Engineering Sustainability Through Language

Ruzanna Chitchyan
Department of Computer Science
University of Leicester
Leicester, UK
Email: rc256@leicester.ac.uk

Walter Cazzola
Department of Computer Science
Università degli Studi di Milano
Milan, Italy
Email: cazzola@di.unimi.it

Awais Rashid
School of Computing and Communications
Lancaster University
Lancaster, LA1 4WA, UK
Email: marash@comp.lancs.ac.uk

***Abstract*—As our understanding and care for sustainability concerns increases, so does the demand for incorporating these concerns into software. Yet, existing programming language constructs are not well-aligned with concepts of the sustainability domain. This undermines what we term *technical sustainability of the software* due to (i) increased complexity in programming of such concerns and (ii) continuous code changes to keep up with changes in (environmental, social, legal and other) sustainability-related requirements. In this paper we present a proof-of-concept approach on how technical sustainability support for new and existing concerns can be provided through flexible language-level programming. We propose to incorporate sustainability-related behaviour into programs through *micro-languages* enabling such behaviour to be updated and/or redefined as and when required.**

I. Introduction

Presently, there is no single, universally accepted definition for *technical sustainability of software*. For instance, [13] states that: "Technical sustainability refers to longevity of information, systems, and infrastructure and their adequate evolution with changing surrounding conditions". On the other hand, the ICSE-SEIS 2015 call for papers [8] states that it is "... software that can evolve, adapt to changing requirements, adapt to ever-changing society". Both these definitions use the terms "evolve" and "adapt" to subsume the numerous software engineering characteristics. These characteristics, to name a few, include: adaptability, maintenance, innovation, reuse, obsolescence, etc.

Given the present state of software engineering research and practice, no single technique can possibly address all technical sustainability issues of software. But what are the impediments to its evolution and adaptation? If we consider the difficulties faced when integrating fast changing requirements for software in society, we observe drivers of un-sustainability. Although the following drivers equally apply to other fast changing requirements, here we focus on the sustainability domain itself, i.e., environmental, social, and human sustainability issues:

- **Construct Asymmetry:** The existing programming constructs are first and foremost computational by nature leading to a disconnect between societal sustainability properties and the constructs used to operationalise them. This *asymmetry* increases programming complexity with regards to sustainability requirements (negative social impact) and the effort spent on addressing them within the software (negative economic impact).
- **Change Integration:** Sustainability requirements change over time since systems operate in constantly changing contexts. As the regulatory environment and societal understanding of sustainability issues change, these *changes need to be integrated into existing software*.
- **Emergent Concepts:** With the improved understanding of sustainability issues and continuously modified targets, *new concerns frequently come to the fore*. Incorporating these new requirements into the software is challenging. For instance, the recent EU regulation on power use for vacuum cleaners has led to full review of both the hardware and software components of these appliances.

In this paper, we present the first step towards countering these drivers – a flexible, micro-language based approach. A *micro-language* is essentially a very small language – very much like a domain-specific language (DSL) – dedicated to expression of a single aspect of a domain, such as battery life in (environmental) sustainability domain. Micro-languages can be used to (i) incorporate domain concerns, (ii) manage changes/adaptation of such concerns, and (iii) handle emergent ones that could not be foreseen at the conception stage of the software. The underlying language implementation is such, that, when the semantics of a given domain concern, and hence the micro-language, change, these can be updated without any side effects on other micro-languages. Thus, the language environment allows each language element to be updated in a localised fashion, with all the change effects absorbed into that same element. Thus, the contributions of this work to furthering technical sustainability are, twofold:

- Illustration of how concepts from the wider sustainability domain can be meaningfully represented and integrated into a new or evolving code base, and
- Demonstrating that, the improved technical sustainability, can also better support other sustainability-related behaviour (i.e., behaviour realising sustainability needs).

In the following, Section II motivates the need for micro-languages. Section III outlines the language environment we use as a proof of concept implementation. Section IV presents two sustainability scenarios and their implementation in our approach. Section V concludes with a discussion.

II. Motivating Micro-Language Use

Existing research in programming languages strives for improved modularity to limit change ripple effects and sep-

arate concerns. The limitations of object-oriented languages with regards to scattering of broadly-scoped properties are well-documented [10]. Though paradigms such as aspect-oriented programming aim to address these limitations, the noted problems of construct asymmetry and change integration cannot be readily addressed – the well-known problem of coarse granularity of pointcuts will either limit the approach to code where input and output operations are confined to specific methods or force to restructure the code to fit the granularity limit [14]. Other research focused on aligning domain and programming language concepts, e.g., [12], [15] provides a valuable starting point for research into tackling the technical sustainability issues highlighted above.

A significant research has already been done on numerous topics related to software and sustainability. Examples include: systems for minimising energy use and emissions in data centres [4], energy-efficient algorithms to minimise CPU cycles [7], tools for ultra-large-scale community engagement in design of socio-technical systems [5] and study of software impact on social sustainability [1]. However, to date, the changing and emergent nature of sustainability concerns and its impact on technical sustainability of software has not been adequately researched. Nor have there been any approaches that explicitly account for the asymmetry between sustainability concepts and programming language constructs.

Recent studies [6] suggest that nearly 80% of changes in software are evolutionary, i.e., not caused by bug fixes. Such changes arise due to new and changed requirements and necessitate re-writing of (parts of) the software code, increasing its complexity and reducing understandability. As discussed above, sustainability concerns pose many new, rapidly changing requirements. Using micro-languages to capture and codify sustainability concerns would ease the task of programming to such requirements through better cognitive alignment of the available language constructs with the programming task.

A single DSL would not be well-suited to addressing the challenges as the core concepts of the sustainability domain are still emerging. Committing to any one DSL will still leave open the challenge of frequent code re-writes as well as frequent changes to the semantics of the language elements. In order to utilise the advantage of conceptual alignment afforded by DSLs and yet avoid frequent code re-write due to evolution of the element semantics, a set of micro-languages, as envisaged in our approach, provides an effective solution. A micro-language captures semantics of a single concern and can fluently evolve as that concern does, without negatively affecting the other, related micro-languages. Thus, both changes to semantics of a concern, and hence the micro-language, as well as emergent concerns can be supported.

III. Neverlang Language Environment

To demonstrate the feasibility of our approach, we utilise the Neverlang environment [2], [3], [16] which considers language units as first-class concepts. In Neverlang, language components are developed as separate units that can be compiled and tested independently, enabling developers to share

```
module StateModule {
  reference syntax {
    State ← StateName "=" Expr
  }
  role(execution) {
    0 .{ newstate($1.state, $2.action); }.
  }
}
slice State {
  concrete syntax from StateModule
  module StateModule with role execution
}
language HooverLang {
  slices Program StateDecl EventDecl TransDecl StateList State
      StateName EventList Event EventName Expr BExpr TransList
      Transition Support
  roles syntax < execution
}
```

Listing 1: Slice syntax, semantics for the state concept and hoover language configuration.

and reuse these units across different language implementations. Here the base unit is the `module` (Listing 1). A module may contain a `syntax` definition or a semantic `role`. A role defines actions that are to be executed when some syntax is recognised, as prescribed by the *syntax-directed translation* technique. Syntax definitions are portions of BNF grammars, represented as sets of *grammar rules* or *productions*. Semantic actions are defined as code snippets that refer nonterminals in the grammar. Syntax definitions and semantic roles are tied together using `slices`. Thus, module `StateModule` in Listing 1 declares a reference syntax for the state concept (used by the Hoover example in Section IV-1) and actions are attached to the nonterminals on the right of the production. Semantic actions are attached to nonterminals by referring to their position in the grammar: numbering starts with 0 from the top left to the bottom right, so the first `State` is referred to as 0, `StateName` as 1, and the `Expr` is referred as 2. The slice `State` declares that we will be using *this* syntax (which is the `concrete syntax`) in our language, with those particular semantics. Finally, the `language` descriptor (Listing 1) indicates which slices are to be composed together to generate the language interpreter or the compiler. Composition in Neverlang is, therefore, twofold: (1) between modules, which yields slices, and (2) between slices, which yields a language implementation. The composition result is independent of the order of specified slices. The grammars are merged to generate the complete language parser. Semantic actions are performed with respect to the parse tree of the input program; roles are executed in the order specified in the `roles` clause of the `language` descriptor. Please see [3] for a more detailed description of Neverlang's syntax. Although we use Neverlang, there are also other frameworks able to support seamless sustainability at the language level, e.g., Lisa [11], StrategoXT/Spoofax [9] and Silver [17]. Lisa provides an inheritance-based language composition mechanism that is less flexible than Neverlang's but still efficient. Spoofax has a granularity similar to that of Neverlang, but does not support

separate compilation and any extension requires recompilation of the whole compiler. Silver is functional by nature and provides flexibility similar to Neverlang.

IV. Seamless Language Support for Sustainability

We demonstrate our approach to technical sustainability via two scenarios: (i) change to micro-language semantics, and (ii) extension of a general purpose language's (GPL) semantics.

1) Technical Sustainability via Change to DSLs semantics: Consider a hoover controller designed to turn the hoover on and off when the switch is turned to "ON"/ "OFF" respectively. A new requirement is added later stating that the controller should switch the hoover off when it is not in use (e.g., when accidentally switched on by a toddler), to reduce energy waste and emissions. When in use the hoover should be in motion; if it is not in motion for 10 sec., it is considered not in use. Instead of having to modify the controller program with condition checks for motion monitoring, we propose to initially implement and then re-define the semantics of the "turn on" construct through a micro-language. This enables realising the newly added requirement, leaving the controller code unchanged.

The behaviour of a hoover (or any other electrical household appliance) can be easily modelled with a state-machine: it has some feasible states (e.g., *on*, *off*, etc.) and some transitions that, under given events (e.g., clicks on a button), move the appliance from one feasible state to another. Such behaviour can be implemented using a simple event-based micro-language as in the code snippet in Listing 2.

```
states
  on  = turn_on() ;
  off = turn_off()
events
  click = get_click()
transitions
  on  { click ⇒ off }
  off { click ⇒ on  }
```

Listing 2: Hoover's Code

This snippet realises the previously described naïve on/off behaviour (Fig. 1(a)). Supporting the behaviour of "turning the hoover off when inactive for a given period" in this event-based language would require (Fig. 1(b)) addition of new:

- state *stand-by* to move to when hoover is on but inactive,
- event *time-elapsed* which is triggered when the specified number of seconds have elapsed without activity,
- transition from the *stand-by* state to *off* state when the *time-elapsed* event occurs, and
- events handling activity/inactivity of the hoover.

Although such adaptation is quite simple and general, it is also very invasive and thus not advisable if it is to be applied to a large set of various types of electronic appliances. The same level of adaptation can be achieved when acting at the *language implementation* level by changing the meaning of the language constructs. Here we need to: (i) collapse the two "active" states (i.e., *on* and *stand-by*) into one (i.e., *on*) and (ii) change the meaning of "being in a such a state".

```
module StateModule {
  reference syntax {
     State ←— StateName "=" Expr ;
  }
  role(execution) {
    0 .{ newstate($1.state, $2.action); }.
    2 .{ if ($1.state == "On")
           $2.action = add_code($2.action,
              "counter = System.currentTimeMillis()")
      }.
  }
}
```

Listing 3: New version of the `State` module.

In short, the `turn_on()` operation will not only turn on the appliance but also store when it is turned on. Two (internal) events: *activity* and *time_elapsed* are added (cf. Fig. 1(c)), as well as transitions from the state *on* to the state *on* (guarded by *activity*) and to state *off* (guarded by *time_elapsed*). The former event will reset the stored time as a consequence of some activity, while the latter will check the current time against the stored time to detect if it is time to turn the appliance off due to prolonged inactivity.

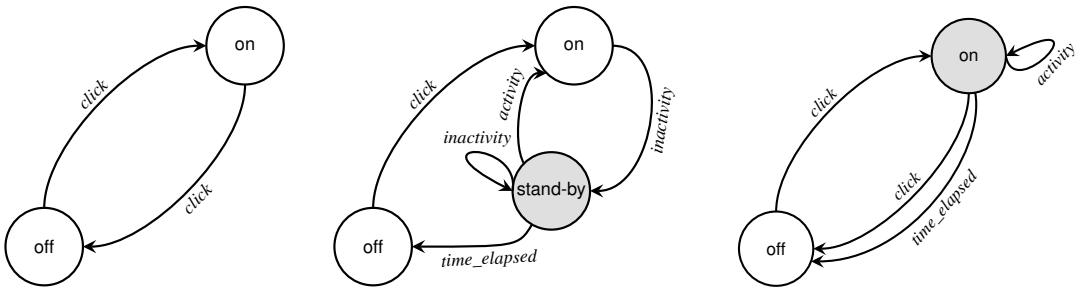
To understand the entirety of the change, compare Listings 1 and 3. Essentially, we extend actions associated with the state *on* when it is initially declared. The original action is stored in the attribute `action` and calculated during the evaluation of the `Expr` concept (i.e., the corresponding slice); in the `State` slice we add an instruction to save the current time to it before creating the state with the associated action. Note, that due to a peculiarity of Neverlang, the semantic action associated with position 2 is evaluated before that of position 0.

2) Technical Sustainability via Extension to GPLs: Consider a new requirement aimed at extending phone battery life, stating that when phone battery is low a less resource-intensive mode (e.g., with darker screen, no internet access, etc.) of the application should be used. Instead of extensively changing the application to create various modes, we propose to extend the programming language by introducing a **`battery`** construct through a micro-language leaving all else unchanged. This construct will permit to dynamically switch through the different code implementations according to the battery level.

The implementation is shown in Listing 4. Here the new statement smoothly composes with any language that supports the statement concept (represented by the `Stmt` nonterminal) and the code block concept (represented by the `Block` nonterminal). These two concepts are provided by other slices and compose on the corresponding nonterminals. So, the **`battery`** statement can be used where a statement can be used including a code block. Basically, the execution role has to determine which piece of code should be executed comparing the current battery charge to the ranges specified for the various branches.

V. Discussion and Conclusions

The two scenarios above highlight various benefits of the micro-language approach with regards to technical sustainability of software. Firstly, it *preserves program clarity throughout*



(a) without stand-by (b) with stand-by (c) with "active" nodes merged

Fig. 1. Hoover's behaviour

```
module BatteryModule {
  reference syntax {
    Stmt ⇦ BatteryBranch ;
    BatteryBranch ⇦ Battery ;
    BatteryBranch ⇦ Battery "□" BatteryBranch ;
    Battery ⇦ "battery" "(" Number ")" "⇾" Block ;
  }
  role(execution) {
    0 .{ $0.code = $1.code }.
    2 .{ $2.level = $3.number; $2.code = $3.code; }.
    4 .{ int bl = getBattery();
         if (bl<=$5.level and bl>$6.level) {
            $4.code = $5.code; $4.level=$5.level;
         } else { $4.code = $6.code; $4.level=$6.level; }
    7 .{ $7.level = $8.number; $7.code = $9.code; }.
  }
}
```

Listing 4: Neverlang `Battery` statement implementation.

evolution. The language implementation adaptation has the benefit of changing only the language semantics, preserving its syntax. Therefore, any program written in the language remains unaltered and benefits from the new behaviour. Also, extensions can be easily introduced by affecting only the components that implement the related concepts.

Secondly, it supports *seamless alignment with domain constructs* providing the ability to define a language element that directly aligns with sustainability concepts and can be realigned as these concepts evolve. Finally, it *reduces code repetition and maintenance costs* by moving the behaviour required by the sustainability semantics into language constructs.

The work presented here contributes towards technical sustainability research by addressing such characteristics as: (i) **Adaptability:** micro-languages facilitate unimpeded evolution of both concerns and language semantics; (ii) **Maintenance:** the narrow scope of a micro-language localises changes within this scope, reducing ripple effects; (iii) **Innovation and Understandability:** each new concern can be equally well-supported and directly aligned with the domain; (iv) **Obsolescence:** as software can evolve without affecting hardware and also accommodate hardware change.

While, this paper largely focused on technical sustainability of software, its application examples are from the wider sustainability domain (i.e., reduced emissions and resource use), showing that wider sustainability too is facilitated by this work. Indeed, since it directly maps the vocabulary of the sustainability domain onto that of software development, it also allows for the more direct expression of relevant behaviour in programs. Thus, any concept and its behaviour perceived important for sustainability (e.g., privacy, health, etc.) can be directly captured into its own micro-language.

As for further work, studies are needed to understand the overhead arising from design of multiple micro-languages. Furthermore, the risk of mistaken use of language constructs needs to be analysed, i.e., when using the same language element to convey evolving semantics and behaviour, one may risk confusion as to which version of semantics one is using.

References

[1] M. Al Hinai and R. Chitchyan, *Social Sustainability Indicators for Software: Initial Review*, Proc. RE4SuSy'14, pp. 21-27, 2014.

[2] W. Cazzola, *Domain-Specific Languages in Few Steps: The Neverlang Approach*, Proc. SC'12, LNCS 7306, pp. 162-177, 2012.

[3] W. Cazzola and E. Vacchi, *Neverlang 2: Componentised Language Development for the JVM*, Proc. SC'13, LNCS 8088, pp. 17-32, 2013.

[4] S. Garg, C. S. Yeo, A. Anandasivam, R. Buyya, *Environment-Conscious Scheduling of HPC Applications on Distributed Cloud-Oriented Data Centers*, J. of Parallel and Distributed Computing 71(6): 732-749, 2011.

[5] P. Greenwood, A. Rashid, J. Walkerdine, *UDesignIt: Towards Social Media for Community-Driven Design*, ICSE NIER, pp. 1321-1324, 2012.

[6] G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, A. Mockus, *Does Code Decay?*, IEEE Trans. SW Eng., 27(1): 1-12, 2001.

[7] F. Héliot, *Near-Optimal Energy-Efficient Joint resource Allocation for Multi-Hop MIMO-AF Systems*, Proc. PIMRC, IEEE, pp. 943-948, 2013.

[8] *ICSE-SEIS 2015: SEIS Track Call for Contributions*, URL: http://2015.icse-conferences.org/call-dates/call-for-contributions/seis.

[9] L. Kats, E. Visser, *The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs*, OOPSLA'10, pp. 444-463.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*, ECOOP'97, pp 220-242

[11] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, *LISA: An Interactive Environment for Programming Language Development*, CC'02, 2002.

[12] A. Moreira, A. Rashid, J. Araujo, *Multi-Dimensional Separation of Concerns in Requirements Engineering*, Proc. RE 2005, pp. 285-296

[13] B. Penzenstadler, *Infusing Green: Requirements Engineering for Green In and Through Software Systems*, Proc. RE4SuSy'14, CEUR pp. 44-53.

[14] A. Rashid, R. Chitchyan, *Persistence as an Aspect*, AOSD'03, 120-129.

[15] P. L. Tarr, H. Ossher, W. H. Harrison, S. M. Sutton Jr., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, ICSE, 107-119, 1999.

[16] E. Vacchi, W. Cazzola, *Neverlang: A Framework for Feature-Oriented Language Development*, Comp. Lang., Systems & Struct. To appear.

[17] E. van Wyk, D. Bodin, J. Gao, L. Krishnan, *Silver: An Extensible Attribute Grammar System*, Science of Comput. Programming: 75(1-2), pp. 39-54, 2010.