Software Development Process: An Action Grammars Perspective

Diana Kirk
Independent Researcher, Auckland 1010, New Zealand

Keywords: Software Process Modelling, Software Methodologies, Action Networks.

Abstract: Practitioners adapt development methodologies to suit local needs, for example, by combining agile and traditional elements. To support this, we need a deeper understanding of the underlying mechanisms behind the various software development approaches, with the aim of finding a perspective that is common to all. In the field of IT, it has been suggested that a change in perspective from 'technology and people' to 'action' might help address the disconnect between artifact- and people-centric approaches. In this position paper, we explore this idea for software development by considering a change in viewpoint from *methodology* to *patterns of action*. Actions are the core functions that are the building blocks for software development and are defined in a *lexicon*. Possible relationships between actions are depicted in a *grammar* and processes are modelled as *action grammars*. We represent some common development methodologies as action grammars and discuss insights gained. Our contributions are the exploration of a novel way of viewing the software process, some insights gained and the exposure of issues with popular terminology.

1 INTRODUCTION

There are numerous approaches to developing computer software. Although many practices are common to both (Clarke et al., 2016; Larman and Basili, 2003), the underlying paradigms differ, with the more traditional approaches viewed as document-centric and the more recent agile methods viewed as people-centric. The software community thus finds itself in a dilemma. In the first instance, there is wide acceptance that practitioners mix and adapt methodologies to suit specific environments (Avison and Pries-Heje, 2008; MacCormack et al., 2012; Müller et al., 2009; Petersen and Wohlin, 2009; de Azevedo Santos et al., 2011; Turner et al., 2010) and researchers are unable to confidently support such adaptation. Secondly, there is a growing move towards greater 'agility' by organisations that have followed more traditional approaches due to a need for standards compliance, for example, in the avionics industry (Petersen and Wohlin, 2010). Indeed, Petersen and Wohlin have identified the need for a "research framework for agile methods" that describes the "characteristics of the processes studied" (Petersen and Wohlin, 2009). We would like to be able to view all methodologies through a common lens. We would then be in a position to start with the problem to be solved and then design a process solution, rather than begin with a methodology (process solution) and try to fit the problem into it.

The disconnect between technology- and people-centric approaches in the field of Information Technology (IT) has been recently addressed by a suggested change of perspective to that of the 'action' (Pentland and Feldman, 2008; Pentland, 2008). Pentland explores the use of *grammars of action* as a means of reconciling artifact- and people-centric viewpoints. As an engineering tradition, IT focus has largely been on *artifacts*. However, the human-centric nature of IT systems has resulted in behavioural science based studies into *actors*. The author presents an illustrative example of mice running through a maze. Engineering-based questions might be "Is virtual cheese as good as real cheese?" and "How many buttons should the mice have?" whereas behavioural science-based questions might be "How do mice learn to navigate the maze?" and "How can we control the behaviour of opportunistic mice?".

Pentland argues that an approach based on *actions* might be fruitful as a means of reconciling these viewpoints. Possible actions are defined in a *lexicon* and an *action grammar* is created by stating the rules for combining actions. Surface *patterns of action* (i.e. individual instantiations) based on the grammar can be studied with the aim of exposing underlying structures i.e. a *grammar* is used as "tool for describing

Kirk, D.
Software Development Process: An Action Grammars Perspective.
DOI: 10.5220/0007759205020509
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 502-509
ISBN: 978-989-758-375-9

and discovering structures" (Pentland, 2008).

We see parallels between the situation described above for IT and the current issues in software development. In both cases, the ontologically irreconcilable viewpoints of artifact versus actor render it difficult, if not impossible, to evaluate software process tailoring efforts. In this position paper, we explore the possibility of applying the ideas of Pentland to software development. We hope to evaluate their suitability as a means of reconciliation. We use a lexicon based on some of our previous work (Kirk and Tempero, 2012) and introduce rules based on some common ideas, for example, 'iteration'. We then represent some popular development approaches as 'action grammars' and discuss insights gained. Our contributions are an abstraction that reconciles development approaches and the exposure of issues with popular terminology.

In section 2, we describe our lexicon and grammatical elements and in Section 3, we model some popular development methodologies and discuss insights. In section 4, we discuss findings and in section 5, we summarise the paper and discuss future work.

2 RESEARCH GRAMMAR

In this section, we explore the notion of *action grammars* for software development. Our study is driven by a long-held frustration with the focus within the discipline on the *differences* between agile and plan-driven approaches, when it is clear that all approaches involve the same kinds of *function* (for example, scoping) and indeed implement the same kinds of *solution* building blocks (for example, iteration, incremental delivery). Incremental, iterative and evolutionary approaches to developing software intensive systems have been implemented since the early 1970s (Larman and Basili, 2003). The notion of delivering products in increments grew from the 1930s quality guru Walter Shewart. Shewart introduced a series of short 'plan-do-study-act (PDSA)' cycles in a bid to control product quality. This approach heralded the 'continuous improvement' paradigm and the cyclical delivery paradigm was adopted by NASA engineers for software development in the 1960s. Short iterations (1 month), time-boxing and test-first development were practised during this decade, mostly on extremely large military projects (Larman and Basili, 2003). Even the much-maligned single-pass waterfall model has its roots in a paper by Winston Royce, who in fact recommended that linear process was suitable for only the simplest projects and that in general long project should be delivered in several iterations to solicit feedback and manage technical risk (Royce, 1970).

Based on the above, our allowed *actions* (*lexicon*) relate to common development *functions* and our *action grammar* contains the 'organising' notions, for example, iteration.

2.1 Lexicon

Our lexicon should contain the various possible actions for software development. If we relate 'action' to 'software development activity' the number of possible actions prescribed by the standard process assessment models is extremely large. For example, the assessment standard ISO/IEC 15504 (SPiCE) contains 46 processes, organised into ten process areas (International Standards Organisation, 2006). In addition, we have discovered in previous research that these models are unsuitable when the aim is to *understand* how software organisations go about developing software-intensive products (Kirk and Tempero, 2012). We found that the terminology used in the standard was often not understood by practitioners and so it was extremely difficult to align what was practised with the standard. We also found that, when discussing, for example, an engineering activity such as design, practitioners wanted to discuss all aspects of the activity at the same time. The attempt to split the discussion into into engineering and support aspects such as documentation led to frustration for all parties.

Our response to the problem of how to discover what is actually happening in industry in a flexible, repeatable way was to focus on what organisations *need to achieve* rather than on the processes required to achieve it. We applied an abstraction based on the high level functions that must be carried out when producing any software. (Kirk and Tempero, 2012). The organisation must:

- Define what is to be made.
- Make it.
- Deliver it.

For example, an organisation may elicit the requirements for a pending release in a formal way, capturing requirements in a software specification document from which developers implement the product. A smaller organisation may have developers speak directly with clients and implement required functionality according to understanding and feedback. There is no 'right' way — the 'best' way depends on the project environment.

In our earlier study, we expanded *Define* into *Roadmap* (e.g. strategic and product-line planning)

Table 1: Lexicon.

Roadmap	Plan for product evolution.
Scope	Establish requirements for next release.
Architect	Create high level structure.
Implement	Design and implement modules.
Integrate	Integrate the modules.
Release	Hand over to intended recipient.
Support	Support the situated product.

and *Scope* (release scoping). We expanded *Make* into *Architect*, *Implement* (design and code) and *Integrate*. We expanded *Deliver* into *Release* (hand over to intended recipient(s)) and *Support* (support the situated product). These functions become the *actions* for our lexicon and are shown in table 1.

A similar categorisation based on function is also applied by Petersen and Wohlin in a study into issues in agile and iterative development (Petersen and Wohlin, 2009). In this study, the authors divide interviewee roles according to:

What. Decide what to develop from a strategic perspective (maps to *Roadmap*).

When. Plan the timeline from a technical perspective (maps to *Scope*).

How. Architect and implement the system (maps to *Make*).

Quality Assurance. Test the software and review documentation (maps to *Release*).

Mappings are not exact. For example, a final category used by Petersen and Wohlin is *Life-cycle management*, which addresses a range of activities, including *configuration management*, *maintenance and support* and *packaging and shipment*. In our scheme, the first would be viewed as covering several functions (for example, configuration control maps to *Release* and change control maps to *Scope* (functionality evaluated) and *Support* (mechanism for client requests). However, the precedent of categorising according to *function* is set.

A resulting characteristic of the actions included in the lexicon is that each describes *what* must be achieved but does not place any constraints as to *how*. Our aim is that this approach will provide some insights into similarities and differences that are not generally apparent when we view methodologies at the process implementation level. We would like to expose process 'families' that we may reason about and which lead to increased understanding of where replacing activities is appropriate. For example, we do not include activities such as 'testing' or 'inspections' because these represent *how* a goal is achieved (Kirk and Tempero, 2012).

We include in our lexicon only actions that relate directly to the product being developed. We also do not include actions describing management activities, for example, planning and monitoring, because we view these as activities that affect function implementation (*how*), rather than relating directly to the product-under-development.

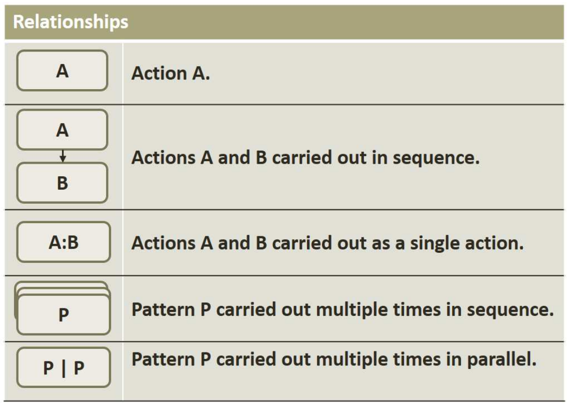
Relationships
A
Action A.
A
B
Actions A and B carried out in sequence.
A:B
Actions A and B carried out as a single action.
P
Pattern P carried out multiple times in sequence.
P | P
Pattern P carried out multiple times in parallel.

Figure 1: Relationships for software methods.

2.2 Grammatical Elements

For the relationships between actions, we include patterns commonly found in software process methodologies. These are shown in figure 1. Actions may be carried out as a sequence (one following the other, as in a traditional approach) or my be combined as a single action (for example, the 'implement and integrate' pattern commonly practised in an agile context). To support comprehension, relationships can be hierarchically abstracted into a higher level block, P(attern). A patterns can be carried out multiple times in sequence (i.e. iterated) or multiple times in parallel (i.e. simultaneous development).

3 ANALYSIS

In this section, we model some popular development approaches as action grammars and examine these for insights.

3.1 Waterfall

In figure 2(a), we show the basic grammar for the waterfall model. Functions are carried out in a linear way, with no merging of functions. Integration generally uncovers implementation issues and so 'Implement' and 'Integrate' iterate with progressive builds until a 'successful' build is achieved. 'Release' generally involves some quality-related decisions.

Waterfall projects were generally delivered as 'alpha', 'beta' and final releases, with feedback from

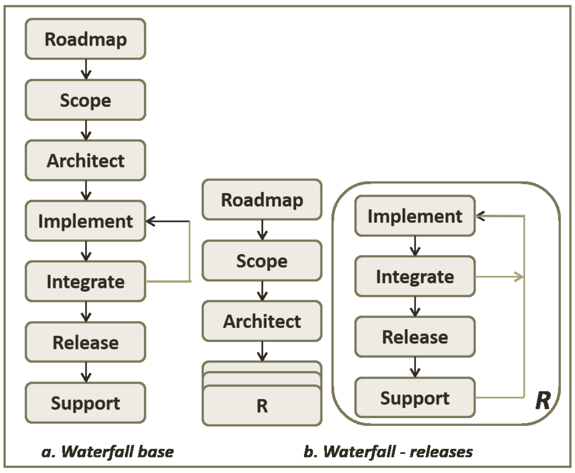

Figure 2: Grammar for waterfall.

clients or users resulting in some alterations in implementation. ‘Support’ generally involves providing a mechanism for such feedback. In figure 2(b), we show the same grammar with the iterated element abstracted as ‘R’ (for ‘create *r*elease’).

However, in practice, as each function was implemented, issues from previous functions were exposed and the problem may have originated at any point earlier in the process. For example, a problem uncovered during integration might have originated in a misunderstanding of functionality i.e. during product scoping. This means that the grammar for waterfall-as-practised requires a set of feedback loops from each action to all previous actions. Far from being the ‘simple’ model it is reputed to be, the grammar for waterfall-as-practised indicates many possible paths (patterns).

3.2 Parallel Development

Large projects are often spit into modules for implementation. Each *M*odule is implemented and integrated and then integration takes place for all modules. This grammar is shown in figures 3a and 3b. However, if a module is complex in its own right, it might be that it will include its own architecture function 3c. There is no one single grammar that is appropriate in all cases. In addition, the possible feedback mechanisms discussed in the previous section apply.

3.3 Incremental Development

In this approach, a specified and architected software system is developed incrementally, with each increment delivering a more complete subset of the complete requirements than the previous. As each increment is completed, new functions are added and

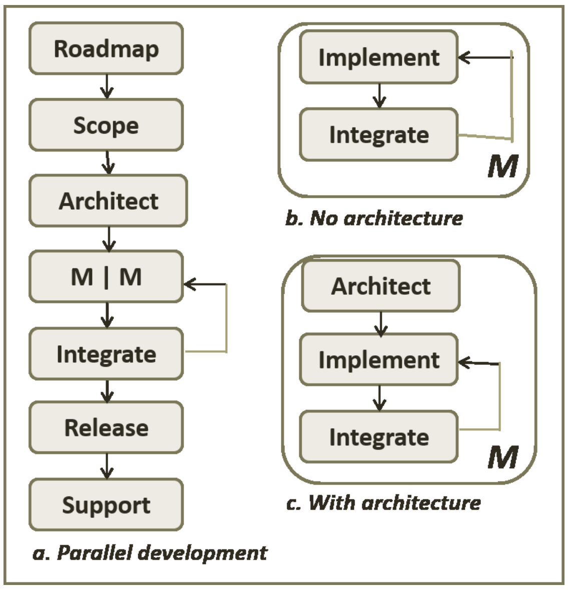

Figure 3: Grammars for parallel development.

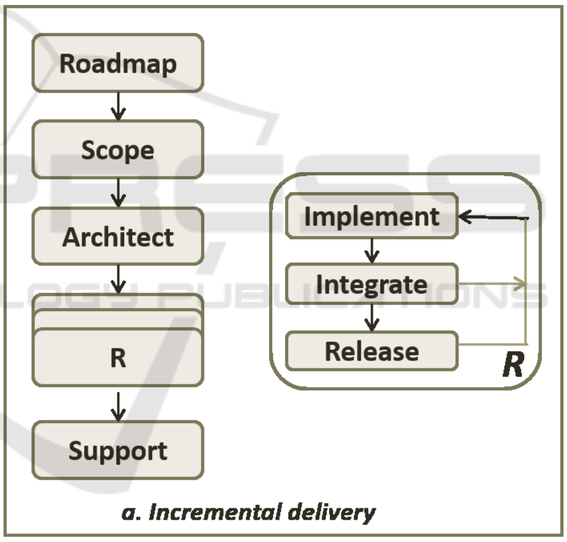

Figure 4: Grammar for incremental delivery.

design modifications made (Basili and Turner, 1995; Larman and Basili, 2003). Yet again, there are multiple possible scenarios i.e. grammars. If the increments remain within the development organisation, the grammar is as shown in figure 5. If the product is released, we have *incremental delivery*. In this grammar, the iteration includes the release activity (figure 4). If a mechanism for client feedback is included, *Support* would also be included in the iteration.

3.4 XP (eXtreme Programming)

For XP, we use the original definition of practices as cited by (Beck, 2000). XP was selected because its

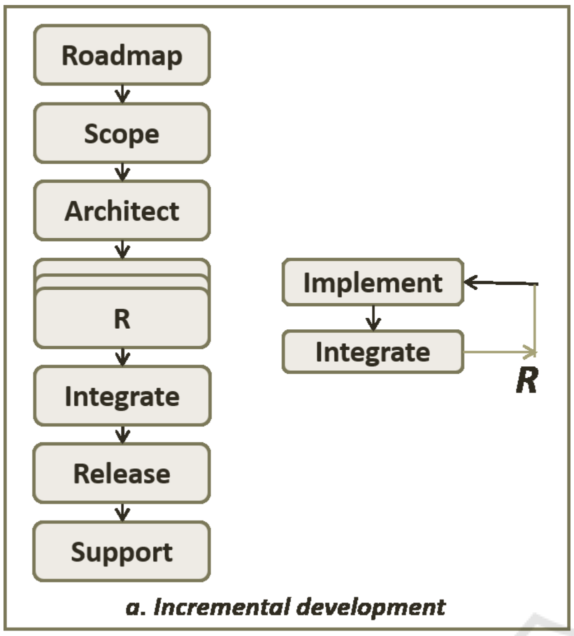

Figure 5: Grammar for incremental development.

practices are well-defined but do not obviously fit into a standard process paradigm. For example, what is the function that is described by *Metaphor*?

From our earlier analysis (Kirk and Tempero, 2006):

- *Planning Game*, *Metaphor* support the understanding of what is to be made. These relate to the *Scope* function.
- *Simple Design*, *Pair Programming*, *Collective Ownership*, *On-Site Customer* and *Coding Standards* are practices to support developers in designi and code and relate to *Implement*.
- *Continuous Integration* demands that code is integrated after every change with a view to quickly identifying and resolving defects. This maps to *Implement* and *Integrate* carried out as a single action.
- *Small Releases* is a statement about iteration length and *40-Hour Week* has the aim of ensuring team members remain motivated. These do not relate to any actions.
- *Testing* has two aspects. The first is the instruction to a developer to continually write unit tests. This relates to *Implement*. The second is the need for a client to test all software at the point of delivery. The aim is somewhat unclear. As the software is delivered no matter what, the main aim appears to relate to planning for the next iteration i.e. relates to the *Scope* action. However, it is also a requirement of handover and so relates to *Release*. This is an example of a practice that has two functions — we discuss in section 4.

Table 2: Mapping of lexicon to XP practices.

Roadmap	
Scope	Planning game, Metaphor, Testing (client).
Architect	
Implement	Simple design, Testing, Pair programming, Collective ownership, Coding standards.
Integrate	Continuous integration.
Release	Testing (client)
Support	

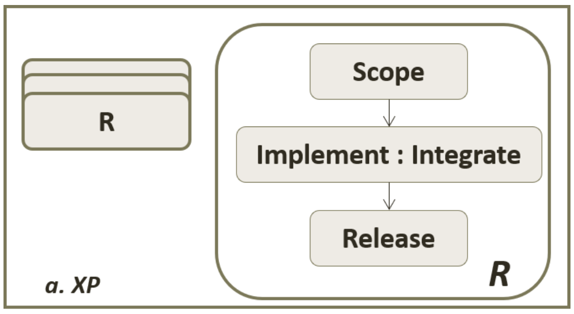

Figure 6: Grammar for eXtreme Programming.

The relationship between XP practices and actions are shown in table 2.

In figure 6 we show the grammar for XP. The inclusion of *Scope* inside each iteration characterises the evolutionary nature of product definition. The practice of continuous integration means that the *Implement* and *Integrate* actions can be viewed as a single action. The functions of *Roadmap*, *Architect* and *Support* are not represented.

There are many situations that result in modifications tht effectively change the basic grammar. For example, if there is an overarching plan for the product, or the intended product is large and/or complex, *Roadmap* and *Architect* may need to be included.

3.5 Continuous Delivery

This approach has emerged more recently with the emergence of startup organisations. The aim is to release an innovative product with minimal functionality quickly to the market, and to 'test the waters' as regards delivered functionality. Each release is delivered as a working product to a large number of customers. As a result of feedback, scope is altered to an improved product. We show a possible grammar for continuous delivery in figure 7.

The figure as shown does not include the *Architecture* function. This may be required if the product is large and complex or will for the basis of a product line.

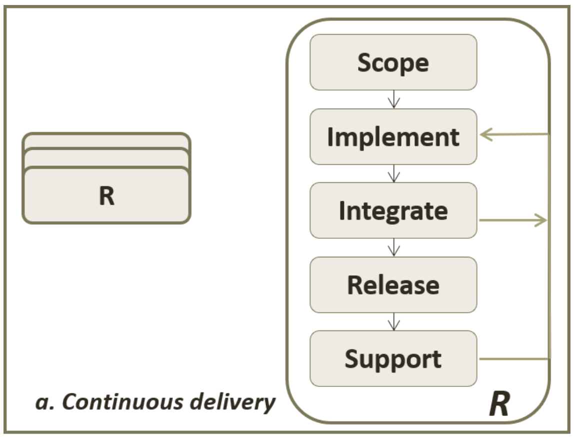

Figure 7: Grammar for continuous delivery.

4 DISCUSSION

The objective of this study is to explore the use of action grammars in the quest for a lens that will enable us to view all development methodologies in a common way. The hope is that such a lens will provide a mechanism for supporting a problem-space approach to architecting custom development methodologies.

We observe from the analysis in section 3 that, when we try to represent popular implemented approaches as action grammars, we realise the terms we have been using are less defined than we thought. Waterfall, far from being simple, is characterised by a myriad of possible action paths i.e. the basic grammar is complex; 'incremental' can be interpreted in at least three ways i.e. by three possible grammars; XP as-defined requires amendment in many instances, with amendments likely to require different basic grammars. The widely quoted 'iterative development' term is essentially meaningless, as all methodologies are iterative in essence. The lack of a clear definition of the terms we use during research efforts would suggest that we are building empirical castles upon shifting sand. We posit that designing a custom process by first focussing on establishing a suitable grammar may be a first step in addressing this problem.

The change in perspective from 'implement fixed methodology' to 'identify grammar and explore common and unused patterns of action' provides us with at least two possible benefits. First, as researchers, we would like to compare approaches, perhaps to identify practices that are best suited to specific environments. We hypothesise that an understanding of the underlying grammars will support this. Second, a common lens provides us with a mechanism for identifying important, specific questions. Some examples are:

- For grammars that include delivery to the client in the iteration, what are the problem-space characteristics that define the 'best' iteration length? For example, not all clients wish to accept delivery on a regular basis (Shahin et al., 2017).
- How do choices in one part of the grammar affect other parts? For example, for modules developed in parallel, if one implements a grammar containing an iteration that includes *Architect*, how might this affect the final integration of the parts?
- Can problems occur if we are not clear about the function that is represented by an implemented activity? For example, we identified the 'Client testing' practice in XP as trying to fulfil two functions, relating to release handover and scoping. Is the client allowed to refuse a release if release testing fails? Forced to pay for a further release just to fix bugs i.e. no scope changes? What happens if the client is not empowered to make scoping decisions? What if the product has no user interface?
- When is it appropriate to combine the *Implement* and *Integrate* feedback loop into a single action and when should these be treated as two actions?

We would hope that having a grammar from which to make decisions about practices to implement might force consideration of such matters in advance. Of course, the action grammars and questions above relate only to *what* must be done, in what order, rather than *how*. The perspective supports questions relating to the organisation of functions but does not support questions such as 'Should we have a customer on-site?' or 'When are daily stand-up meetings contra-indicated?'. Such questions would come after the establishment of a grammar. The usefulness of our approach is dependent upon whether we can successfully map problem space characteristics to the organisational aspects (i.e. pieces of an action grammar). Process design would then involve combining the pieces together into a grammer tailored to the problem. We might then investigate, for example, under what circumstances a specific pattern within a grammar is more / less likely. This represents an exciting area for future work.

Some initial thoughts on possible problem-space mappings are overviewed below.

- If what is to be built is unclear, an iteration that includes *Scope* is indicated.
- If the organisation is a startup, or is 'testing the waters' in a new country, it probably wants feedback relating to a *deployed* product i.e. the iteration include *Support*.

- Continuous integration (implemented by *Implement:Integrate* as a single action) may not be appropriate when the product is legacy software with many dependencies between components (Shahin et al., 2017).
- If the product represents the start of a new product-line involving safety-critical software, perhaps *Architect* must be an integral part of every module.
- A large, complex product suggests a need for *Architect* up front, regardless of any desires to be 'agile'.

Our analysis has raised many questions. We believe the need to answer these is a pressing one. The ability to adapt development process has been shown to affect performance (Clarke et al., 2015) and a common lens is needed to support such adaptation.

5 SUMMARY

In this position paper, we have explored the possibility of using *Action Grammars* to provide a common lens on the various paradigms underlying software development methodologies, for example, agile and plan-driven. Our position is that, rather than starting with a process solution (methodology) and attempting to retrofit the problem, we would like to start with the problem and design an appropriate process solution. We modelled some popular methodologies as Action Grammars. We found that many commonly used terms, for example, 'incremental development' can be interpreted in multiple ways i.e. have different underlying grammars. This means that we are discussing methodologies without a clear understanding of the applicable rules and constraints. Our contributions are a novel abstraction that reconciles development approaches and the exposure of issues with popular terminology.

For future work, we will a) analyse some common scenarios with respect to our lens, and b) apply Pentland's notion of using surface patterns of action as a tool for "describing and discovering" how processes are typically enacted. The first will serve as a means of evaluating the potential of the approach. The second will help us to understand, for example, the factors that affect 'best' iteration length. Our overall aims are to more deeply understand the relationships between problem space characteristics and custom grammar design.

REFERENCES

Avison, D. and Pries-Heje, J. (2008). Flexible information systems development: Designing an appropriate methodology for different situations. In Filipe, J., Cordeiro, J., and Cardoso, J., editors, *Enterprise information systems : 9th International Conference, ICEIS 2007*, pages 212–224, Berlin, Heidelberg. Springer.

Basili, V. R. and Turner, A. J. (1995). Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, SE-1(4):390–396.

Beck, K. (2000). *eXtreme Programming eXplained - Embrace Change*. Addison-Wesley, United States of America.

Clarke, P., Mesquida, A.-L., Ekert, D., Ekstrom, J., Gornostaja, T., Jovanovic, M., Johansen, J., Mas, A., Messnarz, R., Villar, B. N., O'Connor, A., O'Connor, R. V., Reiner, M., Sauberer, G., Schmitz, K.-D., and Yilmaz, M. (2016). An Investigation of Software Development Process Terminology. volume 609 of *Communications in Computer and Information Science (CCIS)*, pages 351–361. Springer International Publishing, Switzerland.

Clarke, P., O'Connor, R. V., Leavy, B., and Yilmaz, M. (2015). Exploring the Relationship between Software Process Adaptive Capability and Software Organisational Performance. *IEEE Transactions on Software Engineering*, 41(12):1169–1183.

de Azevedo Santos, M., de Souza Bermejo, P. H., de Oliveira, M. S., and Tonelli, A. O. (2011). Agile practices: An assessment of perception of value of professionals on the quality criteria in performance of projects. *Journal of Software Engineering and Applications*, 4:700–709.

International Standards Organisation (2004-2006). ISO/IEC 15504: Information Technology - Process Assessment (Parts 1-5)). The International Standards Organisation.

Kirk, D. and Tempero, E. (2006). Identifying Risks in XP Projects through Process Modelling. In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*, pages 411–420, Sydney, Australia. IEEE Computer Society Press.

Kirk, D. and Tempero, E. (2012). A lightweight framework for describing software practices. *Journal of Systems and Software*, 85(3):581–594.

Larman, C. and Basili, V. R. (2003). Iterative and Incremental Development: A Brief History. *IEEE Computer*, 36(6).

MacCormack, A., Crandall, W., Henderson, P., and Toft, P. (2012). Do you need a new product-development strategy? *Research Technology Management*, 55(1):34–43.

Müller, S. D., Kræmmergaard, P., and Mathiassen, L. (2009). Managing Cultural variation in Software Process Improvement: A Comparison of Methods for Subculture Assessment. *IEEE Transactions on Engineering Management*, 56(4):584–599.

Pentland, B. T. (2008). Desparately Seeking Structures: Grammars of Action in Information Systems Research. *The DATA BASE for Advances in Information Systems*, 44(2):7–18.

Pentland, B. T. and Feldman, M. S. (2008). Designing routines: On the folly of designing artifacts, while hoping for patterns of action. *Information and Organization*, 18:235–250.

Petersen, K. and Wohlin, C. (2009). A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case. *Journal of Systems and Software*, 82:1479–1490.

Petersen, K. and Wohlin, C. (2010). The effect of moving from a plan-driven to an incremental software development approach with agile practices. *Empirical Software Engineering*, 15:654–693.

Royce, W. (1970). Managing the Development of Large Software Systems. In *Proceedings, IEEE WestCon*, pages 328–339. The Institute of Electrical and Electronic Engineers, Inc.

Shahin, M., Babar, M. A., and Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943.

Turner, R., Ledwith, A., and Kelly, J. (2010). Project management in small to medium-sized enterprises: Matching processes to the nature of the firm. *International Journal of Project Management*, 28:744–755.