Lantern: A domain specific language for energy awareness in smart-homes

Jon Robinson [a]* Kevin Lee [b] and Kofi Appiah [c]
[a] *School of Science and Technology, Nottingham Trent University, Nottingham, UK*
E-mail: jon.robinson@ntu.ac.uk
[b] *School of Information Technology, Deakin University, Melbourne, Australia*
E-mail: kevin.lee@deakin.edu.au
[c] *Department of Computer Science, University of York, York, UK*
E-mail: kofi.appiah@york.ac.uk

Abstract. This paper argues that energy consideration should be central to software development. It speculates that including the notion of energy awareness in programming language design for domain specific languages (DSLs) is a novel way in which energy-aware and energy-efficient applications can be developed. It outlines the design criteria and rationale for using a language-focused approach for energy-awareness. It proposes Lantern, a DSL for supporting energy awareness in Cyber-Physical Systems software development. Lantern allows the development of applications that better manage and reduce the carbon footprint of devices. The design of Lantern is aimed at supporting the general development of Cyber-Physical Systems. This paper focuses on the scenario of smart homes, using statically defined locations within a specified environment.

Keywords: Home Automation, Energy Awareness, Distributed Systems

1. Introduction

Energy-awareness and the ability to dynamically adapt the power consumption of electronic devices has become an important challenge in reducing the carbon footprint of individuals and organisations. Today's computers and smartphones may appear to be energy efficient, but they consume approximately 10% of the world's electricity [1]. For example, the higher power consumption of an iPhone (322kWh per year) has been attributed to wireless communication and data usage. With the expected exponential increase in wireless communication and data usage due to the prevalence of Internet of Things (IoT) technologies [2], there will be more demands on energy consumption to power these abilities or services. Thus, the next generation of computing will be driven by enormous amounts of data transmission, storage and processing, calling for new and efficient energy-aware programming approaches.

Over the past several decades programming languages have been designed either to be general purpose for a wide range of needs or domain specific to support the development of specific applications. This paper focuses on designing and implementing a domain specific language (DSL) called Lantern to act as mid-way point between full programming languages and specific languages to solve a particular problem domain.

The research described in this paper builds upon existing work [3] by investigating how energy demands within smart environments can be effectively managed by developing a new DSL which is specifically for efficient energy consumption within environments, rather than as an extension to an already existing language. This paper argues that controlling the energy consumption of highly distributed connected devices (e.g. IoT or other smart devices) can be better managed by designing a new language and middleware around the notion of low-energy.

The remainder of the paper is organised as follows. Section 2 provides a background on the current state-of-the-art in energy awareness in software development. Section 3 introduces Lantern to support the development of energy-aware software along with the case study of smart-homes. Section 4 provides a language centric evaluation and discussion for the next language iteration. Finally, Section 5 provides some conclusions and discusses future work.

2. Energy-awareness in software development

There is an ever-increasing burden on moving away from the use of fossil fuels to clean or renewable energy which presents today's society with a number of challenges. Hence, reducing carbon production to limit the exposure of greenhouse gases has become a major challenge for the 21st century.

While efforts are being made to reduce the greenhouse gases through the production of clean energy, energy-awareness is vitally important when applied to society and their consumption habits. This area is being investigated and acted upon through different regional and government initiatives to provide more energy aware cities and buildings. With the advent of pervasive computing, and more latterly with the introduction of IoT devices, sensors are becoming more integrated into buildings and homes. Coupled with actuators, sensor systems can impact the environment in which they are located. Energy-awareness in embedded systems has already resulted in extensive research into efficient energy use in relation to their primary components (e.g. CPU and wireless communication) and ability to power scavenge.

The current approach to energy-aware systems development is to include loose notions of energy into an existing system by introducing functionality that adapts power use rather than modifying the underlying development. Another common way of reducing energy in the embedded world is to take a more efficient way of controlling the underlying hardware. The purpose of Lantern is to introduce energy-awareness into the design of a language rather than as an add-on or secondary consideration which provides a unique way in developing energy-aware systems.

Embedded systems that have been designed for improved performance may offer some form of energy efficiency. However, energy efficiency in embedded systems can be achieved if the architectural or circuitry levels have been designed with this in mind [4]. Static techniques can be used at the architectural level as embedded systems are typically used for well-defined applications. Other approaches may include the use of multi-core and multi-level caches.

In [5] a technique for mapping concurrent tasks onto a heterogeneous multiprocessor architecture in order to minimise the power consumption of a multimedia based embedded system is introduced. The design in [5] explores Pareto curves and scenarios generated at design time to minimise energy consumption.

Ma et al. [6] proposed an approach similar to [5], where a security-aware and energy-efficient scheduling algorithm aims to reduce energy consumption while ensuring real-time and security requirements. Based on feedback control theory, they employ a feedback unit to keep track of the CPU utilisation and manage the security level dynamically. Using low power modes, Awan and Petters [7] proposed a novel enhanced race-to-halt approach to reduce the overall system energy consumption to be integrated into practical embedded systems. Their technique computes the break-even time for each mode using offline analysis and uses it to save extra leakage energy in lower priority tasks.

Several proposals focus on micro-architectural techniques for saving energy in specific components of embedded systems. These techniques leverage application properties or variations in workload to dynamically reconfigure the components of the system to save energy [8].

The notion of energy-awareness also applies to mainstream electronic products (such as mobile phones, tablets, laptops and other mobile devices) and network systems. This can manifest itself through power usage policies to limit and throttle the devices' CPU, wireless and cell power use in communications to adaptive screen brightness to improve the power drain on batteries between recharge cycles. This consideration is becoming more socially important [9] considering the change in technology used by the general population over the past 20 years.

Complementing the advances in energy-awareness found within IT, embedded and mobile systems, there are challenges in the production of energy-aware software systems. There are several main areas of focus of energy-awareness in software development. Application-level approaches advocate applications being energy-aware and controlling their own energy use [10]. Tools can support the monitoring of applications to provide information as to their energy usage [11, 12] or middleware's can enable efficient energy usage in applications without them explicitly being aware of this focus [13, 14]. The need for a dedicated programming language with resource constraints to streamline usage was identified in [15] and has been a research trend mainly for security [16]. In [17], updates to the International Technology Roadmap for semiconductors (ITRS) [18] are discussed.

The ITRS provides a roadmap of hardware and software technologies in the design and development of silicon systems. The road map outlines the trends of future technologies to address challenges regarding the cost of design and power / energy use. The later challenge is where the Lantern system is placed. Future trends within the ITRS show that power-aware systems are currently a challenge in the control of electronic devices. Thus, with the popularity of the uptake of IoT devices, programming them in an energy-aware manner is an important problem to address.

The majority of the approaches to energy-awareness are runtime-based, however, this can come at the expense of requiring more energy. The alternate approach, advocated in this paper, is to build energy-awareness into the programming language used to develop software solutions and aligns with the trends in the ITRS. Current programming languages provide developers with functionality to coordinate and control devices (for instance, MQTT and IFTTT) when producing software systems. However, when considering the energy-awareness issues associated with, for example, mobile devices, power efficient algorithms have to be developed within a general-purpose language or development platform. Currently, languages which are built around the notion of energy have not been explored thoroughly. The remainder of this paper proposes and demonstrates the Lantern DSL with special emphasis on home automation, which generally fills the gap in energy aware programming languages.

3. The Lantern energy-aware domain specific language

The primary contribution of Lantern is to investigate how programming and control languages can become more energy-aware by taking the approach of building a new language from the ground up and incorporating the notion of energy-awareness in its design and implementation. This ensures that language concepts, constructs and processes can be seamlessly introduced into a language which promotes energy-awareness and provides the developer with a way of expressing how to control and allow environments to react based on their energy demands. This allows us to directly address the question of *controlling the energy consumption of highly distributed connected devices (e.g. Internet of Things (IoT) or other smart devices) can be better managed by designing a new language and middleware around the notion of low-energy*.

The focus of this iteration of Lantern is on the relatively static domain of smart-homes; it is acknowledged that Cyber-Physical Systems (CPS) also contain mobile and dynamic devices - these will be considered in future versions of Lantern. The requirements of the language are to offer an expressively rich DSL by introducing the concept of energy-awareness within its core. This is to allow the language to adapt to changing power demands within the environment that requires language constructs which simplify and facilitate in the management and coordination of an environment. Some of the fundamental design considerations in driving the design and implementation of the language and middleware would be: finding, binding and adapting to constantly varying environment smartness (through embedded or mobile sensors and devices); compositional expressiveness to adapt to changing environmental and energy contextual information; and middleware requirements underpinning the language concepts and notions. By providing a service oriented architecture, where services will be running within an ad hoc adaptive middleware layer, the compositional and energy adaptive capabilities of the language and middleware will also include the rudimentary notion of identity. Linked with these identities will be an expressive predicate-based adaptive language which will monitor, adapt and refine the power usage of the location and subsequent environment.

A relevant scenario in which energy-awareness is vital is to address the issues faced within the older community living within their own homes. Fuel poverty in the UK is defined as low income households where fuel (Gas and Electricity) bills cost more than the national average or the amount spent on Gas and Electricity would mean any income that is left would make them fall below the UK official poverty line [19]. Therefore, fuel poverty is a major concern for older people within the UK as they live on limited funds [19]. In the UK, over 40,000 people died unnecessarily during the 2014/15 winter period due to fuel poverty whereby older members of the community have reduced and rationed their energy use by turning off or limiting sources of heating because of high energy bills [20]. This anxiety and fear resulting from fuel poverty is a relevant and pressing social problem. Tackling fuel poverty also improves the living standards and conditions for people with low income, results in fewer winter deaths and reduces cost for the National Health Service (NHS) [21].

Lantern positions itself well with being able to address the issues relating to fuel poverty and the anxiety it can cause with older people. However, Lantern is not designed primarily for the purpose of the case study but is instead addressing the problem of energy-awareness and control of distributed devices. The case study does show an application of the language which could address a related real-world energy usage problem. This does not address issues regarding anxiety or distrust of older people with technology. However, this does complement existing work on how these systems can be used by these types of users [22]. Lantern can monitor the energy consumption of devices, coupled with sensor streams providing contextual environmental information can help manage devices within a suitably smart home. A simple example would be the monitoring and turning on or off heating appliances within the home based on the activity and location of the individual. A somewhat contrived example could be where an individual has left the fire on within the living room and then has gone to sleep (either within the same room or a different one). If they had moved, telemetry will be generated based on their current location which would result in the fire being turned off. Alternatively, if they fall asleep in the chair the system can maintain an ambient temperature until they have awoken. However, this does not fully embrace the notion of energy awareness advocated by the authors that all areas of energy reduction need to be considered.

It is not until considering more complex situations in which reducing their overall energy consumption and habits can a benefit be seen. In this case, by monitoring the energy use and including a recommender component, Lantern can have an impact on the behaviour of the older person so that they are more confident in their actions having a more positive effect on their energy expenditure. The recommender system can also link into tariff rates of energy suppliers and determine more optimal times to take advantage of cheaper energy and recommend to the user to change their times or habits. For example, cooking their lunch using an oven or microwave, kettle to boil water or water heating can be affected by the time of day. Even by changing their habits by a few minutes can result in a cumulative effect where financial savings can be made over a period of time. Lantern underpins this but in addition, can provide the monitoring and adaptive awareness to reduce the energy consumption for their entire home as a whole.

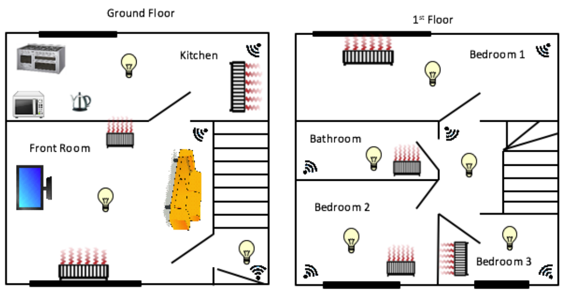

Fig. 1. An example home

Figure 1 illustrates the home of an elderly person. The home has two floors; a ground floor consisting of kitchen, front room and stairs, and a first floor consisting of 3 bedrooms, a bathroom and stairs where energy usage will be monitored and responses performed.

Therefore, this will expose developers to a language which allows them to express programs in terms of how energy-aware they are and how to adapt to changing energy demands. The approach taken has been to design a language and middleware around the notion of energy consumption and awareness rather than be added as an extension to an existing language. Lantern allows the control of a set of intelligent devices by making them energy-aware and adaptive within an environment. Building systems to control ad hoc devices within environments is not new, but the focus on building the language and middleware around the notion of energy consumption, saving and awareness is novel. The Lantern compiler (which is written in C++) translates Lantern code into Java which is then compiled and linked in to the Lantern Middleware (which is currently implemented in Java).

The middleware that has been developed provides just enough functionality to support the language. This is because the Lantern system is an exploration into representing energy-aware concepts within a language and to see what things need to be considered for developing a more encompassing language and middleware. See section 4 for more details.

Interfacing with devices (hardware) take place through the use of proxy services (written in Java) which exposes the capabilities of the underlying hardware to the Lantern Middleware. These boundary based proxy services will be found and accessed through constructs of the language and expose developers to functionality on offer from the device.

Lantern provides a logical and intuitive set of language constructs in which developers can build systems to control, coordinate and adapt to energy requirements within an environment. The language introduces the concept of energy-awareness in its make up but there are other existing methods in which software can be designed and implemented. For example, SysML [23] and MARTE [24] provide designers with a rich UML based modelling language to design and produce software for embedded devices. However, these methods do introduce problems regarding the amount of time, complexity, terminology and constructs required to design and produce a sophisticated system. Lantern on the other hand provides a more intuitive language to express the requirements of the developer and provides them with hierarchical based approach to controlling the real-time energy requirements of devices within the Lantern environment. This is based on the language being created from the ground up with notions of energy and control within its syntax. Section 4 discusses this further by focussing on the expressiveness of the language within the problem area. Another area in which Lantern provides benefits is in the dynamic reconfiguration and coordination of devices through the use of aliases and binding variables which protects the developer from the low-level intrinsic detail of the composition and linkage of relevant devices. This is opposed to the aforementioned approaches where timing and coordination can pose problems. Hence, this approach provides a more natural way for developing these types of systems.

3.1. Key language constructs

The notion of energy-awareness is represented through the use of control zones and language constructs to manage and adapt resources with those zones depending on individual's energy needs. The central concepts and notions of energy consumption, awareness and usage are modelled in both aspects of the Lantern ecosystem (i.e. language and middleware) while the compositional and adaptive nature provided by Lantern is implemented as part of the middleware. Only those concepts that are exposed within Lantern are usable by developers. An overview of the hierarchical nature of the constructs and how they fit together with other concepts are illustrated in Figure 2(a).

The central design criteria for the language was to take a low-energy/energy-aware approach to programming language design. In this case, *environments* allow direct mapping to physical buildings while areas located within these buildings can be linked to *locations*. Once these physical mappings have been defined within the language, the power use of devices and smart devices can be monitored, maintained and adapted based on what instructions the end-user provides. The language can adapt and react based on environmental state information and user generated complex events within the system. The decision to take this direction was because of the nature of the exploratory work of the Lantern system. Limiting the domain to statically located devices rather than mobile physical devices was to ensure that a core language could be developed. This is being built upon in another iterative development cycle which includes support for mobile systems and provides a more expressive language.

A similar approach has been taken to other languages where a compiler has been developed which will compile and then link each generated service or user-generated control program into the middleware [3]. Figure 2(b) shows how the language relates to other system components.

User generated services and programs are translated into Java and compiled and then linked into the middleware as service agents. These can be either *control agents* (i.e. the user generated programs for controlling the power utilisation of an area) or *system agents* which control devices and allow them to interface with the middleware.

The main components that have been developed in this phase are the compiler, middleware and device

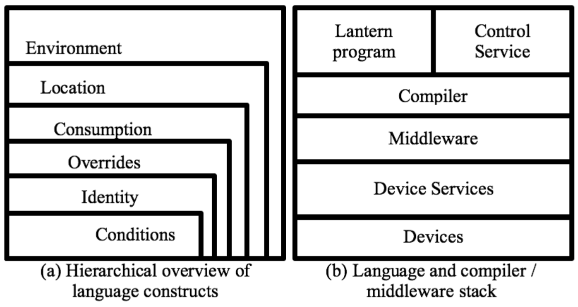

Fig. 2. An overview of Lantern

services. The middleware layer is to provide support to test the ideas behind the language rather than providing a fully rich middleware. By evaluating the language, and determining what works well will inform the next language iteration.

3.2. Environment and Location

The *environment* construct is to provide a top-level container definition for a set of defined sub-locations within an area. Multiple environments can be defined which provide configuration, adaptation, control and accessibility to the devices which are on offer within the sub-locations defined within them. These locations could cover a wide variety of different types and smartness levels. As each location can provide access to a number of devices which exposes telemetry and capabilities to the middleware, the linkage of what is available would be defined within the *location* construct. This would allow the association between an area and a number of sub-locations where the end-user would be exposed to the conditional and adaptive nature of the system for controlling their power consumption. The syntax of the *environment* construct is shown as follows.

```
[ environment( <area> ): {
[location( <internal location> ): { [devices]* }]*
} ]*
```

The *environment* construct will, when used in context of the rest of the Lantern description, provide the constraints on the environment by determining how to manage the effective use of energy within the enclosed area. The *location* construct is used to define one or more sub-areas within an environment. The purpose of the *location* is to allow the linkage and association of controllable smart devices within that sub-area to be identified and linked into the middleware. Once linked in, telemetry and the control of the devices can then be maintained. The syntax of the *location* construct is as follows.

```
location( <room/location name> ) : {
[ uses device( <device name> ) <—
input( <telemetry type> ) ]*
|
[ uses device( <device name> ) —> output( <device id> ) ]*
}
```

Devices allow the connectivity of smart objects to the environment. Devices defined here are place holder references that can be used within *condition* constructs and allows the linkage of device variables to receive input (i.e. telemetry) or as a means to control by associating them to corresponding *aliases*. This provides an interface for not only receiving telemetry data that can be generated from the device but also act as a conduit in the control and use of the device. The *input* keyword is used to allow the capture of a type of telemetry on offer by the device while the device connector construct −> or <− provides the direction of flow of information. In the case of receiving information, the device connector specifies the direction coupled with the type of data which is to be associated within the middleware. The *device* keyword allows a device to be created and maintained and any data or control requirements will be accessible under the device variable name that has been specified. Therefore, any environmental readings generated by this device will be internally represented, accessed and available through the associated variable name. Information can flow from a device to the middleware but control can also be maintained on the device by using the −> and *output* constructs. This provides the middleware with the ability to regulate whether the device is turned on or off.

Figure 3 illustrates locations in the scenario where the *kitchen*, *front room* and *bedroom* have been defined along with a set of devices and aliases linking to existing sensors and devices.

3.3. Consumption and Override

The *consumption* construct allows the developer to define the consumption rules for a given location based on how much energy is being consumed. Rules can be provided that allow the customisation and adaptation of areas based on environmental factors and stimuli. The *consumption* construct acts as a container unit which binds together all actions that need to be taken in an area. For instance, the

```
environment(house):{
location(front-room) : {
uses device( heating ) <- input(temperature)
uses device( heating ) -> output(temperature-control)
uses device( tv ) <- input( power1 )
uses device( tv ) -> output( power1-control )
uses device( lights ) <- input( power3 )
uses device( lights ) -> output( power3-control )
uses device( movement ) <- input( movement )
}
location(bedroom) : {
uses device( bedroom-lights ) <- input( bedroom-lights)
uses device( bedroom-lights ) -> output( bedroom-lights-control)
uses device( movement ) <- input( bedroom-motion)
uses device( bedroom-heating) <- input( bedroom-heating)
uses device( bedroom-heating ) -> output( bedroom-heating-control)
}
location(kitchen) : {
uses device( kitchen-cooker ) <- input( cookerpower )
uses device( kitchen-cooker ) -> output( cookerpower-control )
uses device( kitchen-kettle ) <- input( kettlepower )
uses device( kitchen-kettle ) -> output( kettlepower-control )
uses device( kitchen-lights ) <- input(kitchen-lights)
uses device( kitchen-lights ) -> output(kitchen-lights-control)
uses device( kitchen-movement ) <- input(kitchen-motion)
}
}
```

Fig. 3. The environment of the home

consumption of energy within an area (i.e. the environment) can be controlled so that if it exceeds an energy threshold, the language and middleware will re-configure devices appropriately to reduce their total energy usage so that it falls below this threshold. The syntax of the *consumption* construct is as follows.

```
consumption( <environment> ):{
[ override( <location> > <energy threshold> )
-> { /* ... */ } ]*
}
```

Introducing a consumption concept into the language allows for the monitoring and maintenance of the total energy use within the given environment. When defining the use of energy, a set of *overrides* are specified which allow fine grained energy usage monitoring and control of the locations within the environment. For example, a building could have a number of locations defined within it. When coupled with the override construct, the consumption levels of each of these locations can be monitored and adapted depending on environmental stimuli and user generated rules governing the forcible reduction of energy usage of devices.

When defining how to respond to the changing power usage within a location, the *override* construct is used to provide the fine tuning of how to handle the devices within the location which can impact environmental changes in state. Overrides allow the middleware and language to react to a stimuli or condition occurring which results in some form of intervening reaction. Each *override* definition identifies the location and how much the energy consumption threshold is for the area. Lantern will automatically calculate the total power usage for the location by using the associated devices specified by the user in the corresponding *location* construct. As each of these associated devices will be able to provide power usage telemetry as well as environmental state data, the associated location variable will be able to provide a snapshot on the current energy usage for its domain. This energy level is consumed by the corresponding *override* statement which executes a number of conditional checks to allow the adaptive reduction in energy consumption. These overrides only come into play once the threshold has been exceed, in most cases, these will not be performed and normal reconfiguration and use of the environment outlined in the identity construct would take precedent. Only when there is an exceptional spike in energy consumption would the consumption overrides start altering the power usage and availability of associated devices within the location, thereby overriding and ignoring any user specific conditions for normal use. The syntax of the override construct is as follows.

```
[ override( <location> > <energy threshold> )
-> { /* ... /* } ]*
```

When used in conjunction with the *consumption* construct, a set of *overrides* can be configured which allows a variety of scenarios where different energy thresholds can be encountered. The effect or resulting action to take place on the firing of an override clause is specified using a set of *condition* constructs which can allow the dynamic reconfiguration of devices based on the current environmental state.

The *consumption* section defines what needs to be done for the *house* environment when power usage exceeds a pre-defined amount. For example, the *kitchen* override specifies how many Watt's need to be consumed before which the associated *override* kicks in and adapts the power utilisation within that area. In this case, a condition for checking if the microwave and cooker are on then to reduce the power provided to the cooker by 30%. Another condition is to check to see if the microwave, cooker and kettle are on at the same time and to perform multiple actions as a result (i.e. reduce the cooker power by 20% and set the buzzer). It is through these overrides that the location will adapt itself so that power is reduced until a later time when additional conditions can re-configure the power usage of devices. In this iteration of the language, a weak notion of power has been adopted (where everything is in Watt's). By introducing % of power to control devices, allows reducing power needs using a simple representation. However, in practice, this was found to be not powerful enough (see section 4.4) and the next version of the language includes more strongly typed energy, temperature and time representations.

3.4. Identity

User identity is represented within the language using the *identity* construct. The notion of identity is weakly represented within the middleware itself as users are equipped with RFID and NFC badges/tags as well as smartcard swipe access. This loose notion of identity does lead to a number of challenges which still need to be addressed within the middleware. For instance, the issues relating to identity conflict and identity resolution where more than one person is contained within a sub-location such as a shared office. However, from a language point of view, it takes an agnostic approach where users can specify how they want devices and the environment to react based on a set of conditions describing what to do in specific cases. By allowing the language to have an opaque notion of identity, the issues revolving around how to coordinate and determine the precedence of conditions for multiple users in the same area can be left to the middleware.

The language definition of *identity* allows users to specify how they wish the environment (i.e. the sub-location) to respond to both system-based stimuli as well as more complex aggregate conditionals which can occur over time. The following shows the syntax for specifying an identity.

```
[ (identity:<user identity>):(location: <location> ) {
/* ... */
} ]*
```

Identities are associated to an end-user which is maintained and monitored by the middleware when they move between locations. Each user can specify any number of details regarding how they want a specific location to react. This is specified by linking the identity to a specific location and then defining the state changes that need to be monitored and reacted to. For example, in Figure 6, an identity for the user *bill* has been defined and linked to each location in the home. Within each room, there are a number of smart devices as well as devices controlling the ambient temperature and light levels. In this particular scenario, the user has specified that the heating is required to be turned on at 7.30am in the morning and then specifies two additional conditional statements which dictate when to turn the heating on or off thereafter to ensure that the ambient temperature of the room is maintained.

3.5. Conditions

Conditional statements provide a degree of flexibility in the minute-to-minute monitoring and adaption of an environment. They are associated to two constructs (*override* and *identity*) which allows the fine grained control of smart devices within a particular location or environment.

Conditions follow a simple *cause* and *effect* relationship whereby any condition that evaluates to *true* will perform the associated effect, or *action*. As events are generated and propagated throughout the middleware, a snapshot of the state is maintained for each location. This allows state information to form over time rather than waiting for the off chance that multiple events and compound events are received at the same instant in time. Conditions access this location specific state information provided by the middleware to determine the validity of the statement.

```
consumption(house):{
override( front—room > 300 ) —> {
condition( heating > high ) —> action( heating = off )
condition( lights == on ) and condition( !movement ) —> action( lights = off )
condition( tv == on ) and condition( !movement ) —> action( tv = off )
}
override( bedroom > 100 ) —> {
condition( ! bedroom—movement ) and condition(bedroom—lights == on ) —> action(bedroom—lights = off )
}
override( kitchen > 1000 ) —> {
condition(kitchen—microwave == on ) and condition(kitchen—cooker == on ) —> action( reduce(kitchen—cooker —> 30%) )
condition(kitchen—microwave == on ) and condition(kitchen—cooker == on ) and condition(kitchen—kettle == on ) —>
action( reduce(kitchen—cooker —> 20% ) ) and action( buzzerwarning = on )
}
}
```

Fig. 4. The Consumption of the home

In its simplest form the *condition* construct syntax is in Figure 5.

Conditions can be used as a single instance or part of a more complex aggregate of conditions. The conditional predicate of the statement either evaluates to *true* or *false* but can be formed out of more complex combinations of conditionals using standard *and*, *or* and *not* operators. It is only once all conditionals evaluate to true will the associated action be performed. Additional keywords and concepts are available to be used within the formation of conditions to expand on their expressiveness. For example, the definition of user types which are of a given type and associated to a form of telemetry from a device can allow a degree of flexibility in the terms used in their construction.

Events are used to transmit information around the middleware and are used as a basis to populate state information regarding the devices within a given location. The middleware maintains this state and provides the ability to form unbounded time-based event states. This means that multiple events can be received in a short time whereby they are deemed to have a causal relationship with one another. Building upon this is the notion that user generated events can be triggered based on a set of expected events and can be used in the composition of more complex states. In order to create a user generated event, the *new-event* keyword is used which will relay it to the middleware layer and is based on a *key-data* tuple pair. For example, in the following, a new user generated event is created once the condition has been received.

```
condition( temperature > 20 ) —>
action( new—event( "stable—temp", temperature ,10 ) )
```

In this case, the *action* is triggered which allows the *new-event* keyword to be used to generate a message with data that needs to be sent. By adding the ability to create new user generated events allows the language to be extendible and react to other non-system related events generated by devices.

Direct manipulation of devices can also be achieved through the *action* construct by specifying directly the state that device needs to be in. For instance, by explicitly setting a device to *on* will direct the middleware to interact with the device and change its state. This is accomplished within the middleware through the use of services residing on individual devices which provide a way of manipulating and controlling the underlying device, as illustrated in Figure 6. In this example, the device associated to one of the heating variables can allow the direct manipulation of its state. However, the comparison of the state/value contained within this variable is checked against a user-generated typed variable linked to a specific device and sets a specific threshold. The language also allows for comparison against a statically defined numeric value as well. In this example, the *at* keyword is used which allows the condition to trigger based on a temporal event, in this case as soon as the system time becomes 7.30, the condition will be triggered.

Conditions can use the standard set of comparison operators as found in other languages, namely, equality (==) , greater (>), less than (<) and not (!). Conditions can be formed by any combination of *and*, *or*, *not* conjunctions and allows the precedence ordering of combinations. Power handling can also be accomplished by using the *reduce* keyword in the *action* statement. The purpose of this is to allow any suitably equipped smart device which can interact with a physical device (for example a cooker or heater

```
condition:: condition( <clause> ) [ join−operator <condition> ]∗ −> action | [ join−operator action]∗
clause:: <variable | now | at> comparison−operator < variable | time | integer >
action:: action( <perform−action> )
join−operator:: and | or | not
comparison−operator:: > | < | ! | == | =
perform−action:: set−variable | reduce | new−event | notify−message
```

Fig. 5. Conditions

unit) to reduce its power consumption by a given amount - as illustrated in the kitchen location of Figure 6

3.6. Aliases

The final key component of the language is providing the semantics for linking telemetry streams and power control functionality to variables which can be manipulated within the user defined program. *Aliases* are used to allow variables to be bound to real-world objects and sensor streams using a *proxy* pattern. Figure 7 highlights the syntax of the different styles of aliases that can be defined.

A *binding-alias* allows the connection of a user defined binding variable to be associated to a real-world device. The declared binding variable is then used as a conduit through which information and control can flow depending on the type of connectivity (i.e. −> implies that information can flow from system to device, while <− implies information flowing from device to system). Each binding variable provides state (either in the form of telemetry that has been provided) and timing information for use when handling events or combinations of complex events. As binding-alias's provide contextual information from a device they are considered *bounded* variables whereby an association is defined and maintained between construct and device. Information flow is dictated by the connectivity operator which defines which direction information flows. For binding aliases which require telemetry to be received, the *input-connector* is used. However, if control of the device is required, an additional binding-variable has to be created to handle control messages and is used in conjunction with the *output-connector* when associating the binding variable with a device.

The second type of alias is a *declaration-alias* which allows users to define their own variables with a specific integer value or threshold. These types of aliases can be bound or un-bound to a device depending on the context that the variable is to be used. For instance, an alias could be defined thusly, `alias(low : integer) = define (20).` In this case, the variable simply is assigned a static value which can be accessed throughout the programs lifetime. However, as it is *un-bounded*, i.e. it is not linked to a particular device, no contextual information is required or used to determine if the variable is being used in the correct context with regard to a device. If it were a *bounded* variable, an explicit link to a device would be given. For example, `alias(low : integer) = define (20) <− heating`, in which *heating* is an alias connected to a device can be used (of type heating). Here, as context is provided by the linkage between this static variable declaration and a device (through another alias definition / proxy to a device) checks can be made to ensure that the type of telemetry data matches with the type definition of the variable.

When binding a device to a variable or interfacing any hardware, contextual information regarding the capabilities and offerings of a system-based service representing the physical device (hardware) is needed to be specified. As system agents act as proxies to real-world devices, they are programmed to offer a set of capabilities. It is through these capabilities that the resolution and binding process takes place between a binding variable and associated device (hardware). For example, when defining a system device, the capabilities would follow on from previous work [3] in which each device can specify what their capabilities are when defining the service. Figure 8 illustrates an example of this.

Here, the *device-type* acts as type identifier for the classification of the device. If telemetry is generated by the device, the service will have a *telemetry-type* definition which defines the type of information which is generated. If a device is *controllable*, i.e. it can be manipulated by turning on or off, or in some cases reducing the power consumed to a lower amount, the controllable identifier will either be *yes* or *no*. Associated to whether it can be controlled, the *controllable-state* identifier is used to indicate what type of actions can be performed. In this example, *boolean* is specified to indicate that it can be either *on*

```
(identity:bill):(location:front-room) {
condition( at(7:30) ) -> action( heating = on )
condition( temperature < temperature_threshold ) and condition( now > time(07:30) ) and
condition( now < time(20:00) ) -> action( heating = on )
condition( temperature > temperature_threshold ) -> action( heating = off )
condition( lights == off ) and condition (movement ) -> action( lights = on)
condition( at(20:00) ) and condition( !movement ) -> action( lights = off )
condition( at(20:00) ) and condition( !movement ) -> action( heating = off )
condition( at(20:00) ) and condition( tv == on ) and condition( !movement ) -> action( tv = off )
condition( !movement ) -> action( heating = off ) and action( lights = off )
}
(identity:bill):(location:bedroom) {
condition( at(20:00) ) -> action( bedroom-heating = on )
condition( bedroom-temperature < bedroom_temperature_threshold ) and condition( now > time(20:00) ) and
condition (now < time( 08:00) )-> action(bedroom-heating = on )
condition( bedroom-temperature > bedroom_temperature_threshold ) -> action(bedroom-heating = off )
condition( bedroom-lights == off ) and condition (bedroom-movement ) -> action(bedroom-lights = on)
condition(bedroom-lights == on ) and condition (!bedroom-movement ) -> action(bedroom-lights = off)
condition( at(7:30) ) and condition( !bedroom-movement ) -> action(bedroom-heating = off )
}
(identity:bill):(location:kitchen) {
condition( kitchen-lights == off ) and condition (kitchen-movement ) -> action(kitchen-lights = on)
condition( kitchen-lights == on ) and condition (!kitchen-movement ) -> action(kitchen-lights = off)
condition( kitchen-cooker == on ) and condition (!kitchen-movement ) -> action(kitchen-cooker = off)
condition( kitchen-kettle == on ) and condition (!kitchen-movement ) -> action(kitchen-kettle = off)
recommend( condition( kitchen-kettle == on ), now, 10 ) -> action( notify-message(''Kettle should be used at a better time''))
}
```

Fig. 6. Example identity and conditions for a location

```
alias:: <binding-alias> | < declaration-alias>
binding-alias::alias( <binding-variable> ) <input-connector | output-connector> <device>
declaration-alias:: alias( <variable : variable-type> ) = define ( <integer> ) [ <input-connector> <device> ]
device:: [(] device-type [, device-capabilities ]* [)]
output-connector:: ->
input-connector:: <-
```

Fig. 7. Aliases

```
aliases {
alias( temperature-control ) -> device ( temperature , front-room)
alias( temperature ) <- device ( temperature , front-room)
alias( temperature_threshold : integer ) = define(21) <- temperature
# ...
}
```

Fig. 8. Corresponding capabilities list in device system agent

or *off*. More sophisticated devices which can offer a variable control over the device, i.e. reduce its power consumption by offering more states, would link to the *reduce* construct. The *location* identifier allows the device to indicate where it is located as searching for "heaters" could potentially return a set containing all heaters rather than one specific one. To help in the resolution process, key-data pairs can be used to help resolve to a single device.

3.7. Recommendations

Beyond the key constructs introduced in the preceding sub-sections, Lantern incorporates a rudimentary recommender system. The recommender allows processing of tariff information which is provided by utilising information from energy suppliers websites. At present this is done by hand and is stored locally within the system. The purpose of the recommender is to allow the system to offer suggestions to the end-user and inform them if performing the action at an alternative time could provide some cost savings. The *time-window* parameter specifies how much time to consider before and after the current time. If a better alternative has been found within this window, then a recommendation will be made. This language feature has been added to complement other work that the authors are undertaking in looking into ways of reducing the energy footprint of an individual within their home.

The recommend construct is triggered when a *condition* statement or composite set of conditional statements evaluates to true. If this is the case, the associated *action* is performed which can allow either a direct notification to be passed to the user or by generating another event. The basic syntax of the recommend key word is as follows.

```
recommend( <condition> [ and <condition>]*, now, time—window ) —>
action | [ join—operator action] *
```

An example use of the recommender and how it can influence the system is as follows.

```
recommend( condition( kettle == on )
and condition( movement == true)
and condition(oven == on), now ) —>
action( new—event("too—much", true) ) and
condition( too—much == true ) —> action( reduce( oven —> 50% )
```

3.8. Energy-aware software compositions

To demonstrate the usefulness and applicability of the language a supporting middleware has been developed based on previous work [3] to support energy-aware software compositions. The middleware provides control, communication and monitoring capabilities of connected devices and user defined programs for controlling and adapting the power usage of the language. Figure 2(b) provides an overview of the hierarchical nature of the Lantern Stack. The Lantern compiler was developed under C++ and compiles Lantern code into Java which is subsequently compiled to a set of Java files and linked into the Lantern Middleware. The middleware was developed using Java to enable portability and availability of the middleware on different platforms.

Constructs within the language provide exposure to hardware devices that control what functionality is on offer to the middleware. This is through a set of proxy devices which act as an interface between the middleware and hardware. In addition, the middleware handles the control, coordination, adaptation, discovery/resolution, monitoring and environmental state of all devices and services running within the distributed middleware.

Devices (hardware) represent the physical smart assets which have been subsumed within physical locations. To interface with these devices, two types of software agents are used which act as proxies between the physical and middleware representations of devices. The *control agent* provides a conduit for controlling the given device by offering a set of capabilities which describe the type, location and controllable aspects of the device. In addition they allow any user generated contextual information to help in the discovery and resolution process as well as allow the middleware to control the operation of the device. There are three modes that are supported, *on*, *off* and *reduce* which align with system concepts present within the language. Conversely, *System agents* provide telemetry from the device. Each service representation only allows information to flow in a single direction: *control information flows to the device* while *telemetry flows from a device*. Keeping these separate allows a simpler model for representing the devices so that they align with their corresponding Lantern constructs (in this case the *uses* and *input* or *output* connectors). One or more environments can be defined which act as zones of control and map to real-world notions of a physical environment such as a building. Inside each environment, one or more locations can be defined which represent physical locations within a building. Inside of these locations, devices will be found which can allow for the transmission or manipulation of state.

When writing system or control agents, contextual information is used to define their type along with their capabilities. This information is used in the lookup / discovery process by which variables are created within the language and then dynamically bound to an associated service at run-time.

Lantern allows the developer to write systems which uses some of the principles from Design by Contract. For example, the use of a similar notion to those of assertions through *condition* and *alias* variables. These are used to specify condition matching through the use of late dynamic binding variables for the matching of devices to services. Verification is handled in a number of ways by providing syntax and type verification. For example with user variables or conditions, if the variable is not linked to the correct type of device, an error will occur at run-time. This allows for a stronger type association to the condition or alias to ensure that the correct context is used, or identify an ambiguity if more than one variable is defined but linked to another device. In addition, Lantern also supports the developer with contextual associations between the main constructs so that services can be written which verify the presence of devices within the context of their use. This ensures that a logical set of abstractions of devices used within strongly defined areas. Verification of systems is an extremely important issue in the development of robust, adaptive software. In future versions of the language this issue will be addressed more deeply.

The middleware also provides the coordination and control functionality to maintain all devices located within environments. State information which is transmitted as telemetry event messages are passed throughout the middleware layer. These event messages are exposed to Lantern and forms the basis of triggering corresponding conditional statements. As events are bound to time, location and device, the middleware layer provides a comprehensive way of providing state information which can be accessed and used within the language. State information can represent one or more device states / telemetry as well as offering the ability to form complex user generated conditional statements where events can occur over a given time rather than at the same instant.

Lantern allows users to generate programs based on their requirements for an environment and how to control and adapt to changing demands of power utilisation. These programs are compiled by a compiler which has been implemented to generate agents which work on behalf of users. At present, this is controlled by a single instance whereby all environments are maintained by a single top-level management agent which controls all requirements specified within each of the abstractions. However, this is to be expanded upon to fit in with the rest of the model of the middleware whereby each will be represented by a variety of agents.

4. Evaluation

The central purpose of this paper is to introduce the Lantern language for developing energy-aware systems. To evaluate the language, a set of metrics need to be considered which look at the *language comparisons*, *expressivness*, and *energy-awareness*. Therefore, the metrics used intend to highlight:

- *comparison*: this looks at high-level aspects of Lantern versus C++ to provide an indication of some of the differences between an energy-aware and a general-purpose language.
- *expressiveness*: this requires looking at concepts and notions within the language and will inform a comparison of language constructs to base an evaluation on.
- *energy-aware*: this focuses on energy-aware concepts and structures which facilitate in energy-aware programming notions.

4.1. Language comparison

The purpose of the Lantern language is to address whether energy-awareness can be expressed within a language rather than as an addon to an existing language. Because of this, Lantern is feature rich when considering how it can adapt to changing needs within an environment and facilitate the binding, and reconfiguration of connections to other agents and devices. Table 1 shows a high-level comparison between Lantern and C++. Other languages could have been used for this comparison but C++ has good constructs like object-oriented, exceptions and templates suitable for system development, and thus, a reasonable choice for comparison.

Table 1
Language comparisons

Metric	**Lantern**	**C++**
Extendable	Y	Y
Object-oriented	N	Y
General purpose	N	Y
Energy-aware ready	Y	N
Environment representation	Y	N
Location representation	Y	N
Consumption requirements	Y	N
Power telemetry	Y	N
Power representation	Y	N
Adaptive power needs	Y	N
Device Discovery and Binding	Y	N
Device re-discovery and configuration	Y	N
Event based	Y	N
Type based	Partial	Strong
Inheritance	N	Multiple

This is through the nested constructs dealing with the *environment* and *location* and how different rules can be attributed to areas. These constructs allow for the initial bind and subsequent re-binding and discovery of agents within the middleware. Primarily, adapting to changing energy demands requires interpreting and reacting to telemetry generated through service agents interfacing with physical devices. Constructs that deal with initial discovery (or devices for example) are the *alias*, *device* , *input* and *output* constructs. When interpreting and responding to changes, the constructs dealing with *override*, *condition* and *action* enable altering and adapting the environment based on environmental state information. Higher-level constructs, such as *environment* and *location* enable higher-order configuration and adaption details regarding the physical smart space being controlled.

4.2. Expressivness

When determining the expressivness of a language, it can be subjective when coming up with metrics which can be used. The approach that has been taken is to use a scenario-based method whereby three scenarios have been created and compiled to show the expressivness of the language when modelling these situations. The first scenario models a smart house (as introduced in the example used in this paper). The second models a smart office environment, while the third scenario is more specialised and is used to control the energy needs for an older person to help them make reduce their energy footprint.

For each of these scenarios, a comparison is made between the constructs and key language notions used. Table 2 compares each scenario.

From this it is possible to observe the following. Although the scenarios were programmed using Lantern, it can be seen that the amount of keywords and lines of code to produce each scenario highlights how little code is needed. No C++ equivalent was produced to directly compare against but it can be assumed that more code would need to be produced to provide for a similar level of functionality.

The lantern language provides specialised constructs for the linkage to physical devices, discovery, and subsequent binding to devices located within a defined area. By providing structures to represent an environment and sub-locations within it provides a clean way of grouping requirements and power adaptability together.

Table 2

Expressivness comparisons

Metric	**Scenario 1**	**Scenario 2**	**Scenario 3**
Lines	119	66	140
Blank Lines	4	4	7
Comment Lines	1	2	5
Environments	1	1	1
Total keywords	692	366	886
Locations	3	1	2
Consumptions	1	1	1
Overrides	3	1	2
Override Conditions	6	4	5
Aliases	20	10	21
Identities	1	1	1
Variable declarations	1	1	1
Conditions	19	9	15
Actions	19	9	15
Notifications	1	0	11
Time bound	8	5	5
Complex conditions	14	8	12
Telemetry Devices	11	5	12
Control Devices	8	3	9
Input flow	11	5	11
Output flow	8	4	9
Reductions	1	0	1
Recommendations	0	0	11

4.3. Energy-aware constructs

As Lantern is a domain specific language for expressing energy-aware concepts within a language, an important area to evaluate is the energy-aware expressiveness and how this impacts on the generation of code to adapt to changing energy needs. Table 3 shows the use of energy-aware constructs for each of the scenarios.

Table 3

Energy notion comparisons

Metric	**Scenario 1**	**Scenario 2**	**Scenario 3**
Overrides	3	1	2
Override Condition Dependancies	6	4	5
Reductions	1	0	1
Reconfigurations (based on conditions)	19	9	15
Devices controlled	8	3	9
Recommendations	0	0	11

When evaluating the language, the constructs that facilitate adapting the environmental energy use (*overrides* and *conditions* which lead to a change in the environment) vary depending on the complexity of the scenario. For instance, scenario 1 was for a smart house with several devices in a few rooms when compared with scenario 2 (a smart office) show that the complexity and number of constructs used depend on the number of devices available in the environment. The more complex the environment and higher number of devices, translates into a greater number of *conditions* and *overrides* used overall within the system.

Scenario 3 models a house which has a couple of rooms to notify older users of when to do things. Specifically, this is recommending when a better time to perform the user action should take place to help improve the energy use within the home. This pulls in the recommender component of the system which uses tariff information relating to how much it costs to use energy from a supplier at different times of the day. Used in conjunction with conditions allows the scenario system to offer suggestions. Data to see if this changed the end user habits were not monitored.

Overall, introducing energy-aware constructs within the language allows producing fairly straight forward systems to adapt to energy needs.

4.4. Discussion

In this section, a comparison of the Lantern language has been provided using several scenarios to highlight the expressivness of the language. However, the intention behind the Lantern system was to investigate ways in which energy-awareness and control can be integrated into a language for coordinating cyber-physical systems with an emphasis on statically located devices and environments. This was a multi-step process where an initial language (Lantern) was first developed to explore what issues lay in providing a domain specific language for controlling the energy use of devices and services within an environment. Based on what was learnt from this initial phases, a more in-depth language (PLECO) is under development which learns from the ewillxperiences gained from Lantern. From these experiences the areas which need expanding from this first phase can be summarised as:

- Mobility: As the system was for statically located physical devices (e.g. embedded hardware IoT device integrated into the environment), there was no constructs for dealing with the possibility of devices physically moving. These mobile issues are currently being considered in the current iteration of the language and how this impacts on the coordination and control.
- Energy: The Lantern system provides a simplified notion of energy representation within the language (i.e. how much power does an environment utilise). This is currently being expanded on to consider strongly typed energy constructs which allow for a better energy representation.
- Identity: Identity is weakly defined within Lantern and a better way of representing identity is being investigated. At present there are issues with scalability when linking users to locations which needs addressing. In this instance, a group-based membership is being developed which also allows security considerations to be considered. In addition, anonymous and the obfuscation of identities are being accommodated for.
- Improved control structures and language constructs for handling more complex interactions and reacting to non-time bound interactions.
- Providing a more advanced middleware and organisation layer for developing and managing complex environments.

Another area of focus in the next iteration of Lantern is to look at scalability and optimisation. Comprehensive benchmarking will be performed to determine the effectiveness of Lantern against other approaches. This will consider language-based metrics (lines, man hours, keywords, etc.) along with middleware metrics (adaptation, connectivity, discovery, etc.). These as well as other enhancements and considerations are being considered in the next iteration phase of the system called PLECO [25].

5. Conclusion and future work

This paper has argued that designing energy-aware languages from first principles is a better way for introducing the concepts of low-carbon usage than through existing approaches of integrating systems into existing languages. The approach of exploring concepts through successive exploratory energy-aware languages has been taken. The initial iteration, Lantern, has introduced a language and middleware which has been outlined within this paper whereby the approach to energy-awareness and efficiency has been through the design and production of a language where the central notion of reducing energy has informed its design. This has resulted in an energy-aware language where developers are exposed to language constructs and processes aimed at providing an efficient way for managing resources within an environment. As an iterative approach has been taken, a number of areas have been highlighted in this cycle which informs the next iteration.

For instance, one such expansion would be to integrate a more powerful recommender system and an activity/behaviour identification component which has been developed separately to the main Lantern system [22]. This will enable the system to be more powerful and be able to expose the developer to external tariffs and activity identification which can have a bearing on the efficiency of managing a static environment.

Another area of work is on identity resolution and conflict management. At present the Lantern system has been based on the assumption of a single occupant. However, this is inadequate when faced with real-world scenarios where environments will be shared between multiple people. The middleware at present provides a fairly simple model of the world. The scalability of the approach which has been taken will be expanded upon along with more seamless integration of technology.

Lantern was the first iteration of exploring energy-aware languages; the next cycle is considering a number of areas that have been identified as requiring more work. Principally, these fall into the areas of: identity resolution; location and identity mapping; improved energy representations and other language improvements; recommending new actions; and, increasing the scalability of the infrastructure and language to accommodate mobile ad-hoc systems rather than be solely based on statically located devices. The next step is incorporating these areas into the next iteration of the language (PLECO).

References

[1] B. Walsh. “The surprisingly large energy footprint of the digital economy”, *Time Magazine*, (2015).

[2] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami. “Internet of things (iot): A vision, architectural elements, and future directions”, *Future Generation Computer Systems*, **29(7)**, pp. 1645–1660, (2013).

[3] J. Robinson, I. Wakeman, D. Chalmers. “Composing software services in the pervasive computing environment: Languages or apis?”, *Pervasive and Mobile Computing*, **4(4)**, pp. 481–505, (2008).

[4] S. Mittal. “A survey of techniques for improving energy efficiency in embedded computing systems”, *International Journal of Computer Aided Engineering and Technology*, **6(4)**, pp. 440–459, (2014).

[5] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, R. Lauwereins. “Managing dynamic concurrent tasks in embedded real-time multimedia systems”, *Proceedings of the 15th international symposium on System Synthesis*, pp. 112–119, (ACM, 2002).

[6] Y. Ma, N. Sang, W. Jiang, L. Zhang. “Feedback-controlled security-aware and energy-efficient scheduling for real-time embedded systems”, *Embedded and Multimedia Computing Technology and Service*, pp. 255–268, (Springer, 2012).

[7] M. A. Awan, S. M. Petters. “Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems”, *2011 23rd Euromicro Conference on Real-Time Systems*, pp. 92–101, (IEEE, 2011).

[8] S. Mittal, J. S. Vetter. “A survey of methods for analyzing and improving gpu energy efficiency”, *ACM Computing Surveys (CSUR)*, **47(2)**, p. 19, (2015).

[9] C. A. Björkskog, G. Jacucci, L. Gamberini, T. Nieminen, T. Mikkola, C. Torstensson, M. Bertoncini. “Energylife: pervasive energy awareness for households”, *Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing-Adjunct*, pp. 361–362, (ACM, 2010).

[10] F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer, D. S. Nikolopoulos. “Application-level energy awareness for openmp”, *International Workshop on OpenMP*, pp. 219–232, (Springer, 2015).

[11] N. Amsel, B. Tomlinson. “Green tracker: a tool for estimating the energy consumption of software”, *CHI’10 Extended Abstracts on Human Factors in Computing Systems*, pp. 3337–3342, (ACM, 2010).

[12] M. Sabharwal, A. Agrawal, G. Metri. “Enabling green it through energy-aware software”, *IT Professional*, **15(1)**, pp. 19–27, (2013).

[13] N. Nikzad, O. Chipara, W. G. Griswold. “Ape: an annotation language and middleware for energy-efficient mobile application development”, *Proceedings of the 36th International Conference on Software Engineering*, pp. 515–526, (ACM, 2014).

[14] Y. Xiao, R. S. Kalyanaraman, A. Ylä-Jääski. “Middleware for energy-awareness in mobile devices”, *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, p. 13, (ACM, 2009).

[15] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, I. Stark. *Mobile Resource Guarantees for Smart Devices*, pp. 1–26, (Springer Berlin Heidelberg, Berlin, Heidelberg, 2005), URL https://doi.org/10.1007/978-3-540-30569-9_1.

[16] D. Franzen. “Quantitative bounds on the security-critical resource consumption of javascript apps”, , (2016).

[17] G. Smith. “Updates of the itrs design cost and power models”, *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 161–165, (2014).

[18] Semiconductor Industry Association. “The international technogy roadmap for semiconductors”, URL http://www.itrs2.net, (2013).

[19] Department of Energy & Climate Change, UK Government. “Annual fuel poverty statistics report”, URL https://www.gov.uk/government/statistics/annual-fuel-poverty-statistics-report-2016, (2016).

[20] Office for National Statistics, UK Government. “Excess winter mortality in england and wales”, URL https://www.ons.gov.uk/peoplepopulationandcommunity/birthsdeathsandmarriages/deaths/bulletins/excesswintermortalityinenglandandwales/2015to2016provisionaland2014to2015final, (2016).

[21] J. Hills. “Getting the measure of fuel poverty”, *Hills Fuel Poverty Review*, (2012).

[22] J. Robinson, K. Aappiah, R. Yousaf. “Improving the well-being of older people by reducing their energy consumption through energy-aware systems”, *Proceedings of the 9th International Conference on eHealth, Telemedicine, and Social Medicine (eTELEMED 2017)*, pp. 161–165, (2017).

[23] OMG, Systems Modelling Language SysML, URL http://www.omgsysml.org, (2017)

[24] OMG, UML Profile for MARTE: Modelling and Analysis of Real-Time Embedded Systems, V1.1, URL http://www.omg.org/spec/MARTE/1.1/PDF/, (2011)

[25] J. Robinson, K. Lee, and K. Appiah, “Pleco: New energy-aware programming languages and eco-systems for the internet of things”,’ in *The Eighth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. IARIA, 2018.