey [2]. Alexa and WinkHub both provide very attractive graphical interfaces. However, they utilize IFTTT and therefore inherit all of its shortcomings, detailed in the previous subsection. Homey has a complete graphical programming environment which allows the users to build scenarios in which the systems performs tasks automatically. These are referred to as “Flows”. Individual flows follow the when-and-then principle, which is similar to if-this-than-that: an event is selected to act as a trigger and an action is executed. The framework expands on the basic IFTTT by using logical operators which allow the combination of multiple triggers and actions. Conditions are also possible in order for the “Flows” to be dependent on the current environment. The abstractions used are a lot closer to the objects they represent in the real world therefore making it easier to understand than the abstract elements used in NodeRed. These objects however are predetermined and therefore the end-user is limited in their capabilities by the imagination of the programmer of the system. It also means that the system requires continuous updates of cards as new technologies become available, making it high maintenance and reducing expressiveness [2].

3.4 State of the Art Summary

It is clear that, while there are many solutions available, none seem to be entirely appropriate for Igor. In addition, the literature review further identifies the gap between programming models which provide great expressiveness at the cost of a high learning curve and those which are very user friendly but lack applicability in more complex use cases. Nevertheless, certain aspects of the presented state of the art appear very effective in tackling some of the challenges. The declarative programming paradigm offers the context-independence which appears to be crucial in such a heterogeneous application domain (R1.4). The abstraction from memory management and specific imperative instructions helps reduce the learning curve by allowing the programmer to think in terms of what they want to achieve rather than how to, which is more appropriate for a technology used by the general consumer. The inheritance mechanism from object-oriented programming is a good way to achieve high expressiveness while keeping complexity low (R1.3). Furthermore the encapsulation mechanism offered by the paradigm, allows for easier to read code as the data and methods that act upon it are kept together in the same block. Additionally the feature allows for higher security when dealing with system critical devices (R1.2). The commercial environments show that providing programmers with abstractions which are analogous to the real world objects they represent (R1.1), greatly improves their usability by programmer who are not as experienced. IFTTT and Zapier also make a good case for keywords (If this, then that) and concepts which resemble the way humans would think about the problem and more importantly its solution (R2.2).

4 REQUIREMENTS ELICITATION

Most design methodologies start with answering the question "What is it that we are designing?" or in other words what are the requirements for the final output of the process [28]. This study is no exception. The first step of the elicitation process is making sure that the language is expressive enough to facilitate common use cases. These are drawn from academic literature, as well as from an analysis of the applications of the available commercial platforms. In addition, section 3.4 outlines some non-functional requirements elicited through the analysis of the related work in the field. Finally, using Igor as the case study for the development of this model brings some constraints to the design. However, some of these may not stand if the model is to be implemented in a different framework. Instead others may be introduced.

While this list is meant to cover all of the bases, it is by no means exhaustive. The decentralised nature of the development of this technology means that it is applied in many different ways and one complete list of its desired capabilities and applications is non-existent. In addition, new applications keep appearing constantly and therefore covering all of them in the scope of this study is unrealistic due to the time constraints associated with it. Therefore the requirements listed in this section are meant to be as broad as possible in order for the language to apply to as many use cases as possible and while everything is initially motivated by literature or existing cases studies, the final selection of what was important and what not was left to the discretion of the research team.

4.1 Use Cases

Within the boundaries of the consumer market, IoT technology finds some of its most popular applications in the area of home automation [15]. The main objectives being increased efficiency in the usage of utilities and resources [31], higher comfort through an adaptive environment [14], enhanced security for the occupants and fault detection to minimise any adverse effects that an overlooked malfunction may have [25].

The following subsection describes the behaviour of a house which is fitted with all of the required sensors and actuators as well as an Igor system to control them. It outlines the context in which the user finds themselves in, as well as the behaviour of their IoT system in that context. The use cases are presented in the form of a scenario, as according to Ramirez et al. [27] it is a methodology which helps produce more interesting research with a broader outlook. The story line is chosen in order to illustrate at least one instance of the above mentioned applications. In addition, it depicts a mixture of different programming challenges so that the

[4] NodeRed website: https://nodered.org/
[5] Alexa website: https://www.amazon.com/Amazon-Echo-And-Alexa-Devices/b?ie=UTF8node=9818047011
[6] Wink Website: https://www.wink.com/products/wink-hub/

use cases can be most exhaustive. The situations from the scenario will be used throughout the rest of this paper as example programs to demonstrate the language capabilities and syntax. Furthermore the scenario is representative of the use cases found on current popular commercial platforms, [2, 22] therefore ensuring that all of the capabilities which current users have already come to expect from the technology are there.

4.1.1 Scenario. Jack is a 30 year old light technician who lives in Amsterdam and works at the city theater. He has a wife called Jill, who is a nurse. Their home is fully equipped with an Igor system and all the latest sensors and smart appliances. When Jack comes back home from work around 18:00, he unlocks his door with a key card. As soon as the door opens the lights come on, as it is winter and the brightness sensor outside the door tells Igor that it is dark. The LED lighting shines in a warm yellow color as it is cold outside and Igor tries to create a warmer atmosphere inside. As Jack walks in the house, the door locks behind him and his favorite playlist starts playing through the stereo. As Igor knows that Jack is alone, the volume is set quite high (R1.8). An hour later Jill comes back from work. The music reduces in volume so that Jack and Jill can have a conversation more easily, it also switches the playlist to something they both like. Around 23:00 the lighting in the house dims automatically prompting both inhabitants that it is time for bed. As Jack goes to bed, the pressure sensor in his mattress turns on the light on his night stand. When Jill joins him, the main light goes off and her nightstand also illuminates. The smart wall sockets switch off and turn the power off to all unnecessary appliances such as TV, toaster, washing machine etc. to conserve energy overnight while they are not being used. The thermostat lowers the temperature until half an hour before Jack wakes up at 7:00 (R1.7). When that time comes, Jack's morning playlist starts playing through the speakers and the lights come on and slowly start increasing in intensity. As soon as Jack gets out of bed, the lights in the bedroom dim again and the music stops in order not to disturb Jill, as it is a Wednesday and her agenda shows she has the late shift at the hospital. When Jack goes in his kitchen the lights come on, and his morning playlist starts playing through the speakers there. He finds a cappuccino waiting for him at his coffee machine. He drinks it and leaves for work. The door locks 10 seconds after he has left.

4.2 Requirements

In order to program an IoT environment to behave according to the scenario described above using Igor and the required devices, Yogurt will have to meet the following requirements. Each requirement is given a unique ID, which are used as references to its motivation throughout this paper for the convenience of the reader.

4.2.1 Functional Requirements.

- R1.1 The programming model must provide an abstraction for IoT devices
- R1.2 The programming model must provide an encapsulation mechanism
- R1.3 The programming model must provide an inheritance mechanism
- R1.4 The programming language must comply to the principles of the declarative programming paradigm
- R1.5 The programming model must be able to represent the state of devices
- R1.6 The programming model must be able to facilitate state changes as triggers for other events/state changes
- R1.7 The programming language must allow the user to schedule events based on time
- R1.8 The programming model should provide a facility to deal with conditions
- R1.9 The programming language must allow the programmer to convert the units of a sensor output to the units of an actuator input through mathematical operations

4.2.2 Non-Functional Requirements.

- R2.1 The programming language must comply with the criteria for good programming language design (Section 3.4)
- R2.2 The keywords used in the language should be unambiguous
- R2.3 The language model should enable the programmer to code in such a way that all dependencies can be tracked easily

4.2.3 Constraints.

- R3.1 The programming model must be state based.
- R3.2 The programming model must allow making state changes triggers

5 MODEL

Programming languages are software tools which describe the behaviour of a system by modeling it using abstractions [29]. Therefore a crucial step of the design of a new programming language is deciding what is the best way to model the domain which the programming language is trying to facilitate.

5.1 Abstractions

5.1.1 Actor. As already mentioned, the Internet of Things is made of devices. Therefore, the top level abstraction, called Actor, in the presented model, fulfils this purpose. Actors are any devices (sensors, actuators, etc.) that are part of the user's IoT ecosystem. For example an LED light, a smart TV or even just a simple light switch. The common feature between actors is that they change their own state. (Section 5.1.3) All the other abstractions are children of an actor. This should make the model easier to conceptualise as it is analogous to the real world. More specifically, the actor abstraction is similar to a class definition in an object-oriented programming language such as Smalltalk [8]. It describes the behaviour of a particular type of device. To elaborate, if a user has two presence detection sensors (PIR) and four lights, provided that the sensors behave the same way, he or she will need to create one actor to describe their behaviour and another to describe the lights' behaviour. The user will then have to construct two instances of the PIR actor and four of the lights actor and connect them as appropriate. This feature of the model allows the reuse of blocks of code making writing programs for large ecosystems, with a lot of instances of the same device, a lot more efficient. Furthermore, it gives Igor the potential to be adoptable by novice users, as the system can come pre-loaded with a library of standard device behaviours. Therefore, leaving only the connecting of physical devices to behaviours, to the user. This is similar to the set up of the Homey programming environment discussed in section 2.3 and it will bring the same advantages.

5.1.2 State. As mentioned in section 3.1. Igor is a state based system. In this context, a device's state refers to every unique combination of values of its properties. For example, an LED strip typically has five state attributes: three values indicate its color (one each for R,G,B), one indicating its intensity and the last one indicating the count of illuminated diodes on the strip. Changing the value of any of these five attributes results in transitioning the LED strip from one state to another.

Hence the first and perhaps most important child abstraction of Actor is state. An instance of it could simply consist of one value for one attribute such a light switch, where the state will be simply represented by one Boolean value depending on whether or not the switch is active or not. Alternatively it can be made up of several values for more complex devices such as the LED strip example in the paragraph above.

5.1.3 Action. Since the device's abstraction is called an "Actor", it makes sense to call its activity "Action". In the presented model an "Action" is nothing more than changing the state of an actor by altering one or more values associated with its state parameters. The realisation that performing an action is nothing more than changing values in the data store is an important one here, as this simplicity is not only essential to the model but it is also what removes the need for APIs to each individual device. Each actor can have the ability to perform multiple actions, however an Action can only change the state of an actor that it belongs to. For example, an LED strip could have an action which changes its color, but it cannot contain an action which changes the temperature of the thermostat in the user's house. This is done for security reasons [36] as well as to force the programmer to keep all of the dependencies of a particular state in one place, making any conflicts easier to identify. An action consists of two child abstractions: trigger and guard. Each is detailed below.

Trigger. A trigger is designed to represent the event which sets off an action. In a state based system such as Igor, this is simply a listener for a change in a state attribute value of another actor. This can be illustrated with another use case form the scenario in section 4.1.1. In it, Jack's LED lights come on when the PIR sensor detects Jack's presence. Therefore the state attribute "presence" of the actor "PIR" is the trigger of the "turn on" action, belonging to the "LED light" actor. However, this is not enough as Jack will not want his lights turning on while it is still light outside and therefore there is plenty of natural light in the room. Therefore the model should provide a mechanism to describe the conditions which have to be satisfied for an action to execute.

Guard. A guard is the abstraction which facilitates the conditions upon which an action is executed. It is an expression which evaluates to either true or false. If the guard is true, then the subsequent code gets executed. In the case of the model provided these conditions simply describe a state of the user's IoT ecosystem or more specifically the state attributes which matter to that action. So in order to achieve the desired behaviour, Jack will need to not only program the "presence" trigger of the action but also set up a guard which checks if the state of the dusk sensor shows that it is night and the sun has set. This guard will consist of an expression which only evaluates to true when the value of the actor "dusk sensor"'s "night" state is true.

Some actions may have different results depending on the conditions which are satisfied by the rest of the environment. Therefore several different guards can be nested under one trigger, with the state update expressions underneath them. Each trigger only listens to one state attribute value changing, once the change occurs it then checks the guards underneath it. This design is motivated by computational efficiency as only the relevant conditions will have to be checked once a trigger occurs, rather than running through all guards every time a value somewhere in the system changes [8]. While a strong case can be made for having allowing the guards to work as an implicit triggers, separating the two, allows for some more complex functionality in the language implementation of the model. (See section 7.2)

5.1.4 Timer. Section 4.1.1 describes several use cases where time is of importance. One type is where the system reacts to a specific time occurring, such as it becomes 23:00 on a given day or it is a Wednesday. The other type of use case is where an event occurs after certain time has elapsed following an action, such as the door locking 10 seconds after Jack has left the house. In order to facilitate this the model includes an abstraction, which can be classed as a special case of an actor, called timer. The latter use case is achieved as follows. Similar to an actor, the user can set up a trigger listening for a change of state of any other actor in the system to start it. The user also needs to pass an argument to specify the duration of the timer. The abstraction has its special "timer" state attribute which changes between zero and one every time the timer starts and the specified duration elapses. Action which are then dependent on this timer need to be set with triggers listening to this state value.

In the event that a user is only interested in a specific absolute time reference, such as 23:00 on Wednesday the 23rd of June, the user can set up a guard which checks the second state attribute of the timer abstraction, "absolute time". This attribute is configured to display the system time of the device which acts as the Igor server. The difference here being that this state attribute cannot be changed by an action but rather changes itself automatically based on the system time.

Similar to the normal Actor abstraction, a user can have multiple instances of the timer objects running independently of each other in order to satisfy different unrelated processes which require this functionality.

5.2 Model Features

Figure 1 shows a graphical representation of the model for the proposed language, which depicts how the above mentioned abstractions fit together and interact with each other. The one missing from the diagram is the timer, however that could be treated as a special actor case and therefore will follow the same logic. The model allows for several different features, described in detail in the following subsections. These features make the resulting programming language multi paradigm.

5.2.1 Declarative. A reoccurring conclusion in the related work and an enduring requirement is that the programming language be declarative. The model has been designed by constantly referencing this requirement and as such all the abstractions included are ones that describe a desired behavior and end state. Abstraction for memory and complex control structures have been omitted as

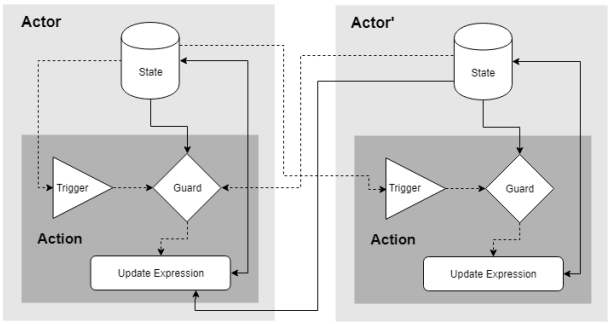

Figure 1: Programming Model

such constructs are not typical for the declarative programming paradigm [8].

As a result, the model does not suffer from the complexity that memory management and control flow bring to a programming task [19]. It also makes the solution context independent as the line "turn LED on" should turn the led light on regardless of weather the model is implemented in Igor or another system like Homey.

5.2.2 Object-Oriented. As already mentioned, the "Actor" abstraction is a concept borrowed from class definitions in object-oriented programming. This not only allows for the more efficient programs by reusing the same block of code over and over again as appropriate but it also allows for encapsulation and inheritance. Encapsulation refers to the bundling of data with the processes which operate on it. In the case of this model, it is the state attribute values with the actions that change them. This is done not only for convenience, but also in order to hide certain information from other actors for security reasons. This attribute becomes important for use cases where states are critical to the operation of the IoT environment [36]. One example is that the user doesn't want to allow other actors to be able to change the state of their fire alarm.

The other object oriented concept is that of inheritance. In the model described in this section, this is achieved by allowing an actor to be comprised of other actors. Therefore giving the top level actor access to all of the actions and states of its children. This helps represent complex systems, such as the elevator example from section 3.1.2, as a collection of smaller sub systems in turn making them easier to manage and write [9].

6 LANGUAGE

In order to turn the abstract model into something usable by the IoT community, this paper proposes a language based on it, called Yogurt. The following section presents the possible text representation through example code (Appendix B-F), the features of which are then explained.

6.1 Program Structure

It is a tradition in the programming community to introduce a new language by showing how to write the simplest program which outputs a "Hello World" message. While outputting a message is not really the core aim of this programming language, this paper tries to honor that tradition by showing a simple example of a program which facilitates the kitchen light being turned on and off with a switch. (Appendix C) The lines starting with a hashtag are comments to help the reader find their way through the code.

The example helps depict the three main parts of a Yogurt program:

6.1.1 Actor Definition. The first and most complex section of the program is the Actor Behaviour Definitions. This is the part in which the user can define the actors which make up their IoT system and more importantly specify how they want to behave. Each definition starts with the keyword `actor` followed by an identifier, brackets to specify any inputs and a colon. Inputs in this context refer to other actors which interact with the one being defined. In the given example the "light" actor receives input from the "switch", while the "switch" actor is passive and therefore does not receive any inputs. It is important that the name written in the inputs matches the ID given to that actor. Multiple inputs can be separated by commas. If an actor needs to receive inputs from multiple instances of another actor class, this can be done by giving each instance a unique ID and using that inside the actions section. (Appendix D)

The language uses indentation to signify a block of code rather than brackets. This is consistent throughout the whole program and it is a choice motivated by readability [8]. As per the model described in the previous section an actor contains states and actions. As seen in the provided example (Appendix C), each of these child abstractions has its own section in the textual representation started by the corresponding keyword.

States. This is the part of the program where the user can assign state attributes a local name to be used later in the actions. In the example we see that both types of devices have one state attribute each, the switch is on which can be true or false and the light is illuminated which is also a Boolean value. On the right hand of each statement is the expression which points to the location of that state attribute in the data store. As Igor is XML based that path is written using the XPath syntax. The user-assigned names of state attributes are only unique locally within that actor, in other words a programmer can have multiple different actors that contain the state "on".

Actions. This section contains the meat of the logic and behaviour of the program. The action is really comprised of three parts itself: a *trigger*, a *guard* and one or more *update expressions* which actually performs the action since, as already mentioned, an action is nothing more than updating a state attribute value. The action starts with a trigger. This is specified using the `on` command followed by the state which is being monitored for a change. The example found in Appendix B, uses the "on" state attribute of the switch as a trigger, therefore every time that attribute changes its value, the code following the statement will run. In the given example, there is no need for a guard so the update expression of the language then follows. This is constructed by specifying the state attribute of the actor which is being updated, followed by the update operator ("<-") and the value to which it is being updated. The language restricts this value to be one of the following types: string (surrounded by quotation marks), number (both integer or floating point is accepted), Boolean (True/False), another state attribute like in the example or a function call. The update expressions are executed atomically, in other words all together and there is no sequencing. This means that if the programmer has put the same

state attribute to be updated twice on the left hand side or used a value that gets updated by that block on the right hand side, the compiler will return an error.

Config. There is one additional part of the actor definition and that is the "config" section. It is an implementation detail, which deals with the connection between the underlying architecture, in this case Igor, and the abstractions presented to the user. As this section will have to interact with the device's plugins, it will have to be written in a different programming language, since this is functionality that is considered beyond the scope of this research and the presented programming model. This is because there are already many languages which do this well. Furthermore, in order for the implementation to be able to work with the biggest number of devices a more common language should be used such as C or Java. It is therefore not a section which has an effect on the presented model and therefore shall not be discussed in detail in this paper.

6.1.2 Actor Instantiation. The Actor behaviour definitions work similarly to class definitions in an object-oriented programming language. After specifying the behaviour the programmer has to create an instance of that actor and tie it to the address of the physical device it represents. In Yogurt, this can be done in one statement. The instance is given a unique identifier, followed by an equals sign, followed by the actor behavior identifier. The path that references the device record in the XML database is then given in brackets. For Igor, this is done in XPath. [7]

6.1.3 Actor Connections. By this point of the code, Igor knows how to reach the physical devices and their behaviour. It now needs to connect the instances of the devices to one another. This is again done through a straightforward statement. The unique identifier of the instance of the device, where the inputs are coming into, is given followed by a dot and the command "input". The identifiers of the inputs as specified in the actor definition are used followed by an equals sign and the unique identifier of the actor instance as it appears in the actor initiation section. Because of the explicit nature of using both unique identifiers, the order in which these are written does not matter.

Now that the basic program structure has been covered, the following subsections explore, in more detail, the advanced features of the language.

6.2 Triggers and Guards

Triggers and guards are the two child abstractions of the actions and while the simplest form of them was presented above, they can be a lot more powerful in order to accommodate for more complex use cases. Appendix E shows an example of the action section of a light actor which gets triggered by a presence sensor (PIR) and a dusk sensor. In addition the light is an LED strip which can change colors depending on the temperature outside, measured by a thermometer.

The provided example once again uses the trigger construct. It is then followed by a guard, implemented using the `when` keyword followed by brackets containing an expression (state attribute) followed by an comparison operator (<,>,=,!=) and another expression (value or another state attribute). If the expression between the brackets evaluates to true then the block below is executed. Guards also contain implicit triggers. In other words, they will get reevaluated every time one of the state attributes which they contain changes. Guards are not mutually exclusive and therefore multiple guards can evaluate to true at one time. For convenience the `else` keyword is allowed to be used so that the programmer can specify an action for the block of code in the case that none of the guards evaluate to true. This however is only allowed under an explicit `on` trigger.

Lines 3 and 5 of the above listing also show the use of two other convenience features of the language. `whenall` and `whenany` are both designed to allow the programmer to write cleaner code which is easier to read and less prone to mistakes. `whenall(Expression A, Expression B)` means the same as `when(Expression A) AND when(Expression B)` Nesting guards into one another will also achieve the same effect. While `whenany(Expression A, Expression B)` means the same as `when(Expression A) OR when(Expression B)`

6.3 User Defined Functions

The example shown in Appendix E also illustrates the use of the user defined function `ChColor`. As the language restricts what the user can put on the right side of the assignment operator (See section 6.1.1) the programmer is forced to do any arithmetical operations, conversions or more complex checks in separate functions. In the example provided, the function takes in the temperature and returns the appropriate red, green or blue value that the LED light takes as a state. The function needs to be defined in a separate block of the program, outside of the actor definition and can be used globally.

In the current version of the language, these functions have been defined using Python, as it is a language which already provides great facilities for mathematical operations. Future implementations of the language can also allow these to be defined in more than one language such as JavaScript or PHP in order to give the programmer the option to use the one they are most familiar with. In addition libraries of standard operations can also be provided to save the programmer the need to define them all together.

6.4 Time

As per requirement R1.7, the proposed language should allow for programmer to reference time in their program. This is the purpose of the timer abstraction described in section 6.1.4. Appendix E shows an excerpt of code from a program which unlocks and locks a door with an RDF reader. In it the timer is used to facilitate the functionality that the door needs to lock 20 seconds after it has been unlocked.

As already mentioned the timer is actually an actor which in this representation is called "time". Its `timer` state lets you set a desired duration after which the block of code nested bellow it gets executed. *time.time* on the other hand lets the programmer create expressions which evaluate to true only at the specified moment. These can be used in guards to check that a specific action happens at a precise time. Different arguments can be passed through for different needs. For example `time.time(month=July)` evaluates to true every July while `time.time(01/05/2020)` evaluates to true on the 1st of May 2020. The last page of the sample sheet provided in Appendix J contains complete list of arguments and options that can be used with the time actor.

[7] https://www.w3.org/TR/2014/REC-xpath-30-20140408/

6.5 Complex Actors

Section 6.2.2 explains about the inheritance mechanism which the model provides and how complex actors can be built of other simpler actors. Appendix G shows the actor definition of a fridge, which is made out of 3 other actors, each representing one of the fridge's different subsystems. The parent actor is then given states, which do not refer to a path in the database but rather to a state attributes from its child actors. These attributes can then be accessed by other actors which take the fridge actor as an input. The rest of the states of the child attributes stay hidden to the rest of the system.

7 EVALUATION

This section presents the evaluation procedure undertaken as part of this research. The goals of the activity are first specified, followed by the methodology. Finally the results are presented and discussed.

7.1 Goals

The main goal of this evaluation is to test the proposed programming language design with some users and see if it is easy to pick up as well as if it satisfies the gap stated in the research question. In order to achieve this the following atomic queries have been used to make the research question more manageable:

- Is the language easy to pick up and use?
- Are the abstractions of the model at the right level to keep the real world problem and its implementation as close as possible?
- Is the programming structure easy to understand and apply?

7.2 Method

Even though the development of programming languages has now reached a certain maturity with a well established approach, the same cannot be said for their evaluation. Both evaluation methodology and criteria are often areas where the academic community tends to disagree and according to Kurtev et al., universally accepted options for either have not yet been established [17]. This study utilises the Discount method for programming language evaluation as the main evaluation strategy due to its suitability for languages at their design stage rather than at the end of the development cycle. In addition the method allows for a more empirical approach at a much smaller cost compared to the other methods centered around user testing [17]. Even though the procedure followed quite closely the one specified in the above cited paper, the following subsections give the details of the experimental setup for this study.

In addition unstructured interviews were conducted with the test participants in order to gain a better insight into how the users perceived the various language features and how it compares to others they have used. As part of the evaluation process the test participants were also asked to score the language on the criteria for good programming language design as outline by literature. The chosen criteria are: simplicity, uniformity, abstraction, safety, modularity, expressiveness, efficiency. The criteria is consistent between various sources on the subject [8, 33].

7.2.1 Participants. In total six participants were selected for the study including one for the pilot test and five for the rest of the trials. According to the research conducted for this project, there is a lack of studies which suggest that IoT adoption or programming skill has any correlation to gender or age, therefore while demographic data was gathered from the test participants, it was not used as a selection criteria. What was important is that the test participants had at the least a substantial experience in programming. This would allow them to think more critically about the language. In addition, the languages already in their arsenal will allow for direct comparison. Another requirements for selection was the the users had at least a vague idea of the IoT domain as a whole. This requirement makes it the test more realistic as it is reasonable to assume that people who will be interested in the language will also be vaguely familiar of what they want to do with it and how the concept works. Last but not least, some of the participants had some experience in programming IoT devices which allowed them to do a direct comparison between some of the solutions described in section 3 and the new proposed language. A detailed breakdown of the participant infromation can be found in Appendix G.

7.2.2 Test Set Up. The set up used during the test involved a laptop running Notepad++ [8] with two views. On the left hand side was a blank file where the test participants could write their programs. On the right side of the screen was a file containing the XML database where Igor stored the information about the devices which they had to use in their programs. The participants were provided with a sample sheet (Appendix C), which explained the code with examples and a task sheet containing five task (Appendix D), each of which tested different functionality of the language. The test participants were given sixty minutes to complete as many tasks as possible, however as time was not an important factor in this study, that requirement was flexible and the participants were allowed to finish the task they were on, even if the time ran out. A portable audio recorder and screen capture was also used to record their efforts.

7.2.3 Interviews. After the test had finished, an unstructured interview was conducted to gain some more insight into the user's thoughts about the language. Before the interview, the participants were made aware of any major mistakes in their code so that they could comment on how the language actually is meant to be used rather than how they think it is. The interview was recorded with the portable recorder, then transcribed and coded in order to extract more meaningful qualitative data [3]. The interviewees were also asked to score the proposed notation on the following criteria for good programming language design. A Likert scale was used for that purpose where 1 was the lowest and 5 is the highest [32].

- Simplicity: The language should have just a few basic concepts which make it easy to understand [8]
- Uniformity: The basic concepts should be applied universally without change in their form [36]
- Abstraction: are the abstractions provided appropriate for the application domain [8]
- Safety: Are semantic errors easy to make and detect [8]
- Modularity: Interfaces between the different units should be made explicit [8]
- Expressiveness: The language should allow for a large variety of programs from the domain to be expressible [36]
- Efficiency: The language should enable the programmer to produce efficient code [8]

[8] https://notepad-plus-plus.org/

7.3 Results

The following section presents the results achieved by the evaluation methodology. The results here are grouped and summarised by categories. A full list of the results can be found in Appendix E. The treats to validity are also examined.

7.3.1 User Testing. During the user testing, the mistakes that the participants made were recorded and categorised in three categories as per Kurtev et al. [17]:

Critical. These are fundamental misunderstandings of the language structure and programming model. The mistakes which fall into this category were:

- Test participant did not understand that the input names given in an actor input, need to match the names given to those actor classes. This is done to allow for some error checking and therefore the compiler would return an error when trying to execute the input command in the linking, since that class of the actor trying to be input will not match the one that is expected by the actor receiving the input.
- Participants also tried to assign data to temporary or support variables in the action section. This is also something that is not allowed by the model. This was something every participant tried to do on at least one of the tasks.
- All participants but one ignored the restriction of the type of values that can go after the assignment operator in the actions section of the code and performed mathematical operations directly.
- Another aspect which appeared problematic on more than one occasion was the use of the "time" object. Participants did not seem to understand what the expressions outputted and how it operated and therefore it was either placed at the wrong place or used as part of expressions which did not expect it. In some cases it was missed out all together, however that is considered to be more of misunderstanding of the logic of the program rather than a problem with the language construct.
- Three of the participants had problems with assumptions of implied logic. In other words, they only specified half of the actor's behaviour, assuming that the reverse is implied in the logic of the language.
- In the cases of the more experienced programmers there were a few instances where they were introducing concepts from other languages as the syntax reminded them of them and therefore they assumed that they are allowed.

Serious. These are structure errors which will result in a program which fails to produce the intended output or one at all, but can be fixed with small changes

- Usage of disallowed operators such as `AND` and `OR`
- Using the `else` construct without an explicit trigger (on), which results in the compiler never evaluating the update expression as it doesn't know what to changes to listen out for to trigger the action.
- Another aspect that was commonly missed was the explicit assignment in the linking of devices section. Example: *lightKitchen.input(switchKitchen)* instead of *lightKitchen.input(switch = switchKkitchen)*

Cosmetic. These are typos and small syntax errors that can generally be fixed with a couple of character changes.

- Use of "==" instead of "="
- Use of "=" instead of "<-"
- Indentation missing or colon after a statement which expects it
- Typos in names of variables

Other Observations. A trend appeared that the participants who knew more languages were making more errors than those who knew less or had less experience. This is most likely caused by more assumptions made on their part, rather than just sticking to what the sample sheet showed. It also appeared that since most of the participants had experience with imperative programming languages, they tried the tasks with that mindset and tried to incorporate aspects of that paradigm rather than keeping their programs declarative.

The `else` construct, while not explicitly shown on the sample sheet, was another favorite of all of the participants. Test subjects were also unsure of the time construct, however after some clarification they said that it makes sense and therefore it is probably just not explained very well in the sample sheet.

Participants also liked the `whenany` and `whenall` conveniences, however they expressed doubts towards having to list all of the state and a couple of them suggested having a default where the states section gets populated automatically with the same names as the state attributes appear in the database as the explicit assignment might be too laborious for complex devices with a lot of states.

7.3.2 Interviews. After transcribing them, the interviews were coded in order to make better sense of the results [3]. The following conclusions were reached.

- After completing the first task, all participants agreed that the language is very straightforward to pick up. *"I thought it was really intuitive to begin with it"* (Participant 2)
- Four out of the five participants explicitly mentioned that the abstractions provided really helped them think in terms of the problem as it is presented in the real world, rather than having to translate it to abstract computer science concepts. This is perhaps the most important observation, as it directly answers the research question posed by this study. In addition, it not only made the program easier to write, but also saved time as they didn't have to worry about the processes happening in the background. *"just the whole idea that you have your states and you know, your actions and everything that happens in the background is abstracted from you is a big time saver, I think."* (Participant 3)
- All of the participants agreed that while they can see that the same results are possible to achieve with other languages, they could appreciate that with the presented model the tasks will be a lot quicker and more efficient to complete. *"it's pretty well defined. I think that it's designed for this task, and you can see that."* (Participant 5)
- The majority of the participants discussed how they can see this language becoming very easy to adopt with the addition of a few standard libraries of functions and device behaviours. This way even really novices can utilise it fully. *"there could be some other method where you don't have to define things so much"* (Participant 4)

The table in Appendix I shows the results of how the participants scored the language on the above mentioned criteria.

7.4 Discussion

The tasks presented in the evaluation procedure showed that the language satisfies all of the functional requirements presented in section 4.2. In addition, while the results revealed some aspect of the language that can benefit from being redesigned (Section 9.1), the evaluation procedure revealed that the language presented does in fact reduce the gap between what the programmer is trying to achieve and how they actually manage it. The study therefore manages to answer affirmatively the question it sets out to. The abstraction accomplish exactly what they were designed to do, that is to allow the programmer to think in terms of the real world problem and not have to translate the domain into metaphysical concepts in order to be able to fit into a programming paradigm. As shown by the user test, it doesn't take too much to explain the language fully to someone who has never seen it before. This is further a testament to the model's simplicity and how a few key concepts can be applied to solve a variety of use cases.

Furthermore, the test participants all scored the language highly on all of the criteria for good programming language design. The efficiency criteria received the lowest score due to the fact that participants did not like the restrictions on values in the update expressions, however they agreed that the addition of a standard library of functions which will save them the hustle to specify them themselves should resolve that issue. The expressiveness criteria also received a lower score, however two of the participants who gave them admitted that they did not feel confident enough to score the language highly on this criteria, as they felt that their knowledge of the domain is fairly limited. Nevertheless, they confirmed that the language seems powerful enough to facilitate the use cases which they have in mind. This concern also represents the biggest threat to the validity of these results. The small number of test participants gives rise to high proportion of bias. A user study with a higher number of participants at a later stage where a compiler or IDE can produce some feedback to the users will help alleviate this problem. While the user's were all experienced programmers, their knowledge of the IoT domain was relatively limited. This poses some treat to the validity of their evaluation of the expressiveness of the language. Performing further test with domain experts will be advantageous. However as the results were consistent between all testing trials and therefore considered to be robust.

8 FUTURE WORK

Based on the results of the evaluation and the some of the opportunities identified in the related work section, this research can be expanded with the following future work:

8.1 Language Improvements

Even though the current form of Yogurt was received well by the test participants, the evaluation process revealed some possible improvements:

- Provide a default for the state section. As stated in the results section, a couple of test participants noticed that a lot of the time they use the same names for state attributes as the once already given in the data store. Therefore, it was suggested that there should where they are allowed to use them without specifically rewriting them.
- The code produced by the test participants involving the use of `AND` and `OR` operators in the guards reduced the readability of the blocks, therefore these should be disallowed all together. Rather programmers should be forced to use the `whenall` and `whenany` constructs
- The fact that almost all participants tried to do something procedural during the study, suggests that order and sequencing of the action execution is important and perhaps should be allowed. One elegant solution could be do take advantage of the indention formatting for sequencing. Blocks of code can be executed left to right in order. To facilitate this the keyword `then:` can be introduced. Code following it will have to indented further to the right and therefore will be executed after the compiler is done will all commands on the same level as the `then`
- The time functionality seemed to confuse a lot of people and while some of that can be attributed to poor documentation on the Sample sheet, some more thought can go into the timer object. One potential redesign could be to separate the timer functionality from the absolute time checks all together.
- Create libraries for standard functions and device behaviours in order to improve efficiency. While the explicit nature of the program helped the programmer understand the language better, it makes the coding too verbose especially for longer use cases.
- Additional functionality that was also supported by the tests was to allow multiple languages to be used for the function definition section.
- Another improvement is to give the language features so that the "config" block of the code can also be written in Yogurt
- Remove the `.input` command from the device connecting block for consistency with the rest of the language.

8.2 Implementation

While the evaluation of the programming model and the design of the language yielded promising results, a fully capable compiler or interpreter is yet to be developed for the language. Implementing the language will not only be required for its adoption but also provide opportunity to conduct more comprehensive user tests as well as evaluate execution speed and computational efficiency. The latter being of particular importance as the solution will have to operate on devices with limited processing power due to their size.

8.3 Graphical Representation

Several of the solutions in the related work section are presented through graphical user interfaces (GUIs). Undoubtedly the practice has the benefits of reducing errors caused by typing or misinterpretation of the data flow through the program. Therefore a graphical representation for the programming language may be greatly beneficial for its adoption. In addition, this research failed to identify any studies that prove weather or not GUIs have a role in reducing the learning curve required to learn a new programming language. Having a graphical implementation will allow to test this hypothesis further.

9 CONCLUSION

Yogurt is a domain specific, declarative, object-oriented programming language, designed for the Igor and the Internet of Things. Its main goal is to achieve high expressiveness by providing its users with the right features to tackle the challenges in the domain of IoT while staying simple and easy to learn. This is achieved by providing the right level of abstraction which allows the user to think in terms of the problem as it is set in the real world and not concern themselves with background process, memory management, control flow and dealing with low-level device outputs and inputs. User testing reveals that, while there is always room for improvement, the language presented in this paper manages to achieve that goal. And while these results are by no means impervious to scrutiny, the methodologies used to achieve them are all based on sound principles. It is clear that the Internet of Things is only going to become larger and larger and as this technology becomes part of the lives of more people, it is essential that the tools created to interact with it reflect both the direction of its development and the needs of its users.

REFERENCES

[1] Krzysztof R. Apt. 1997. *From logic programming to Prolog.* Prentice Hall, London ; New York.

[2] Athom. 2019. Homey. https://www.athom.com/en/

[3] Z Bauman, U Beck, E Beck-Gernsheim, S Benhabib, RG Burgess, M Chamberlain, P Thompson, P Chamberlayne, J Bornat, T Wengraf, et al. 2011. Qualitative interviewing: Asking, listening and interpreting. *Qualitative research in action* (2011), 226–241.

[4] Ivan Bratko. 2012. *Prolog programming for artificial intelligence* (4th ed ed.). Addison-Wesley, Harlow, England ; New York.

[5] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2017. A high-level approach towards end user development in the IoT. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 1546–1552.

[6] Alan M. Davis, Edward H. Bersoff, and Edward R. Comer. 1988. A strategy for comparing alternative software development life cycle models. *IEEE Transactions on software Engineering* 14, 10 (1988), 1453–1461.

[7] Andrew Eisenberg and Jim Melton. 1999. SQL: 1999, formerly known as SQL3. *ACM SIGMOD Record* 28, 1 (March 1999), 131–138. https://doi.org/10.1145/309844.310075

[8] Raphael A. Finkel. 1996. *Advanced programming language design.* Addison-Wesley, Menlo Park, Calif.

[9] Maurizio Gabbrielli and Simone Martini. 2010. *Programming languages: principles and paradigms.* Springer Science & Business Media.

[10] Narain Gehani. 1991. *Ada: concurrent programming.* Silicon press.

[11] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc.

[12] Patricia Hill and John Wylie Lloyd. 1994. *The Gödel programming language.* MIT press.

[13] Jack Jansen and Steven Pemberton. 2017. An architecture for unified access to the internet of things. *XML LONDON 2017* (2017).

[14] Mengda Jia, Ali Komeily, Yueren Wang, and Ravi S Srinivasan. 2019. Adopting Internet of Things for the development of smart buildings: A review of enabling technologies and applications. *Automation in Construction* 101 (2019), 111–126.

[15] Surabhi Kejriwal and Saurabh Mahajan. 2016. Smart buildings: How iot technology aims to add value for real estate companies. *Deloitte Center for Financial Services* (2016).

[16] Brad Kelechava. 2018. The SQL Standard - ISO/IEC 9075:2016. https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/

[17] Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen. 2016. Discount method for programming language evaluation.. In *PLATEAU@ SPLASH*. 1–8.

[18] Nicola Leonardi, Marco Manca, Fabio Paternò, and Carmen Santoro. 2019. Trigger-Action Programming for Personalising Humanoid Robot Behaviour. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, Glasgow, Scotland Uk, 1–13. https://doi.org/10.1145/3290605.3300675

[19] J. W. Lloyd. 1994. Practical Advantages of Declarative Programming. In *GULP-Prode'94*, Vol. 1. Universitat Politecnica De Valencia, Peniscola, Spain, 3–17. http://www.programmazionelogica.it/wp-content/uploads/2015/12/GP1994-I-000-031.pdf

[20] K.L. Lueth. 2018. State of the IoT 2018: Number of devices now at 7B - Market accelerating. https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/

[21] Mark Lutz. 2001. *Programming python.* " O'Reilly Media, Inc.".

[22] Node-RED. [n. d.]. Node-RED : Node-RED Cookbook. https://cookbook.nodered.org/

[23] Oracle. 2019. Big Data @ Work: 'Internet of Things' promises limitless data, limitless possibilities. http://www.oracle.com/us/dm/lpd100392169-oracle-iot-pa-2430014.pdf?elqTrackId=896cc8cfbc2f47a89ffbf0090f240066&elqaid=38549&elqat=2

[24] Amir Rahmati, Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2017. IFTTT vs. zapier: A comparative study of trigger-action programming frameworks. *arXiv preprint arXiv:1709.02788* (2017).

[25] Pethuru Raj and Anupama C Raman. 2017. *The Internet of things: Enabling technologies, platforms, and use cases.* Auerbach Publications.

[26] Anoja Rajalakshmi and Hamid Shahnasser. 2017. Internet of Things using Node-Red and alexa. In *2017 17th International Symposium on Communications and Information Technologies (ISCIT)*. IEEE, 1–4.

[27] Rafael Ramirez, Malobi Mukherjee, Simona Vezzoli, and Arnoldo Matus Kramer. 2015. Scenarios as a scholarly methodology to produce "interesting research". *Futures* 71 (2015), 70–87.

[28] Suzanne Robertson and James Robertson. 2012. *Mastering the requirements process: Getting requirements right.* Addison-wesley.

[29] Barbara G Ryder and Ben Wiedermann. 2012. Language design and analyzability: a retrospective. *Software: Practice and Experience* 42, 1 (2012), 3–18.

[30] Bjarne Stroustrup. 1995. *Why C++ is not just an object-oriented programming language.* Vol. 6. ACM.

[31] Paula Ta-Shma, Adnan Akbar, Guy Gerson-Golan, Guy Hadash, Francois Carrez, and Klaus Moessner. 2017. An ingestion and analytics architecture for iot applied to smart city use cases. *IEEE Internet of Things Journal* 5, 2 (2017), 765–774.

[32] W. Trochim. 2006. Likert scaling.

[33] Franklyn Albin Turbak, David K. Gifford, and Mark A. Sheldon. 2008. *Design concepts in programming languages.* MIT Press, Cambridge, Mass. OCLC: ocn214322997.

[34] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical trigger-action programming in the smart home. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*. ACM Press, Toronto, Ontario, Canada, 803–812. https://doi.org/10.1145/2556288.2557420

[35] Marcel Walch, Michael Rietzler, Julia Greim, Florian Schaub, Björn Wiedersheim, and Michael Weber. 2013. homeBLOX: making home automation usable. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication - UbiComp '13 Adjunct*. ACM Press, Zurich, Switzerland, 295–298. https://doi.org/10.1145/2494091.2494182

[36] David A. Watt. 1990. *Programming language concepts and paradigms.* Prentice Hall, New York.

[37] Pauline Sia Wen Shieng, Jack Jansen, and Steven Pemberton. 2018. Fine-grained Access Control Framework for Igor, a Unified Access Solution to The Internet of Things. *Procedia Computer Science* 134 (2018), 385–392. https://doi.org/10.1016/j.procs.2018.07.194

Appendix

A ACKNOWLEDGMENTS

Sincere gratitude goes out to Jack Jansen, Steven Pemberton, Pablo Cesar and the rest of CWI's Distributed and Interactive System's group for providing continuous council for the duration of this project as well as sharing their expertise and support. In addition, to Frank Nack from the University of Amsterdam for his guidance in this endeavor. Finally, the same goes to all of the test participants and others who shared opinions or advise in relation to this work.

B BASIC LIGHT SWITCH PROGRAM

Listing 1: Basic Light Switch Program

```
#define switch behaviour
actor switch():
    config:
        # connect underlying architecture in which model has been implemented
    states:
        on: /on
    actions:
        self

#define light behaviour
actor light(switch):
    config:
        # connect to underlying architecture in which model has been implemented
    states:
        illuminated: /active
    actions:
        on(switch.on):
            illuminated <- switch.on

#instantiate light and switch
kitchen_light = light("//data/devices/light[location='Kitchen']")
kitchen_switch = switch("//data/sensor/switch[location='Kitchen']")

#connect the instances of the devices together
kitchen_light.input(switch = kitchen_switch)
```

C MULTIPLE INPUTS FROM THE SAME ACTOR

Listing 2: Input from two different instances of the same actor

```
actor light(sw1 = switch, sw2 = switch):
```

D COMPLEX TRIGGERS AND GUARDS

Listing 3: Complex Triggers and Guards

```
actions:
    on(PIR_sensor.presence):
        whenall(PIR_sensor.presence = True, Dusk_sensor.night):
            on <- True
        whenany(PIR_sensor.presence = False, Dusk_sensor.night = False)
            on <- False
    on(Thermometer.temp):
        when(on = active):
            R <- ChColor(Thermometer.temp, R)
            G <- ChColor(Thermometer.temp, G)
            B <- ChColor(Thermometer.temp, B)
        else:
            R <- 0
            G <- 0
            B <- 0
```

E TIMER EXAMPLE

Listing 4: Timer Example

```
on(RDF.engaged):
        when(RDF.engaged=True):
                locked <- False
                time.timer(00:20:00.0):
                        locked <- True
```

F COMPLEX ACTOR EXAMPLE

Listing 5: Fridge Example

```
actor Fridge():
    states:
        temperature: Thermostat.temp
        on: Thermostat.on
    actor Door():
        config:
            # connect underlying architecture in which model has been implemented
        states:
            open: /open
        actions:
            self
    actor Light(Door):
        config:
            # connect underlying architecture in which model has been implemented
        states:
            on: /active
        actions:
            when(Door.open = True):
                on <- True
            when(Door.open = False):
                on <- False
    actor Thermostat():
        config:
            # connect underlying architecture in which model has been implemented
        states:
            on: /active
            temp: /temperature
        actions:
            when(time.time(07:00)= True):
                temp <- 5
            when(time.time(24:00)= True):
                temp <- 9
```

G TEST PARTICIPANT INFORMATION

Table 1: Test Participant Information

Participant	1	2	3	4	5
Age	26	23	25	24	27
Gender	Male	Male	Male	Male	Male
Nationality	Spanish	British	Dutch	Greek	Greek
Profession	Student	Student	Student	Student	Student
Education	MSc Data Science	BSc Computer Science	MSc Data Science	MSc Data Science	MEng Electrical Engineering
IoT Knowledge	Familiar	Familiar	Vaguely Familiar	Familiar	Familiar
Home Automation Experience	Yes	No	No	No	No
Devices	Light Buld, Chromecast	N/A	N/A	N/A	N/A
Configuration	IFTTT through phone GUI	N/A	N/A	N/A	N/A
Programming Experience	Professional	Substantial	Substantial	Professional	Substantial
Programming Languages	Python, JavaScript, Java	Python, Ruby, Java, JavaScript, SQL, HTML, CSS	Python, Java, Octave, Scala, Mathematica, Delphi	C, Python, JavaScript, R	Python

H TEST PARTICIPANTS INFORMATION SHEET

Participant Information Sheet

Dear Participant,

Before the research begins, it is important that you are aware of the procedure that is followed in this study. Please read the text below carefully and do not hesitate to ask for clarification on this text, if it is not clear please ask the researcher before beginning.

Purpose of the study
During this experiment we are interested in evaluating the design of a new programming lanague designed for Igor, whcih is an architecture for unified access to the Internet of Things. The purpose of the experiment is to evaluate the lanague and **not** your skills as a programmer.

Explanation of task
You will be given five tasks to complete. Each task will contain a scenario describing the enviorment and behaviour of an IoT system. You will then need to write a program to fascilitate this behaviour as specified in the scenario. The program must be written in the programming lanague described in the sample sheet provided. To do so you have to use the computer and text editor available. The experiment is expected to take **60 minutes in total**. You may run out of time before you complete all the tasks and that is ok. Please try to think out loud as much as possible. Your answers and screen will be recorded so that the data can be thoroughly analyzed later. You will be expected to answer a few questions after the task.

Experimental Procedure
1. Read information Sheet (this sheet), and fill in the consent form
2. Fill in the subject information form (e.g. age, gender)
3. Once ready to begin, receive the task and sample sheet
4. Complete the tasks
5. Answer a few post test questions

Freedom to withdraw
If you decide not to participate in this study, this will in no way affect you. If you gradually decide to stop the research, you can do so at any time without giving reasons and without any consequences for you in any way.

Your privacy is guaranteed
Your personal information (who you are) remains confidential and will not be shared without your explicit consent. Your research data are further analyzed by the researchers who collected the data. Research data published in scientific journals are anonymous and cannot be traced back to you. Fully anonymized research data may also be shared with other researchers.

Further information
If you have questions about this research, in advance or afterwards, you can contact the responsible researcher; Ivan Gorbanov (Ivan.Gorbanov@cwi.nl).

Sincerely,
Ivan Gorbanov
Centrum Wiskunde & Informatica

I SAMPLESHEET

CWI Centrum Wiskunde & Informatica

Sample Sheet

General Information

The language tested in this study is a declarative, object-oriented programming lanague for programming Igor, which is an architecture for unified access to the Internet of Things. In other words, Igor acts as a butler who is connected to all sensors and smart devices in its user's environment. The devices and data are stored in an XML repository which allows for bi-directional updates. This means that as a device changes its state (ex.: switch gets turned on from being off) the database will update itself automatically (data/devices/switch/active=true). In addtion, if a program changes the value in the database, the coresponding device state will change as well. (ex.: data/device/switch/active=false -> switch turning off)

General Structure

In this lanague we call IoT devices "actors". The program then defines what type of actors are used, what capabilities they have (what actions can they perform), instantiates instances of those actors and how they connect to eachother. Every program in this new language has the following structure:

Actor Class definitions:

In the first section the programmer must define all classes of actors in his or her system. The example below shows the actor definitions for a switch and a light. Comments are seporated by hashtags.

```
actor light_switch():                          # actor classes are given unique IDs
    states:                                    # each actor has one or multiple states which
        engaged: /active                       are values in the database. Assign it a local
    actions:                                   name (CAN BE ANYTHING) and point it to its path
        self                                   in the database
                                               # as passive sensors cannot do anything apart
actor light(light_switch):                     from chancing their own state, their actions
    states:                                    are always "self"
        on: /active                            # specify any inputs to the actor in the
    actions:                                   brackets
        when(light_switch.engaged = True):     # actuators have actions which have a condition
            on <- True                         that needs to be satisfied, multiple conditions
        when(light_switch.engaged = False):    can be nested in each other
            on <- False                        # the action itself is always updating the
                                               value of a state belonging to that actor.
                                               # the right side of the update symbol can only
                                               contain a Boolean, number, "string", state
                                               variable or a function
```

Actor Instantiations:

After the class definitions are written each instance needs to be instantiated. The definition is given a unique id followed by an equals sighn followed by the actor id and its path in the xml database. The path is specified in Xpath: it starts with two forward slashes, followed by the path. If there are multiple nodes with the same name, filters can be applied in square brackets to select the unique option.

```
dusk_sensor = dusk("//data/sensors/dusk")
bathroom_light = light("//data/devices/light[location='Bathroom']")
bathroom_switch = light_switch("//data/devices/switch[IP='192.168.37.68']")
```

Actor Connections:

After all devices are instantiated, the programmer then needs to connect together the instances which interact with eachother.

```
bathroom_light.input(light_switch=bathroom_switch)
```

User Defined Functions:

As the actions are restricted to only having values to the right side of the update expression, the user can take advantage of functions for more complex needs. The functions are defined at the beggining of the program and are written in Python like syntax. Only pure functions are allowed to be used. #maybe explain

```
# function for converting kg to pounds
def pound_converter(input):
    output = input * 2.205
    return output

actor display(scale):
    states:
        on: /active
        text: /display_text
    actions:
        when(scale.value):
            text <- pound_converter(scale.value)
```

Action Triggers:

Action conditions gets reevaluated every time one of the values it incorporates changes. If the condtion contains just the name of the variable the action will run everytime that value changes, like in the example above: text on screen changes everytime the weight on the scale changes. The trigger in this case is implied. Nested whens can be used for further condition checking. Multiple conditions can be put after the same when with an “and” or “or” logical operator. If the programmer wants to specify an explicit trigger for the condition and action, they can do so with the “on” command, like in the example bellow. Even though the guards check three different values before the “on” state changes, they only get evaluated when “switch.active” changes.

```
actor light(switch, PIR, dusk):
    states:
        on: /active
    actions:
        on(switch.active):
            when(switch.active = true):
                when(dusk.night):  # when checking a Boolean value “= true” can be implied
                    when(PIR.presence):
                        on <- true
```

Complex Conditions:

Some use cases will require the use of complex condtions where many states are checked. To avoid verbosity the language is equiped with the following conviniences:

```
whenany(switch.active, dusk.night) # is the same as when(switch.active) OR when(dusk.night)
whenall(witch.active, dusk.night) # is the same as when(switch.active) AND when(dusk.night)
```

Time:

In order to fascilitate use cases where time is important the lanague provides a built in time function which works as follows:

```
time.timer(00:30:00.0):    -> executes the code bellow after 30 min

time.time(dd/mm/yyyy)      -> evaluates to true when the date matches the day of execution

time.time(month=July)      -> evaluates to true everyday of July in every year
time.time(year=1994)       -> evaluates to true every day of 1994
time.time(date=1)          -> evaluates to true on the first of every month of every year
time.time(day="Monday")    -> evaluates to true on a Monday
time.time(date=1, month=July or August) -> evaluates to true on the first of July and August

time.now(mm/yyyy)          -> returns the current time in the format mm/yyyy
time.now(h/m/s/ms dd/mm/yyyy/) -> returns the current time in the format 12:24:35 21/06/2019
```

J TASKSHEET

CWI Centrum Wiskunde & Informatica

Task Sheet

Please try and complete as many of the tasks as possible but dont worry if you dont manage to do them all in the aloted time. The tasks are placed in order of increasing difficulty so it helps to do them in the order they appear to familiarize yourself with the language. Use the sample sheet to learn the language constructs, you may ask the supervisor for help but please do so as last resort, if he believes that the question is covered by the task sheet, he may reffer you to it. Your 60 minutes start now!

Task 1:

Scenario: Imagine the light in the atrium of your house operates based on a presence sensor (PIR). However, the light only comes on if a separate dusk sensor placed outside the house detects that it is dark.

Task: Please write a program which results in that behavior!

Task 2:

Scenario: Imagine that you have just installed new smart lights and switches in your entire home and you will need to program their behaviour.

Task: Please write a program which connects the switch in the kitchen to the light in the kitchen and the switch in the living room to the light in the living room.

Task 3:

Scenario: Imagine that your door locks by itself and unlocks when the presence sensor in your atrium detects presence (PIR). Once you leave the house the door locks itself 20 seconds after the sensor can no longer detect you. When you come back home, you use a key card on an RDF reader by the door to unlock it. Once you swipe the card, the door stays unlocked for 20 seconds before it locks again.

Task: Write a program for the card reader, door and presence sensor which results in the behavior described above.

Task 4:

Scenario: Imagine that to achieve greater efficiency and comfort at home you decide to program your thermostat to automatically adjust the temperature inside your house (TI) depending on the temperature outside (TO). You install a new thermometer on the outside of the house that records the temperature and configure your thermostat to react accordingly. However, you make a mistake and accidently buy a thermometer that outputs the temperature reading in Fahrenheit, while your thermostat accepts only Celsius (C).

Task: Write a program which results in the following behavior:

- When TO is higher than 30 C set TI to 25 C
- When TO is lower than 10 C set TI to 20 C
- Otherwise TI = TO(0.25) + 20 C

The relationship between Fahrenheit and Celsius is as follows:

- Fahrenheit = (Celsius * (9/5)) + 32
- Celsius = (Fahrenheit - 32) / (9/5)

Task 5:

Scenario: Now imagine that the lights that you configured during task 2 are actually LED and can change colors.

Task: Copy your code from task two and extend it so that the lights change color depending on the season (month) of the year as follows:

- Spring (March, April, May) -> color: direct sun (R = 255; G = 255; B = 255)
- Summer (June, July, August) -> color: overcast sky (R = 201; G = 226; B = 255)
- Autumn (September, October, November) -> color: halogen (R = 255, G = 241, B= 224)
- Winter (December, January, February) -> color: tungsten (R = 255; G = 214; B = 170)

K TEST RESULTS

Task	Critical	Serious	Cosmetic
1	· no expression to turn light off · introduced a variable then tried to inject it in the path pointing to the database like you would be allowd to in Python · used the python format() function · did not understand that input names need to match actor definition IDs	· used an "AND" operator in the value assignment expression · used "else" without an "on"	typo in databse reference
2	· did not understand that input names need to match actor definition IDs	· used "else" without an "on" · missed out explicit assignment in linking of actors	
3	· missing PIR unlock expression · missing an expression to lock the door if the RDF is triggered bu noone comes in · did not understand that input names need to match actor definition IDs · put "time.timer" inside a when · logic on "false" not there · timer at wrong place; not needed there for the logic of the task	· missing timer for the door to stay unlocked once RDF is triggered · missed out explicit assignment in linking of actors	· used "=" instead of "<-", but only in one instance so error is treated as cosmetic · used "==" instead of "=" · colon and indent on timer function
4	· mathematical expression in the action section · assigning values to variables in the action section · mathematical expression in the action section · assigning values to variables in the action section · mathematical expression in the action section · assigning values to variables in the action section · mathematical expression in the action sec-	· used an "AND" operator in the "when" statement · function deffinition after it has been used · missed out explicit assignment in linking of actors · function deffinition after it has been used · missed out explicit assignment in linking of actors	
5			· typo in mames of variables in datastore

L EVALUATION CRITERIA RESULTS

Table 2: Evaluation Criteria Results

Criteria	Part. 1	Part. 2	Part. 3	Part. 4	Part. 5	Avg. Score
Simplicity	4	4	4	4	4	4
Uniformity	4	4	5	4	4	4.2
Abstraction	5	5	5	3	5	4.6
Safety	4	4	5	4	3	4
Modularity	4	4	4	3	5	4
Efficiency	4	3	4	4	4	3.8
Expressiveness	4	3	5	3	4	3.8