A domain-specific language framework for farm management information systems in precision agriculture

Desirée Groeneveld¹ · Bedir Tekinerdogan² · Vahid Garousi³ · Cagatay Catal^{2,4}

Accepted: 20 October 2020 / Published online: 1 December 2020
© The Author(s) 2020

Abstract

Farm management information system (FMIS) is an important element of precision agriculture to support the decision making process in the agricultural business. Developing FMIS is not trivial and requires the proper design and implementation models for supporting the understandability, enhancing communication and analysis of the design decisions, and the communication among stakeholders. To cope with these challenges, a Domain-specific language (DSL) framework for the design and development of precision-agriculture FMISs is proposed and evaluated. The DSL framework is developed based on a domain-driven design approach in which a feature diagram is provided that represents the common and variant features of the precision agriculture domain. The key requirements for the DSL framework are discussed, the scope of the DSL is defined, and the set of DSLs for supporting FMISs is presented. A controlled experiment demonstrates that the proposed DSL framework is easy to learn and to use, and is effective in developing high-quality system models for precision agriculture.

Keywords Domain-specific language · Precision agriculture · Farm management information systems · Empirical study · Software engineering · System analysis and design

✉ Bedir Tekinerdogan
bedir.tekinerdogan@wur.nl

Desirée Groeneveld
desiree_groeneveld@hotmail.com

Vahid Garousi
v.garousi@qub.ac.uk

Cagatay Catal
cagatay.catal@eng.bau.edu.tr

¹ Agri Information Partners, Wageningen, The Netherlands

² Information Technology Group, Wageningen University & Research, Wageningen, The Netherlands

³ School of Electronics, Electrical Engineering and Computer Science (EEECS), Queen's University Belfast, Belfast, UK

⁴ Computer Engineering, Bahcesehir University, Computer Engineering, Istanbul, Turkey

Introduction

Precision agriculture is a farming management concept that applies advanced information technology and the corresponding principles to increase production and economic returns, often also intending to reduce the impact on the environment (Rains and Thomas 2009). In recent years, a large number of studies have discussed and proposed novel developments in precision agriculture (Abbasi et al. 2014; Abt et al. 2006; Fountas et al. 2015; Tummers et al. 2019; Verdouw et al. 2019, 2016; Pinet et al. 2009). By developing smart systems that are integrated with or support decision-support systems (DSS) it is aimed to optimize productivity while conserving resources such as water, fertilizers, and pesticides (Köksal and Tekinerdogan 2019).

For realizing precision agriculture, software engineering has become increasingly important. Software defines the intelligence that is needed for supporting the decision-making process. For helping the agricultural decision problems, it is necessary to adopt a programming language to provide the proper implementation of the system that meets the stakeholder concerns. Traditionally, general-purpose programming languages (GPLs) that are agnostic to a particular domain are being used to develop programs for agriculture. Unfortunately, these GPLs often fail to express the specific concerns of the agriculture domain, have a steep learning curve, and thus are difficult to master in a short time. Bridging the gap between conceptual models of precision agriculture and their implementation on a computer can easily lead to many pitfalls. GPLs are less expressive to represent the specific agriculture concerns and are less accessible by the domain experts in the agriculture domain. *Domain-specific languages* (DSLs) (Kleppe 2009; Stahl and Voelter 2006; Voelter 2013; Tekinerdogan and Demirli 2013) have been developed in several scientific fields to facilitate the construction of models at a level closer to the conceptual model, thereby making model implementation more accessible to domain experts and declarative, rather than imperative (Challenger et al. 2015; Fall and Fall 2001; Fowler 2010; Mernik et al. 2005; Nascimento 2012; Oliveira et al. Oliveira et al. 2009; Tekinerdogan and Arkin 2019).

In contrast to a GPL, a DSL is based on abstractions closely aligned with the domain for which the language is built. Hence a DSL is optimized for a given class of problems in a particular application domain. Because of the focus in a specific domain, DSLs are usually small languages. The notion of DSLs has been successfully applied in many different application domains. Examples of well-known DSLs are languages such as HTML and SQL, which are broadly applied. DSLs have also been used in various other more specific application domains such as landscape modeling (Gaucherel et al. 2006), biology (Anantharayanan and Thies 2010; Bilitchenko et al. 2011), environmental sciences (Channe et al. 2015), and agricultural sciences (Doering et al. 2013, 2016), and robotics (Nordmann et al. 2014). Often a DSL framework is proposed, which is defined as a coherent set of DSLs focusing on the same problem domain but addressing specific concerns in that domain (Tekinerdogan and Arkin 2019).

In recent years, the agricultural sector has changed radically due to rapid technological developments (Lindblom et al. 2017; McBratney et al. 2005; Nash et al. 2009a, b; O'Grady and O'Hare 2017; Paraforos et al. 2016; Tekinerdogan et al. 2018). In parallel, the agricultural sector faces the challenge of fulfilling several goals and societal values at the same time while also expecting to be sustainable and economically viable on a long-term basis (Tummers et al. 2019, 2020; Noppen et al. 2004). Precision agriculture can help meet these difficult challenges but requires the development of large scale software that meets the

stakeholder requirements. Similar to the motivations for other application domains, DSLs can leverage the development of precision agriculture systems, thereby reducing the conceptual gap, increasing the productivity and quality of the developed software systems.

In this paper, a domain-specific language (DSL) framework for the design and development of precision-agriculture software is proposed and evaluated. The DSL framework is developed based on a domain-driven design approach in which a feature diagram is provided that represents the common and variant features of the precision agriculture domain. The language framework consists of a coherent set of separate DSLs that have been designed together, and each of which addresses a particular concern in precision agriculture. The key requirements for such a DSL framework are discussed, the scope of the DSL is defined, and the set of DSLs for supporting FMISs is presented. A controlled experiment demonstrates that the proposed DSL framework is easy to learn and to use, and is effective in developing high-quality system models for precision agriculture. The study shows that, in the context of precision agriculture software, this kind of DSLs has a vast potential to improve the collaboration between domain experts and developers.

The rest of this paper is structured as follows: First a short background on domain-specific languages is provided, which is followed by a discussion of the motivations and requirements for adopting DSLs in precision agriculture software systems. Subsequently, the requirements for the DSL framework for FMIS are described, followed by a domain analysis of precision agriculture, which is necessary to develop the domain models for the DSLs. The article continues with the presentation of the DSL framework. Further the empirical evaluation of the DSL framework is described, followed by the discussion. Finally, the paper is concluded with the lessons learned and future work.

Domain-specific languages

In this section, the key concepts for understanding DSLs, the motivations for adopting DSLs, and finally, the requirements of a DSL for meeting the stakeholder concerns are presented.

Key concepts

General-purpose languages (GPL), such as Python, Java, and C#, have been designed for a wide variety of application domains, and thus do not include language constructs for a particular application domain. A domain-specific language (DSL) instead is a tailor-made language that offers, through appropriate notations and abstractions, expressive power focused in a specific problem domain (Strembeck and Zdun 2009; Voelter 2013). In contrast to GPLs, DSLs have expressive power targeted at a specific domain and are intuitively understandable for domain experts. This implies that a DSL should be declarative, and likewise, designers of DSLs only use essential domain features and concepts to customize the DSL to the domain.

Several types of DSLs can be developed. A DSL can be embedded in an existing language or designed to be an extension of an existing language (Strembeck and Zdun 2009; Voelter 2013). Another possibility is to create an entirely new DSL. Often, the easiest way to design a DSL is to base it on an existing language. However, each method has its benefits and drawbacks. There are also some general characteristics and challenges for DSL development. To avoid misinterpretations of the domain and to avoid making a DSL that

does not fit the needs of the domain experts, software engineers need to increase the participation of domain experts in DSL design activities.

The application of a systematic, disciplined, and quantifiable approach to the development, use, and maintenance of these languages is usually called *software language engineering*. Typically, two different roles can be distinguished: the *language engineer* and the *language user* (Kleppe 2009). The language engineer is responsible for creating the language. The language user is the person who uses the language to develop applications. Figure 1 shows the conceptual model for describing the key concepts of DSLs and software language engineering.

A DSL runs on a *target platform* and is assumed to be something that cannot change (significantly) during the DSL development process. The target platform defines the implementation platform on which the DSL code is deployed and executed. The *execution engine* bridges the gap between the DSL and the platform and can be an *interpreter* or a *generator*. An interpreter is a program running on the target platform that loads a DSL program and then acts on it. A generator takes the DSL program and transforms it into an artifact that can run directly on the target platform. Usually, a distinction is made between *external vs. internal DSLs*. An external DSL is represented in a separate language to the main programming language it is working with. An internal DSL is represented within the syntax of a general-purpose language (Strembeck and Zdun 2009; Voelter 2013).

A DSL consists of the following main elements. The *abstract syntax* defines the vocabulary of concepts provided by the language and how they may be combined to create models.

Fig. 1 Basic elements of DSLs (Adapted from: Tekinerdogan and Arkin 2019)

The *concrete syntax* defines the notation that facilitates the presentation and construction of models or programs in the language. It can be visual or textual. *Well-formedness rules* (*static semantics*) defines additional constraint rules on abstract syntax that are hard or impossible to express in standard syntactic formalisms of the abstract syntax. *Semantics* includes the definition of the meaning of the concepts in the abstract syntax.

One important design decision when developing domain-specific languages is the use of corresponding libraries. In fact, most programming languages have an associated core library made available by all implementations of the language. The inclusion of libraries helps to reduce the complexity of the language itself and also provides additional reuse of recurring program structures. The library can be considered separate from the language or be treated as part of the integrated whole.

A *language workbench* is an environment designed to help people create new DSLs, together with high-quality tooling required to use those DSLs effectively. A language workbench provides a set of tools for supporting the language engineer in creating a DSL.

Motivations for DSL

The motivation for adopting DSLs have been broadly discussed in the literature (Mernik et al. 2005; Challenger et al. 2015; Fowler 2010; Voelter 2013). The following subsections elaborate on these from the perspective of precision agriculture.

Enhancing understandability and communication among stakeholders

Some of the current technology in agriculture results from a technology push rather than a need or a problem (Lindblom et al. 2017). This is not necessarily a problem when the problems solved are non-complex issues, have observable impacts, have high compatibility, and are easy to implement. However, for more complex problems, co-development is desired, where the end-user is an essential participant in the whole development process (Lindblom et al. 2017; O'Grady and O'Hare 2017) since the ultimate goal of agricultural software is to help the farmers (Kerneck et al. 2020). It is the farmer who has to transform knowledge from daily work practices into science while staying within set boundaries. A DSL can help in this context since it supports communication among domain experts by providing a clear and precise language focused on a particular domain. The model implementation becomes more accessible to domain experts, and the model becomes less hidden in the details of the code, what happens when using a GPL. In other words, DSLs are built to let the programmer focus on the application domain and not the programming details. Research on farm-specific models, especially development and application, is still at an early stage (O'Grady and O'Hare 2017), and the uptake and acceptance of new agricultural software are not as high as it could be (Kolovos et al. 2006; Fountas et al. 2015; Tummers et al. 2019; Nikkilä et al. 2010). This is partly because the newest solutions in agricultural software are developed by scientists together with software developers. Their solutions capture what they consider sufficient knowledge, but these solutions sometimes lack the full knowledge about the requirements of the farmers. However, developers and scientists try to solve this problem by performing requirements engineering, which includes collecting, tracing, analyzing, qualifying, and managing user requirements. The mismatch between the requirements of the farmers and the implemented software is not the only reason for the low adaption. Other causes are complexity and poor user interface design, age and education level of the users, lack of confidence, tedious data input requirements,

low adaptation to the farm situation, no frequent information update, lack of incentive to learn and adopt new practices, high investment costs, and the fear of replacing advisors (Kolovos et al. 2006; Fountas et al. 2015; Tummers et al. 2019). Some of these reasons can be eliminated by using the co-development approach. For example, the user interface can be designed in consultation with the end-user. DSLs support the co-development approach since they are often declarative and easier to understand than GPLs.

Interoperability and information exchange

Mernik et al. (2005) mention that data-structure representation is made easier by DSLs. Currently, farmers lack the tools to make informed decisions related to the financial management of their business while taking into account the cost and profit margins and profitability analysis. One of the reasons for this problem is that the data and information sources needed for precision agriculture are fragmented and difficult to use for the farmer, while the effective capture, processing, and management are fundamental for efficient and sustainable decision making (Fountas et al. 2015; Tummers et al. 2019). The fragmentation and difficulty of the use of data are enhanced by problems in interoperability and missing data standardization. Interoperability problems are observed in the compatibility between software and equipment, but also within equipment (Tekinerdogan et al. 2018; Fountas et al. 2015). Compatibility within software would also help farmers. Researchers often focus on one specific area or problem. Farmers, on the other hand, require a complete solution that takes every aspect of the farm (Tummers et al. 2019) as well the crosscutting concerns into account (Bakker et al. 2005).

Balancing flexibility and complexity

The last motivation for a precision agriculture DSL is the desire for flexibility. The agricultural sector has to deal with uncertainty on several levels (Fountas et al. 2015), for example, the weather, the climate, changing governmental rules, and societal expectations. A GPL offers flexibility in programming but introduces additional complexity, while a DSL can reach a balance between flexibility and structure. For example, when the political situation changes, the idea that the DSL Framework can easily be adapted to fit with the new requirements is a comforting idea. Vaquero-Melchor et al. (2017) mention that DSLs could accept the use of additional elements in modeling beyond those specified in the meta-model or grammar; this allows the software developer more flexibility. DSLs could offer flexibility in the development process, where the available time and scope are respected, and adaptations during the process are accepted. A final benefit of flexibility is that it can help to make software that can meet current and future demands.

Code generation and verification

Presumably, the amount of DSL code that needs to be written is substantially smaller compared to the use of GPLs (Arkin et al. 2013; Schmitt et al. 2014). In case a DSL abstracts away from the underlying technology platform, it can also support platform independence (Arkin et al. 2013; Tekinerdogan et al. 2005). Hereby, the specifications in DSLs can be used to generate code for platform-specific execution environments. Further, using DSLs can increase the quality of the created product due to the removal of unnecessary degrees of freedom for programmers, the avoidance of duplication of code, and the consistent

automation of repetitive work by the execution DSL engineering engine. Finally, since DSLs are more semantically rich than GPL programs, a more detailed and precise verification and validation of the domain concerns can be carried out.

Several other benefits could be derived from adopting DSLs. In general, multiple of these listed benefits together lead to the decision to adopt DSLs.

Requirements for the DSL framework in precision agriculture

Kolovos et al. (2006) made an overview of general requirements for DSLs. They determined nine requirements, specified below, of which the last two are desirable but not necessarily valid for all DSLs.

1. The language constructs shall correspond to important domain concepts
2. Each construct shall be used to represent exactly one distinct concept in the domain
3. A DSL shall offer tool-support which are used to provide DSL support for model and program management
4. The DSL shall either integrate or extend other languages (and tools) with minimal effort
5. The DSL shall be used and useful for a sufficiently long period of time to justify the costs of developing a DSL
6. A language shall be as simple as possible in order to express the concepts of interest and to support its users and stakeholders in their preferred ways of working
7. The DSL shall provide general mechanisms for building quality systems, thus support the development of accurate deployment descriptions through the language or tooling
8. The language provides constructs to help manage large-scale descriptions
9. The language should be easy for users to achieve their objectives.

Most of these requirements are also valid for GPLs, but their importance is higher in DSLs. Some of the requirements assume that analysis regarding costs and benefits is performed (Kolovos et al. 2006). Fall and Fall (2001) applied the third requirement by providing a simulation engine, a graphical modeling environment, and tools to help guide model development. The fifth requirement is confirmed by Lindblom et al. (2017), who states that a sustainable ICT system is among others characterized by longevity. Van Deursen et al. (2000) and Fall and Fall (2001) identify simplicity (the sixth requirement) as the main requirement for a DSL since simplicity promotes transparency and communication. Other authors mention quite a few requirements or characteristics that can be categorized under the ninth requirement of Kolovos et al. (2006).

The general requirements of Kolovos et al. (2006) should be specified to match the precision agriculture domain and the motivations. Kolovos et al. (2006) acknowledge that domain-specific requirements could sometimes contradict the general requirements. The first requirement (1) finds its representation in two different aspects. The first is that the DSL should be capable of *meeting the trade-offs and high complexity* that characterize the decision-making process of farmers. If this is not the case, the farmers will not be able to communicate their problems to the software engineer. The second aspect is that each of the three main components of an agricultural decision-making process should be represented. If either the biological processes, the farm management, or the advisory services are missing, the communication becomes harder. The second requirement of the *uniqueness of domain concepts* (2) stands in relation to the first requirement but is not in need of a specification.

The tool-support of the third requirement (3) can be developed after the foundation of the DSL is made, so it has less priority in the beginning. The fourth requirement (4) and the fifth requirement (5) are combined to the following statement: The DSL should be *extensible and adaptable* so that future demands can be incorporated in the DSL.

The sixth requirement of *simplicity* (6) deserves special attention since this DSL has a motivation to promote understanding and communication of the stakeholders in the agriculture domain. To reach simplicity, two sub-requirements are defined. The first is that the DSL should have a *clear structure* to promote the understandability. Additionally, a clear structure shows in which sense the DSL is flexible. The second sub-requirement is that the DSL should be *modular*. Solutions made for different problems should be easy to combine so that the final program can be tailor-made for each farm. Modularity can ease the conceptual burden of creating complex systems.

Finally, the DSL should be generic enough to address a diverse range of problems, but specific enough to be contained (Fall and Fall 2001); thus, a *balance between generality and sector uniqueness* should be reached. This requirement is important to reach understandability and communication because the generality allows communicating across different sectors, but the uniqueness of the sector should be incorporated; otherwise, some concepts are too difficult to explain or understand. A good balance offers the flexibility to switch between different sectors with the same DSL. The DSL should allow modeling multiple sub-domains in one; otherwise, it could limit the effectiveness of the created models.

For most of these requirements, it is clear how to reach them. However, determining when a good balance is reached is mainly a design decision, and most of the time, it is difficult to pinpoint before the development. It is also challenging to state beforehand when the requirement for meeting trade-offs and high complexity is reached.

Domain analysis process

In this section, domain analysis is performed for precision agriculture to support the development of the required DSL. The analysis focuses on farm management information systems for precision agriculture.

Domain scoping and analysis

Precision agriculture covers many technological aspects, but the main goal is to increase food production growth while ensuring profitability and sustainability. Precision agriculture techniques focus on observing, measuring, and responding to variability within a crop or a herd (Lindblom et al. 2017). The different technological aspects include precise point positioning systems, autonomous vehicles, drones, precision planting, precision fertilizing, and data analysis. Precision agriculture is a management system of which the principles can be applied to almost any agricultural product (Rains and Thomas 2009). Agriculture as a biological production system is characterized by a high degree of uncertainty, which is difficult to approach with deterministic models.

Farm management information systems (FMIS) for precision agriculture have additional requirements than FMIS used for traditional agriculture (Tummers et al. 2019). For example, in the arable farming sector, each field receives a single value in a conventional farm, whereas with precision agriculture, more values are assigned to each field, with the use of GPS data. FMIS for greenhouses requires many control

algorithms for climate control. Another point where precision agriculture is different from conventional agriculture is in the way data is entered in the FMIS. Most of the data is entered by hand in a traditional system. A precision agriculture system uses the digital transfer of data, which can be sent from the sensors directly to the management system (Farias et al. 2017; Gogos et al. 2007; Shaffer et al. 2000; Wang et al. 2008). Getting the data from sensors also means that the system should be able to handle different data formats and needs to transform this data to achieve interoperability (Tekinerdogan et al. 2018). In addition to the different formats of the data, the management system should be able to handle greater amounts of data than for a traditional system. With the data obtained from the sensors and the use of biological models, the management system can create operational plans. The creation of operational plans is a distinct feature of precision agriculture management systems. On the other hand, the greater amount of data collected by sensors is an incentive for farmers to adopt more advanced management systems.

An FMIS for precision agriculture should also offer the possibility of keeping records, managing finances, and planning, just as a traditional FMIS. Additionally, an FMIS could offer other functionalities such as traceability, quality assurance, and sales management (Fountas et al. 2015; Tummers et al. 2019; Köksal and Tekinerdogan 2019). In addition to these functionalities, farm management should take care of national and regional differences in agriculture. In sum, an FMIS should ideally offer (all or some of) the following functionalities: (Field) operations management, best practice advice, finance, inventory, traceability, reporting, site-specific (individuality), sales management, machinery management, human resource management, quality assurance and performance management, production planning, environmental resource management.

For each system that is developed, stakeholders influence the final product. According to Nikkilä et al. (2010), the following stakeholders influence the development of software for agriculture: the farmers, the provider of the software, the developer of the software, the maintainer of the software, contractors, advisors, authorities, customers of the farm, suppliers of the farm, service providers, manufacturers of equipment.

The main concern for the farmers is usability, which can be described as “*the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use*” (Lindblom et al. 2017). Farmers are not necessarily well trained in the use of technology. Understanding the software and learning how to use it correctly can take some time. Therefore, the system should have an easy to use graphical user interface (GUI) for the farmer. In addition to usability, availability and reliability are important aspects of the system for the farmer. These three concerns can be combined to provide convenience. FMIS sometimes offer functionalities that not every farmer wants to use. This is not necessarily a problem, but these extra features complicate the user interface and the other parts of the software, and therefore, they add unnecessary steps to the general confusion of the user.

The provider, developer, and maintainer of the software are mentioned as separate stakeholders, but for smaller projects, these three are actually different roles of a single entity. For a DSL, there are typically three stakeholders, namely, software engineers (developer of software), customers of the software (the farmers), and developers of the DSL. Based on Fig. 1, in Fig. 2, the interactions of the DSL with the stakeholders is shown. Additionally, the product built using the DSL is also shown in this conceptual diagram.

Fig. 2 A conceptual diagram for a precision agriculture DSL framework

Domain modeling

The output of a domain analysis process is a domain model. A formal domain model consists of the following elements (Mernik et al. 2005; Tüzün et al. 2015), domain definition defining the scope of the domain, domain terminology & semantics, descriptions of domain concepts, and feature models describing the commonalities and variabilities of domain concepts and their interdependencies. Specifically, feature models consist of a feature diagram representing a hierarchical decomposition of features and their character, that is, whether they are mandatory, alternative, or optional. Each feature has a well-defined semantics. In addition, feature composition rules are described that define the valid configurations.

The main concepts from the feature model are explained in the following. The terms are ordered by order of appearance in the feature model.

- *Farm Management* Farm management covers all the aspects of running a farm.
- *Functionalities* A container concept for ERP, FMIS, and PASF. To show that for farm management, at least one functionality from one of these three groups is needed.
- *ERP Functionalities* Functionalities that appear in most businesses. For example, finance must be done by all organizations.
- *FMIS Functionalities* Functionalities that are specific for agriculture. These functionalities also appear in traditional farm management systems.
- *PASF* The Precision Agriculture-Specific Functionalities are the functionalities that define Precision Agriculture.
- *Sectors* Agriculture can be split into several sectors. The abstract feature Sectors allows for this division.
- *Arable Farming* Cultivation of field crops

- *Livestock Farming* The keeping of cattle, horses, poultry and similar animals for domestic use but not as pets
- *Horticulture* The study and practice of growing plants in orchards or greenhouses
- *Specialized Farming* A container concept for the farming techniques that cannot be directly categorized in one of the previously mentioned groups.
- *Reporting* Creating reports about planning, management, work progress, instructions, and orders.
- *Inventory* Monitoring and managing the inventory of the farm, which consists of production materials, equipment, chemicals, fertilizers, seeding, and planting materials.
- *Finance* With the feature finance, the costs and income estimation are made. Also, actual costs are noted, and finally, an evaluation of the farm's economic viability can be analyzed.
- *Traceability* Tracking and tracing the product but also the inventory. Traceability is reached with ID labelling and recording.
- *Sales Management* The management of orders and packing, also the transfer of expenses between enterprises and charges for services. Finally, accounting systems and costing systems.
- *Machinery Management* Equipment usage details and fleet management and logistics.
- *Human Resource Management* Employee management, the goal is the rapid, structured handling of issues concerning employees.
- *Quality Assurance & Performance Management* Quality assurance includes process monitoring and the production evaluation according to current legislative or contractual standards, and performance management allows for a better definition and management of enterprise indicators.
- *Production Planning* Production planning supports execution data acquisition and permits integration of the farm enterprise in supply chains.
- *Environmental Resource Management* Environmental Resource Management permits agriculture to reach sustainability by performing risk analysis and analyzing the impact on the environment. Another aspect of Environmental Resource Management is managing the influences of weather on the farm and reacting to these influences.
- *Best Practice Advise* “Best Practice Advise” offers the possibility of helping the farmers. It includes production tasks and methods for applying best practices according to agricultural standards.
- *Entity-Specific Features* Precision agriculture focusses on individual plants, animals or locations, to incorporate this aspect into the feature model.
- *Data Entry* Data Entry is entering the data into the FMIS, especially the digital transfer of data is important for precision agriculture.
- *Operations Management* Operations management includes the creation of operational plans, executing these plans, and incorporating the feedback into the next operational plans.

Domain definition

For developing software modules for precision agriculture requires both agricultural specific modules and general modules such as ERP functionality. Agriculture is a broad concept that can be decomposed in several ways. One way to specify is per type of agriculture; examples are biological, precision, and conventional agriculture. Another way is per sector. These different ways of decomposing agriculture can also be performed after each other.

Fig. 3 Meta-model of an agricultural business

Fig. 4 A context diagram for a precision agriculture system

In Fig. 3, the highest level of an agricultural business is modeled in a class diagram. Each Agricultural business has to perform general functions, just like every business. Agriculture is split into different types, each of these types has unique characteristics and requirements, but each also has common characteristics. Agriculture must deal with different sectors. The sectors and the agricultural types also relate to each other since an agricultural type can be performed in several sectors, and within one sector, several agricultural types can be distinguished. Since this study is specifically focused on Precision Agriculture, the other types of agriculture are out of the scope. The general functions are already well developed and researched in other fields (Sancar Gozukara et al. 2020), but for now, they are still taken into account for the domain model.

In Fig. 4, some of the general functions and agricultural functions are put into relation with stakeholders.

Feature model

The functionalities and concepts introduced in previous sections are used to draw a feature model (Fig. 5). The feature model is mostly self-explanatory, but some aspects are highlighted as follows:

The root feature Farm Management leads to two abstract features, namely *Functionalities* and *Sectors*. The feature Functionalities is used to show that in farm management, at least one of the Enterprise Resource Planning (ERP) (Sancar Gozukara et al. 2020), or Precision Agriculture-Specific Functionalities (PASF) should be used. The features in ERP Functionalities are important for Farm Management, and they show similarities with other

Fig. 5 A feature model for precision agriculture farm management

domains. The FMIS Functionalities are features that also appear in traditional farm management information systems. The last group is PASF; these functionalities are the ones that define precision agriculture.

Sectors are put at the same level as Functionalities since it has a large impact on the required functionalities for Farm Management. Sectors is an *inclusive or* feature because a farm must fit within at least one sector but can fit in multiple. The feature Specialized Farming is added in Sectors to act as a container concept since adding every specialized sector confuses the feature diagram, and it is difficult to guarantee that every sector is indeed added.

As can be seen in the legend of Fig. 5, some features are collapsed. The feature models for the collapsed features are shown in the following Figs. 6, 7, 8, 9, 10, 11 and 12.

In Fig. 6, the feature model for Arable Farming is shown. One of the three main features of arable farming is Field Operations Management, where the tasks related to arable farming are sub-features. The other features are Field and Crop.

In Fig. 7, the same is done for the feature Livestock Farming. The subgroups for livestock farming are divided using the abstract feature Type Definition. Livestock farming has General Care Operations, and each subgroup of livestock farming has specific care operations, for example, Milking and Shaving. The other main features for Livestock Farming are the Living Conditions and Herd Management.

In Fig. 8, the features Horticulture and Specialized Farming are explained. Horticulture can be performed inside in greenhouses or outside, for example, orchards. The features Crop and Field are both already elaborated in the feature Arable Farming, so these refer to the elaborated feature with either “ $\geq F$ ” or “ $-> F$ ”. In the feature Specialized Farming, some examples of specialized sectors are given.

In Fig. 9, the first set of ERP functionalities are shown in more detail. The features for each functionality are not extensive and complete but are given to show an example

Fig. 6 A feature model for the feature Arable Farming

Fig. 7 A feature model for the feature Livestock Farming

Fig. 8 A feature model for the features Horticulture and Specialized Farming

Fig. 9 A feature model for the first half of the features of ERP Functionalities

Fig. 10 A feature model for the second half of the features of ERP Functionalities

Fig. 11 A feature model for the feature Farm Management Information Systems Functionalities

Fig. 12 A feature model for the sub-features of precision agriculture-specific functionalities

of features related to the functionality. Traceability is following the product from production to consumption; for this product, labelling is necessary. The features of ERP functionalities are all-inclusive or features (together with the features from Fig. 10) since if ERP functionalities are used, it should be at least one of these functionalities.

The features UAVs and Autonomous Vehicles are sub-features of Machinery; they are unique for precision agriculture. The management of these features still is in Machinery Management, which is the reason why they are added to this location. For Machinery Management, the feature Machinery and Interoperability are mandatory. Interoperability is mandatory since it is important for precision agriculture that machinery has a good connection with other machinery and software.

In Environmental Resource Management from Fig. 11, the features that connect the farm to the environment are specified. An important aspect is weather since it has a major impact on farming. The features of Best Practice Advice show some of the aspects needed to provide advice on future actions. The sign “ $\geq F$ ” behind Data Entry means that this specific feature is elaborated in another location in the feature model.

Figure 12 shows the functionalities specific for precision agriculture, namely, Entity-Specific Feature, Data Entry, and Operations Management. The Entity-Specific features show how precision is considered in Farm Management. For Site-Specific, this is further expanded. For Operations Management, the following three steps are necessary: making plans, executing the plans, and incorporating the feedback.

Development of a DSL for precision agriculture

In this section the DSL framework is discussed. For this the possible DSL framework alternatives are described. The identified alternatives will then be evaluated using a defined evaluation framework, and a feasible DSL framework alternative will be selected. Subsequently, the structure, the syntax and the features of the DSL framework will be explained.

DSL framework alternatives

From the feature model and the motivations, three important aspects of a precision agriculture DSL framework were defined. In addition to “Functionalities” and “Sectors”, “Configuration” is added as an important aspect since the farm needs to be modeled to be able to understand and communicate about it. With these three elements, several DSL framework alternatives can be identified. These alternatives are visualized in Fig. 13 and then explained.

The first alternative (**A1**) is to make one big DSL. The specification for the functionality, the specification for each sector, and the way to configure a farm are put together. The advantage is that everything can be found in one location; a disadvantage is that it can easily become too large and difficult to maintain.

The second alternative (**A2**) is to make a DSL for each sector. One advantage is that specifications for other sectors can be left out of the DSL. One disadvantage is that redundancy increases.

Fig. 13 Six different design alternatives for DSL frameworks

The third alternative (**A3**) is to make a DSL for functionalities combined with DSLs specific for each sector, where the configuration of the farm is specified in the sector DSL. With the functionalities DSL, the general concept of each functionality can be specified. In the sector-specific DSLs, the general concept is transformed to match each sector. One advantage is that the functionalities are separated from the sectors. One disadvantage is that linking the sectors is a bit more difficult.

The fourth alternative (**A4**) is to make a DSL for functionalities, a DSL for sectors, and a DSL for configuration. The configuration DSL refers to the first two DSLs. This setup has as an advantage that the technical details of the sector and the functionality are separated from the farm configuration.

The fifth alternative (**A5**) is to do an additional split compared to A4; here, the sectors DSL is split into the different sectors; this decreases the possibilities of errors where, for example, animals are mentioned for an arable farm. However, this decreases the flexibility of mixed farms.

The sixth alternative (**A6**) is to combine the sectors DSL and the configuration DSL of A4. This allows more freedom and flexibility when making the farm model, but the farmers are exposed to more technical details.

There could be a seventh and eighth option, where the functionality DSL is combined with either the sector DSL or the configuration DSL. However, the farm configuration and the sector are more related to the farm than the functionalities; therefore, it makes less sense to make these combinations.

Evaluation framework

The framework that matches the requirements the best should be chosen. In Table 1, an analysis of the alternatives is performed. The alternatives are compared using the specific requirements determined. In addition to these requirements, an extra requirement is added, which states that the DSL framework should be easy to develop for the DSL developer/ language designer.

For each criterion, it is explained why that alternative received that scoring. For the framework evaluation, the following notations are used: ++, +, 0, −, and --. This notation can be transformed into a 5 Likert scale (5: ++, 4: +, 3: 0, 2: −, and 1: --). From

Table 1 Evaluation of framework alternatives against the specific requirements

Alternative	A1	A2	A3	A4	A5	A6
Criteria/Requirement						
1. Meeting trade-offs and complexity of the decision-making process	++	−	−	+	+	+
2. The uniqueness of domain concepts	++	--	−	+	0	+
3. Extensibility & adaptability	−	--	−	+	++	+
4. Simplicity for language users	+	--	−	++	++	0
5. Clear structure	−	0	+	++	+	0
6. Modularity	−	0	+	+	++	0
7. Balancing generality of the domain & uniqueness of sectors	++	+	+	++	++	++
8. Simplicity for the language designer	−	--	−	+	0	0

Table 1, it can be derived that the fourth alternative framework matches the requirements best; therefore, this DSL framework (A4) has been selected. This decision is made under the assumption that each criterion has the same weight. The evaluation is explained as follows.

Meeting trade-offs and complexity of the decision-making process of farmers

A1 received (++) for this requirement because there is no need to determine where certain complex parts are addressed between different DSLs; also, the interaction between sectors can easily be captured with one big DSL. A4, A5, and A6 received a (+) since there is a separation of concepts, but the complexity and trade-offs of the decision-making process can be captured in the configuration part of the framework. A2 and A3 received a (−) because farms with multiple sectors are hard to model with these two frameworks.

The uniqueness of domain concepts

A1 got (++) for this requirement since overlap in concepts can be prevented. Every concept is in the same DSL; therefore, they can be used for different functionalities and sectors or the configuration. A4 and A6 received a (+) for the uniqueness of domain concepts since they could have some overlap in the concepts of Functionality and Sector. A5 received a (0) since there could be overlap between the different sectors and between the sectors and functionality; however, there is no need to define the configuration part multiple times. A3 got a (−) since the configuration part of the framework is defined in each sector DSL. A2 got (−−) since the configuration part of the framework, and the functionality part of the framework are defined multiple times.

Extensibility and adaptability

A5 received (++) for extensibility and adaptability because to add a new sector, only a new DSL should be created to fit into the framework, and the connection to the configuration part of the DSL should be made. A4 and A6 received a (+) for this requirement because adding new specifications for a single sector requires a bit more work than for A5. A3 received a (−) for this requirement because changes to the configuration part of the DSL framework need to be implemented for different DSLs, which requires more work. A1 also got a (−) for extensibility and adaptability because a single change requires the whole DSL to be updated; additionally, it is tempting to make changes in the wrong location since everything is present in the single DSL. A2 received (−−) for this requirement since changes to the functionality part, and the configuration part needs to be changed in four different DSLs, which discourages changes (Table 2).

Simplicity for language users

Alternatives A2 and A3 have the advantage that the end-user can choose one sector and is not bothered with possibilities from other sectors. The corresponding disadvantage is that it is difficult to create connections between different sectors. If a farm has a livestock part and an arable part, it will be difficult to model this farm. Alternatives A1, A4, A5, and A6 allow for easier interaction between sectors since the configuration refers to the individual

Table 2 An overview of the pros and cons of each DSL framework based on the simplicity for the end-user

	A1	A2	A3	A4	A5	A6
Pros	Interactions between sectors	Choice in sector	Choice in sector	Interaction between sectors	Interaction between sectors	Interaction between sectors
	1 DSL			Separation of technical aspects from the configuration	Separation of technical aspects from the configuration	More flexibility
Cons	More flexibility	No linking between sectors	No linking between sectors	More flexibility	More flexibility	More difficult to address sector-specific concepts
	More difficult to address sector-specific concepts	Configuration and technical concepts combined	Configuration and technical concepts combined	More difficult to address sector-specific concepts	Configuration and technical concepts combined	Configuration and technical concepts combined

sectors or the sectors part is combined with the configuration part. Alternative A1 has the advantage that the end-user always knows which DSL to use since there is only one DSL. Alternatives A4 and A5 have the advantage that the technical aspects are separated from the configuration part. This is beneficial when the DSL user (software engineer) needs to communicate with the farmer. The farmer only needs to understand the configuration, and he is not bothered with the specification of the functionalities. The corresponding disadvantage for A1, A2, A3, and A6 is that the separation of technical details and configuration is less clear/not present.

For alternatives A1, A4, and A6, the sectors DSL is more general than those of A2, A3, and A5. This makes it a bit more difficult for the user to represent sector-specific concepts. An animal in a Livestock DSL can get a unique term while in a sectors DSL, it is a property. Thus, in a Livestock DSL, a specific treatment can be given to an animal. However, this generality of A1, A4, and A6 also gives the user more flexibility in showing in which way certain information is stored. The last disadvantage of alternative A2 is that the general functions of one sector are not usable in another sector. It is assumed that the advantages and disadvantages have the same weight; thus, A4 and A5 received (++), A1 received a (+), A6 received a (0), A3 received a (–), and A2 received a (––).

Clear structure

A4 received (++) for structure because the concepts from the feature model are split, but the interactions do not confuse the framework. A3 and A5 received a (+) because their separation of concepts makes the structure of the framework a bit more confusing. A2 and A6 received a (0) for structure since both have some separation, but the structure of the DSL framework becomes less clear. A1 received a (–) for clear structure because every concept is put in the same DSL; therefore, the underlying structure is not easy to see.

Modularity

A5 received (++) for modularity because all possible modules are created. A3 received a (+) for modularity since the sectors are separated into modules; however, each sector has the configuration in their module. A4 also received a (+). Here the configuration has an own module, but the sectors are combined into one module. A2 and A6 received a (0) since they are less modularized than A3 and A4 but more than A1. A1 received a (–) for modularity since there are no modules.

Balance generality of agricultural domain and uniqueness of the sectors

Each framework allows for the balance between the generality of the agricultural domain and the uniqueness of the sectors. However, since A2 and A3 do not allow for interactions between sectors, these two get a (+) while the rest gets (++).

Simplicity for language designers

A2 received (––) for simplicity for language designers because to design this framework, double work has to be done for each DSL. Additionally, the designer must think of a way to combine technical aspects and configuration so that it still offers flexibility for the user,

but a non-expert should also be able to understand the code. Finally, to develop this DSL, knowledge about all agricultural sectors is needed. A1 received a (–) because the designer needs to combine technical aspects with configuration. For this DSL framework, the whole DSL needs to be developed in one go, and in between tests are difficult. A3 received a (–) because again, the designer has to combine technical aspects and configuration, and he needs knowledge about the agricultural sectors. A5 received (0) for this requirement since it only has the disadvantage of the designer, which needs knowledge about each sector. A6 also received (0) because the designer has to think of a solution to combine technical aspects with configuration. A4 received a (+) because there are no direct disadvantages or advantages to this approach when looking from the language designer perspective.

Structure of the DSL

Several options for creating DSLs exist like Xtext, MetaEdit+, Spoofox, MPS JetBrains, and Dsl.Scala (Nascimento et al. 2012; Schmitt et al. 2014). In this project, the Xtext tool was chosen since it has no steep learning curve and works in the Eclipse platform, which is open source (Eclipse 2020).

The DSL is implemented so that java files are automatically created. Java has been chosen as the target language because of the Eclipse IDE ecosystem and the strong tool support for the Java language. The generated java files could be the basis of the program the farmer will use. Figure 14 shows how the DSL relates to the final program used by the farmer. The DSL consists of the following three parts: Functionality DSL, Sectors DSL, and Farm Model DSL. With the functionality DSL, the ERP, FMIS, and Precision Agriculture-Specific Functionalities (PASF) files are created. With the Sectors DSL, the files

Fig. 14 A visualization of entities used in the development process

Arable, Livestock, Horticulture, and Specialised are generated. Finally, the Farm Model DSL is used to build the Farm file. With the files from the second layer, the program for Software Use is made. This program consists of the java files that are automatically created with the DSL.

Functionality DSL

The functionality DSL is setup in a way that different functionality groups/folders can be created. After the determination of the group, the functionalities can be specified. For each functionality, tasks and properties can be determined. A property can, for example, be the data the functionality needs. A task is an action the functionality can perform. For a task, inputs can be given, comparable to inputs for Java methods. The functionality group becomes the package, and each functionality becomes a Java class.

Sector DSL

For the sector DSL, first, the name of the sector should be given. Later, a list of properties and tasks related to this sector can be created. A property can be any characteristic of the sector; tasks are actions related to the sector. It is not necessary to define the tasks and properties in a certain order.

Configuration DSL

For the configuration DSL, first, a name should be given to the farm configuration, which helps to differentiate between different configurations later on. After the name is given to the configuration, the sector and the functionalities groups can be specified. The order in which these are addressed is not important. The group specification starts with a line to mention from which group the following functionalities are. Then, the functionalities from this group can be listed. The sector specification starts with a statement where the sector is mentioned. Then, subsets are created to order the needed tasks and properties and to show that there are differences between, for example, the feeding of sheep and milk cows. These subsets also allow for using tasks and properties multiple times where needed. The file extension for the configuration DSL is “.farmmodel”. This extension was chosen because the term “model” is easier to understand for a non-expert.

Syntax of the DSL

For each of the developed DSLs, comments can be created in two ways. With the combination of a slash “/” with a star “*”, comments for multiple rows can be specified, at the end of the comment, the combination should be reversed to result in the following combination “*/”. Another way to specify comments is by using two slashes “//”, then a single line comment can be made.

Functionality DSL

In Fig. 15, an example of the functionality DSL syntax is given. The DSL can start with an import statement. However, most of the time, the term “package” is at the beginning of

Fig. 15 An example of the functionality DSL syntax applied to the Precision Agriculture-Specific Functionalities (PASF) group

```
package PASF{

    functionality OperationsManagement{
        task createPlans(){}
        task executePlans(): void {}
        task incorporateFeedback(){}
    }

    functionality EnterData{
        prop standardValues : java.util.List<?>
        task analyseData(){}
    }

    functionality PrecisePointPositioningSystem{
        prop location : int
        task determineLocation(int currentLocation){
            location = currentLocation
            return location
        }
        task applyVariable(String agent) : void{}
        task precisePlanting(String croptype){}
        task preciseSeeding(String croptype){}
    }
}
```

the file and should be used to specify the functionality group. The name of the functionality group should be all capitals. After a “{” multiple functionalities can be specified. This starts with the term “functionality” followed by the name of the functionality. The name of the functionality should start with a capital, where a new word in the functionality is addressed by starting the new word with a capital letter. After the name of the functionality again, a curly bracket is opened “{”. On a new line, the properties and tasks can be mentioned; there is no specific order to mention the tasks and properties. After specifying all the tasks and properties, a “}” is used to show that this is the end of this functionality, Eclipse automatically adds this closing curly brace. Once all the functionalities are determined, the package declaration should be closed with a closing curly brace “}”, again Eclipse adds this brace automatically.

To specify a property, the term “prop” should be used, then, the name of the property should be given, later a “:” symbol is used to show that the type of the property can be specified. The type should be a JVM reference type.

To specify a task, the term “task” should be used, followed by the name of the task. Then, “()” inputs for the task can be given, if there are no inputs, the brackets are empty. Input is specified by giving the JVM reference type of the input, followed by the name of the input. After the input specification, a colon “:” can be used to determine the type of the return statement or to specify that the task is a void method. Finally, between the curly brackets “{}”, some information about the body of the task can be given, e.g., a return statement.

Sector DSL

In Fig. 16, an example of the sector DSL syntax is given. Just as the functionality DSL, files generated with the sector DSL can start with an import statement; however, they will

```

package Horticulture{
  /* properties & tasks for inside horticulture */
  prop inside : String
  prop greenhouseType : Object
  task controlClimate(float currentTemperature, float currentHumidity){return String}

  /* crops for inside horticulture */
  prop tomato : Object
  prop cucumber : Object
  prop flower : Object
  prop lettuce : Object

  /* properties & tasks for outside horticulture*/
  prop outside : Object
  prop field : Object
  prop fieldBoundaries : Object
  prop area : Object
  task determineFieldBoundaries(){}
  task registrateFieldOperations(){}
  task prepareSoil(){}
}

```

Fig. 16 An example of the sector DSL syntax applied to the sector horticulture

often start with the package declaration in which the sector is defined. This goes in the same way as with the functionality DSL. Therefore, after the term “package” the sector name is given followed by an opening curly brace “{”. Eclipse adds the corresponding closing brace automatically. Between the curly braces, sector-specific properties and tasks can be specified. These specifications have the same syntax as the properties and tasks for functionalities.

Configuration DSL

In the files generated with the Configuration DSL, one could also start with an import statement. If this is not needed, the file starts with the term “Farmname”, followed by the name of the farm.

Then, the programmer gets the choice to first specify the functionalities or to specify the sector-specific qualities.

To specify the sector-specific qualities, first, the term “sect” is used. Then, a choice must be made between one of the sector package names specified in the files made with the sector DSL. Again, curly braces are used to show what belongs to the sector. Inside the curly braces, subsets are created to bring more structure in the configuration. A subset is a group of properties and tasks from the corresponding sector, which was specified above. The name of the subset is chosen from the tasks and properties in the corresponding sector; this option is chosen to allow the user more flexibility. A subset is started using the term “subset” followed by a property/task; then, curly braces are used. Inside the curly braces, multiple tasks and properties from the corresponding sector can be specified.

In the sector specification, multiple subsets can be determined. Also, below the farm name, multiple sectors can be used. Functionalities are grouped per functionality group. This also brings structure to the model.

To start specifying the needed functionalities from one functionality group, one should use the term “from” then the name of a functionality group, followed by the term “use” and curly braces. Inside the braces, the functionalities from the functionality

group can be chosen. This can be one or multiple functionalities from the functionality group. Just as one can use multiple sectors, one can use multiple functionality groups. For this, one must start with the term “from”.

The files for the functionality groups (and the functionalities in each functionality group) should be defined before using them in the configuration DSL. The same goes for the sector and the tasks/properties in the sector file. To call properties and tasks from a sector file and functionalities from a functionality group file, one should first use the name of the corresponding sector/functionality group followed by a dot “.”, then the name of the property/task/functionality can be called as shown in Fig. 17.

DSL features

The DSLs provide automatic suggestions for creating the farm configuration, the functionality files, or the sectors files. The automatic suggestion can show which elements of the DSL can be expressed next and which element the DSL expects from the user. Automatic suggestions can also show which elements from other files can be imported/used.

Another feature of this DSL framework is that it automatically generates java files based on the DSLs. For now, the tasks are transformed into Java methods, and the properties are converted to fields. Figures 18, and 19 shows examples of the java files generated with the sectors DSL and the configuration DSL.

Empirical evaluation of the DSL

A controlled experiment has been conducted to assess how the proposed DSL works in a realistic setting. In the following subsection, first the setup of the experiment is explained followed by the presentation of the results of the experiment. Finally, the potential threats to the validity of the experiment are addressed.

```

Farmname exampleFarm
sect Horticulture {
    subset Horticulture.general{
        Horticulture.apple
        Horticulture.applyFertilizer
        Horticulture.area : "10 ha"
    }
    subset Horticulture.inside {
        Horticulture.cucumber
        Horticulture.controlClimate
    }
}
from ERP use {
    ERP.Finance //managing money flows the rest is done by an accountant
    ERP.Inventory // for managing pesticides
}

from FMIS use {
    FMIS.BestPracticeAdvice // expected values compared with real values results in an advice
}

```

Fig. 17 An example of the configuration DSL syntax applied to an example farm

Fig. 18 A part of the java file created from the file *arable.sector*

```
import java.util.List;

@SuppressWarnings("all")
public class Arable {
    private Object general;

    public Object getGeneral() {
        return this.general;
    }

    public void setGeneral(final Object general) {
        this.general = general;
    }

    private List<Object> crops;

    public List<Object> getCrops() {
        return this.crops;
    }

    public void setCrops(final List<Object> crops) {
        this.crops = crops;
    }
}
```

Fig. 19 The java file created from the file *casestudyExample.farmmodel*

```
@SuppressWarnings("all")
public class exampleFarm {
    private String Horticulture;

    private Class Finance;

    public Class getFinance() {
        return this.Finance;
    }

    private Class Inventory;

    public Class getInventory() {
        return this.Inventory;
    }

    private Class BestPracticeAdvice;

    public Class getBestPracticeAdvice() {
        return this.BestPracticeAdvice;
    }
}
```

Experimental design

The purpose of this experiment is to *evaluate the applicability and usefulness of the developed precision agriculture DSL framework*, from the point of the *software engineer* and the *farmer*, when developing an *FMIS model*. The main research question for the controlled experiment is as follows: How applicable and useful is the proposed DSL when it is applied in a controlled experiment? To answer the main RQ above, three sub-questions are derived:

- RQ 1. To what extent is the DSL framework easy to learn and use?
- RQ 2. To what extent does the use of the DSL framework result in complete/high-quality models?
- RQ 3. Does usage of the DSL framework lead to more efficient in terms of required development time than the case where the DSL framework would not be used?

The case is the modelling of an FMIS for precision agriculture, where the software engineer needs to model a specific farm. For the controlled experiment sixteen students were involved who were studying their bachelors in Agrotechnology and MSc in Biosystems Engineering at Wageningen University, when this study was conducted (Summer 2019). Of these sixteen students, the DSL (treatment group) has been provided to eight students, and did not provide the DSL to the other eight students (control group). These groups of eight are split into groups of four where one group gets the assignment to model a livestock farm, and the other group has to model an arable farm.

The students from these studies were chosen as participants since most of them have some experience in programming. For their study, they followed the course Programming in Python at Wageningen University & Research (WUR). Additionally, also they have some basic knowledge of agriculture; thus, they can give feedback on both aspects of this DSL. To determine the groups, the students are asked four questions about their experience with FMIS, livestock farming, arable farming, and programming. Then, the students were randomly assigned to a group; therefore, the overall knowledge of each group was nearly the same.

The independent variables in this experiment are given as follows: whether the DSL is used or not and the type of farm that is modelled. The dependent variables analyzed in this experiment are the quality of the model and the effort needed for building the model. The quality of the model is assessed by the number of functionalities modelled, the functionalities modelled that were not needed, and the functionalities not modelled that were needed. The effort is quantified by measuring the time needed for making the model with a maximum of two hours, where the learning time is included. For the treatment group, an explanation of the DSL is considered learning. For the control group, analyzing the given examples and a list of possible functionalities is considered learning. This list is also incorporated into the DSL. In the experiment, the term model is used instead of the term configuration because participants are more familiar with this term.

To gather the data, the following procedure is followed. First, two farmers are contacted to supply information for the controlled experiment. In an initial meeting, they are asked to give a description of their farm and what functionalities they would want in an FMIS. This description is written down, and a model is created for this description. Then, the farmers were again contacted for a semi-structured interview. The improved descriptions are given to the participants of the controlled experiment, together with the assignment. Once the participants start the task, the time is noted, and they get a maximum of two hours. When the participant finishes the assignment, the time is noted, and he is asked to fill in a questionnaire. Afterward, the results of the control group and the treatment group are compared.

The results of the questionnaires are compared by taking the averages of the different groups and looking at the additional comments. The result of the assignments is compared by looking at the dependent variables.

Results

In this section, the results of the experiment are addressed. The results are addressed for each sub-research question.

RQ 1-Easiness of learning and usage

In the questionnaire, the participants were asked about their opinion on the ease of learning and using the DSL. The answers were given on a scale from 1 to 5, where 1 means strongly disagree and 5 means strongly agree.

Six of the eight participants of the treatment group agreed (value 4) with the statement that the DSL was easy to learn. One participant neither agreed nor disagreed; this participant found it hard to understand how everything works/would work. This was mainly about the underlying syntax and the next step in the development of the DSL. One participant did not agree with the statement that the DSL was easy to learn. He did not follow the provided syntax and mentioned that he “*had his own way of trying to get things structured and described*”. He said that he was “*too enthusiastic*” in starting with the program instead of taking the time to understand the syntax. This participant had neither agree nor disagree as his answer was, “*Once I learned the DSL, it was easy to use*”. The others all had an agreement or strong agreement.

For the covering of all the needs, there was a bit less agreement; the average was 3.4. The level of detail, the defining relations between tasks, functionalities, etc. and the need for more structure were mentioned.

RQ 2-Quality of models developed using the DSL

In Fig. 20, the quality assessment of the models is shown. The control group scaled the quality as average or low while in the treatment group, three participants scaled their model as high quality. The participant that assessed his model as the very low quality was the student who had trouble understanding the DSL.

Fig. 20 The assessment of the participants about the quality of their model

Livestock assignment

For the livestock assignment, 75 elements were used by the participants and the authors. 30 of these elements were not used by the authors but were introduced by the participants. Five of the elements were only used by the authors, for example, shaving the sheep. For the remaining 70 elements, it was checked if this element was used more often by the control group or the treatment group. The treatment group used 29 of the elements more than the control group. For 21 of the elements was the usage equal for both groups, and 20 elements were more used by the control group. The description above is also visualized in Table 3 and Fig. 21. In the figure, the darker colors of the outer ring are the elements used by the participants, and the author, the lighter colors of the outer ring are the elements only used by the participants.

On average, the treatment group used 7.25 more elements in their model than the control group.

In the livestock group, several elements were introduced by participants, namely 32 of the 75 elements. Most of these elements were deemed as unnecessary by the authors, namely, 21 elements. 11 elements could be an addition to the model of the authors but could also be interpreted as captured in other aspects of the authors' model. Of the 21 deemed unnecessary elements, 4 were used double in the sheep and the cow specification. Of the unnecessary elements introduced by the participants, 9 were used more by the treatment group, 9 were used more by the control group, and 3 were used equally. Of the necessary elements introduced by the participants, 5 were used more by the treatment group, 4 were used more by the control group, and 2 were used equally.

The number of necessary elements used was also analyzed. 54 elements were necessary, which consist of all the elements minus the 21 unnecessary elements. The authors used 42 of these elements, the treatment group used, on average 23.75 of these elements, and the control group used, on average, 20.75 of these elements.

Arable assignment

For the arable assignment, the total amount of elements was 47. Of these elements, five were introduced by participants and not used by the authors. Two of the elements were only used by the authors, but these two were only added after the second meeting with the farmer. Thus, it is understandable that none of the participants grasped these concepts. Also, here a check was done which elements were more used by the treatment group compared to the control group. 23 of the elements were more used by the treatment group. For 17 of the elements, the usage was equal, and for only five elements, the use by the control group was higher than that of the treatment group. This is also visualized in Table 4 and Fig. 22.

On average, the treatment group used 11.5 more elements in their model than the control group.

Of the five elements introduced by the participants, three were deemed unnecessary, namely, task package, traceability, and plant-specific treatment. The first one was used more by the treatment group; the last two were used equally. Of the other two elements, the first was "crop care", which was split into multiple tasks by the authors. This term was used equally by the control group and the treatment group. The other term which

Table 3 The distribution of the elements used during the livestock experiment

Total elements					
75					
More used by treatment group		More used by the control group		Equal usage	
29		20		21	
Also used by authors	Only participants	Also used by authors	Only participants	Also used by authors	Only participants
15	14	7	13	16	5
				Only used by authors	
				5	

Fig. 21 The distribution of the elements used during the livestock assignment

was not unnecessary was “drone”, this was used more by the control group than the treatment group.

The treatment group used on average, 26.5 of the needed 44 elements. The needed elements are all the elements minus the three unnecessary elements mentioned above. The control group used on average 19.25 of the needed elements.

RQ3-Efficiency and effort

In Table 5, the time needed to perform the case-study assignment is presented.

As seen in Table 5, the time needed to perform the assignment for the Livestock group was the same for the treatment and the control group. For the arable group, there were some differences. The higher time needed by the arable treatment group can partly be explained by the fact that one of the participants had dyslexia, and the participant who did not understand the DSL also belongs to this group. However, when looking at the time, each participant needed for an element the dispersion of the treatment groups is smaller than that of the control groups, and the time needed per element is lower, as shown in Fig. 23.

Both the treatment group and the control group agreed with the statement that the next model would be made much faster. However, the treatment group had more confidence in this statement, as shown in Fig. 24.

Potential threats to validity

In this section, the potential threats to the validity of the experiment are explained. The threats are grouped in the same groups as determined by Runeson and Höst (2008).

Construct validity

Regarding construct validity, there are possibilities for confusion. For example, the question regarding the structure and setup could be interpreted in a different way by the interviewed person. Another possible threat to construct validity is introduced by the experiment. Some

Table 4 The distribution of the elements used during the arable experiment

Total elements						
47	More used by treatment group		More used by the control group		Equal usage	
23	Only participants		Only participants		Only participants	
22	Also used by authors	1	Also used by authors	4	Also used by authors	2
				1	14	3

Fig. 22 The distribution of the elements used during the arable assignment, a visualization of Table 4

Table 5 Average time (minutes) needed to perform the case-study assignment per group

	Livestock	Arable	Total
Treatment	76.25	78.75	77.50
Control	76.25	63.75	70.00
Total	76.25	71.25	73.75

Fig. 23 The time needed per element for each of the case-study groups

interpreted the task as only having to mention all the aspects in the description, while others interpreted the task as having to mention all the possible aspects of an FMIS.

Internal validity

For internal validity, there are two threats. The first is that the reading and interpretation of participants are also tested. This threat appeared, especially in the arable assignment. The examples given to the participants were from horticulture. While in the description, arable

Fig. 24 Agreement of case-study participants to the question "I think that if I had to make more models, the next models would be made much faster"

was clearly mentioned, there were participants who used horticulture as a descriptive term for the sector. The second threat is that during the comparison of the model of the authors with the models of the participants, the experience with the DSL and the time could influence the quality of the model. However, none of the participants used the maximum allotted time, and especially in the arable assignment were all the elements of the authors at least used once by one of the participants.

External validity

The participants in this experiment were chosen with a specific concept in mind. Namely that the users of the language, in this case, the participants, all had some basic knowledge about the sector. It is not guaranteed that other software engineers will have the same basic knowledge.

Another aspect that was left out in the setup of this experiment is the interaction between the users, the farmer, and the software engineer. Since enhancing communication and understanding was one of the motivations, this remains an open issue. It was a little bit tested by the initial conversation and semi-structured interview with the farmers on which the assignments were based, but this was not further investigated.

Reliability

Due to time limitations, the description for the arable farm was not improved before the participants received the description. However, after the semi-structured interview was performed, only two major differences would have occurred. The buildings in which the storage is performed were not mentioned in the description. The other point was the general field operations, which include cleaning the ditches around the fields and moving the borders of the fields.

Another threat to reliability is the interpretation of the elements of the control group. Some of the elements of the control group had different names than the elements of the treatment group, while the same concept was meant. In the analysis, the authors aimed to

put these elements in the same location. Also, the interpretation of needed elements versus non-needed elements could vary if a different researcher analyses the data.

Discussion

The notion of DSLs has been broadly addressed in many different studies. For larger domains such as precision agriculture, more than one DSL is often needed. A coherent set of DSLs is defined as a DSL framework. A close analysis of the literature shows that this appears to be the first DSL framework for precision agriculture. The proposed DSL framework has been carefully planned, designed, implemented and evaluated.

The key concerns of DSLs have been explored in detail by analyzing the literature on both DSLs (Strembeck and Zdun 2009; Voelter 2013). and precision agriculture (Abbasi et al. 2014; Abt et al. 2006; Fountas et al. 2015; Tummers et al. 2019; Verdouw et al. 2019, 2016; Pinet et al. 2009). Obviously, the development of DSLs requires an upfront investment in time and costs. Developers of the DSL framework need to have a clear and thorough understanding of the domain as well as in language engineering (Kleppe 2009; Strembeck and Zdun 2009; Voelter 2013). Examples of cost-related disadvantages are the costs of design, implementation, maintenance, and education of the end-users (Mernik et al. 2005; Challenger et al. 2015; Fowler 2010; Voelter 2013).

To ensure that the proper concepts for the DSL framework are included, a thorough domain analysis process has been performed resulting in a feature diagram with the common and variant features of the precision agriculture domain. Since the domain is usually well-defined and stable, it is expected that the derived languages based on this domain model will also be stable. However, besides the proper extraction and modeling of the domain concepts, obviously, scoping is an important concern in designing the DSL framework. Scoping can be done in different ways. A too broad scope might be less useful in expressing the domain concepts while a too narrow scope might miss the flexibility. To define the proper scope, the feasible domain decompositions have been listed and here-with the DSL framework alternatives. To reason about and select the feasible DSL framework alternative, an evaluation framework has been defined. For the evaluation criteria, the requirements as described in the literature have been mainly adopted. If needed, some of the requirements could be eliminated, different weighing factors might be used for the criteria, or additional criteria can be provided. This could result in different DSL framework alternatives. The process for designing and selecting the DSL framework alternative, however, will not be that different.

The selected DSL framework has been implemented with the three DSLs. Further, this has also been evaluated using a controlled experiment. The threats to the validity of the experiments were explained in the previous section. Additional experiments and enhancements to the DSL frameworks can be provided in future work. A general advantage of DSLs is that they allow non-experts in programming to use a programming language. This means that farmers could understand and review software or models written with this DSL framework. The controlled experiment also showed this advantage since even a participant who had trouble with programming could eventually make a model.

Quantitative validation of the usefulness of the DSLs is inherently difficult, especially the trade-off between development costs and the increase in productivity. It is difficult to determine indicators for usefulness without measuring other influences.

Another important issue is the incorporation of the needs of the professionals in the motivations and requirements for the DSL framework. The developed DSL framework was not tested on the final users, but on a group, which could judge both aspects of the DSL framework. However, additional tests could be needed in the future to confirm and justify the findings of this study based on broader stakeholder perspectives.

Conclusions and future work

Precision agriculture is gaining more interest and momentum in the last decade with the advancements in enabling technologies. An important domain that is impacting precision agriculture is software engineering that is needed to develop the required smart software modules. General-purpose languages can be used to develop these modules, but these are agnostic to the application domain and less expressive with respect to the specific precision agriculture concerns. Domain-specific languages (DSLs) can be used to reduce the conceptual gap between the application domain and the implementation. DSLs have several apparent benefits, such as easier communication, faster development, better interoperability, and higher productivity. Precision agriculture is also dependent on the farm management information systems (FMISs) that can be leveraged using DSLs.

In this article, a DSL framework, a coherent set of DSLs dedicated to the precision agriculture domain has been proposed. The design of the DSL framework has been explicitly described, and it has been shown that multiple design alternatives could be derived. The evaluation framework helped to make explicit design decisions and turned out to be very useful in selecting the most feasible DSL framework. From the six identified alternatives, the DSL framework consisting of three DSL for functionalities, a DSL for sectors, and a DSL for configuration were selected. The primary benefit of this DSL framework was that the technical details of the sector and the functionality are separated from the farm configuration, which is beneficial for various practical reasons.

To assess the applicability and usefulness of the developed DSL framework, a controlled experiment was performed. The controlled experiment showed that the DSL framework with the three DSL languages is both effective and practical. It was shown that the DSLs are easy to use and learn with small time investment. The quality of the resulting configurations/ models was also higher for the treatment group when looking at the usage of needed elements.

The DSL framework can be extended for additional concerns or to align with the evolution of the corresponding domains. Other projects might even select other DSL framework alternatives than that were chosen. This could depend on the defined evaluation criteria and the subjective evaluation of these criteria. In any case, it has been shown that a DSL framework for precision agriculture can provide several benefits in practice. The future work will further elaborate on the DSL framework and apply it to various other farming contexts.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Abbasi, A. Z., Islam, N., & Shaikh, Z. A. (2014). A review of wireless sensors and networks' applications in agriculture. *Computer Standards & Interfaces*, 36(2), 263–270. <https://doi.org/10.1016/j.csi.2011.03.004>
- Abt, V., Perrier, E., & Vigier, F. (2006). Towards an integration of farm enterprise information systems: A first analysis of the contribution of ERP systems to software function requirements. (2006). *Computers in Agriculture and Natural Resources*, 23–25 July 2006, Orlando Florida, 2006, p. 469. <https://doi.org/10.13031/2013.21919>
- Ananthanarayanan, V., & Thies, W. (2010). Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering*, 4(1), 13. <https://doi.org/10.1186/1754-1611-4-13>
- Arkin, E., Tekinerdogan, B., & Imre, K. (2013). Model-driven approach for supporting the mapping of parallel algorithms to parallel computing platforms. Proc. of the ACM/IEEE 16th International conference on model driven engineering languages and systems. https://doi.org/10.1007/978-3-642-41533-3_46
- Bakker, J., Tekinerdogan, B., & Aksit, M. (2005). Characterization of early aspects approaches, in Proc. of early aspects: Aspect-oriented requirements engineering and architecture design workshop, AOSD.
- Bilitchenko, L., Liu, A., Cheung, S., Weeding, E., & Xia, B. (2011). Eugene—A domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE*, 6(4), e18882. <https://doi.org/10.1371/journal.pone.0018882>
- Challenger, M., Kardas, G., & Tekinerdogan, B. (2015). A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*, 24, 1–41. <https://doi.org/10.1007/s11219-015-9291-5>
- Channe, H., Kothari, S., & Kadam, D. (2015). Multidisciplinary model for smart agriculture using internet-of-things (IoT), sensors, cloud-computing, mobile-computing & big-data analysis. *International Journal of Computer Technology & Applications*, 6(3), 374–382.
- Doering, D., Vizzotto, M. R., Bredemeier, C., da Costa, C. M., Henriques, R. V. B., Pignaton, E., & Pereira, C. E. (2016). MDE-based development of a multispectral camera for precision agriculture. *IFAC-PapersOnLine*, 49(30), 24–29. <https://doi.org/10.1016/j.ifacol.2016.11.117>
- Doering, D., Pereira, C. E., Denes, P., & Joseph, J. (2013). A model driven engineering approach based on aspects for high speed scientific X-rays cameras. Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pp. 1–8. <https://doi.org/10.1109/ISORC.2013.6913190>
- Eclipse, (2020). Integrated development environment (IDE). <https://www.eclipse.org>. Accessed October 2020
- Fall, A., & Fall, J. (2001). A domain-specific language for models of landscape dynamics. *Ecological Modelling*, 141(1), 1–18. [https://doi.org/10.1016/S0304-3800\(01\)00334-9](https://doi.org/10.1016/S0304-3800(01)00334-9)
- de Farias, C. M., Brito, I. C., Pirmez, L., Delicato, F. C., Pires, P. F., Rodrigues, T. C., et al. (2017). COMFIT: A development environment for the internet of things. *Future Generation Computer Systems*, 75, 128–144. <https://doi.org/10.1016/j.future.2016.06.031>
- Fountas, S., Carli, G., Sørensen, C. G., Tsiropoulos, Z., Cavalaris, C., Vatsanidou, A., et al. (2015). Farm management information systems: Current situation and future perspectives. *Computers and Electronics in Agriculture*, 115, 40–50. <https://doi.org/10.1016/j.compag.2015.05.011>
- Fowler, M. (2010). Domain-specific languages. Addison Wesley Professional, ISBN: 978-0321712943
- Gaucherel, C., Giboire, N., Viaud, V., Houet, T., Baudry, J., & Burel, F. (2006). A domain-specific language for patchy landscape modelling: The brittany agricultural mosaic as a case study. *Ecological Modelling*, 194(1–3), 233–243. <https://doi.org/10.1016/j.ecolmodel.2005.10.026>
- Gogos, C., Alefragis, P., & Housos, E. (2007). Sensor-enabled rule-based alarm system for the agricultural industry, in IEEE Conference on emerging technologies and factory automation, pp. 912–915. <https://doi.org/10.1109/EFTA.2007.4416880>
- Kernecker, M., Knierim, A., & Wurbs, A. (2020). Experience versus expectation: Farmers' perceptions of smart farming technologies for cropping systems across Europe. *Precision Agriculture*, 21, 34–50. <https://doi.org/10.1007/s11119-019-09651-z>
- Kleppe, A. (2009). *Software language engineering: Creating domain-specific languages using metamod-els*. Inc, Boston: Addison-Wesley Longman Publishing Co.
- Kolovos, D.S., Paige, R.F., Kelly, T., & Polack, F.A. (2006). Requirements for domain-specific languages. in Proc. of ECOOP workshop on domain-specific program development (DSPD 2006)
- Köksal, Ö., & Tekinerdogan, B. (2019). Architecture design approach for IoT-based farm management information systems. *Precision Agriculture*, 20(5), 926–958. <https://doi.org/10.1007/s11119-018-09624-8>

- Lindblom, J., Lundström, C., Ljung, M., & Jonsson, A. (2017). Promoting sustainable intensification in precision agriculture: Review of decision support systems development and strategies. *Precision Agriculture*, 18, 309–331. <https://doi.org/10.1007/s11119-016-9491-4>
- McBratney, A., Whelan, B., Ancev, T., & Bouma, J. (2005). Future directions of precision agriculture. *Precision Agriculture*, 6(1), 7–23. <https://doi.org/10.1007/s11119-005-0681-8>
- Mernik, M., Heering, J., & Sloane, A. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 316–344. <https://doi.org/10.1145/1118890.1118892>
- Nash, E., Korduan, P., & Bill, R. (2009). Applications of open geospatial web services in precision agriculture: A review. *Precision Agriculture*, 10(6), 546. <https://doi.org/10.1007/s11119-009-9134-0>
- Nash, E., Dreger, F., Schwarz, J., Bill, R., & Werner, A. (2009). Development of a model of data-flows for precision agriculture based on a collaborative research project. *Computers and Electronics in Agriculture*, 66(1), 25–37. <https://doi.org/10.1016/j.compag.2008.11.005>
- Nikkilä, R., Seilonen, I., & Koskinen, K. (2010). Software architecture for farm management information systems in precision agriculture. *Computers and Electronics in Agriculture*, 70(2), 328–336. <https://doi.org/10.1016/j.compag.2009.08.013>
- Nordmann, A., Hochgeschwender, N., & Wrede, S., et al. (2014). A survey on domain-specific languages in robotics. In D. Brugali (Ed.), *International conference on simulation, modeling, and programming for autonomous robots* (pp. 195–206). Cham: Springer.
- Nascimento do, L. M., Viana, D.L., Neto, P.A.S., Martins, D. A., Garcia, V. C., & Meira, S. R. (2012). A systematic mapping study on domain-specific languages. In Proceedings of the 7th International conference on software engineering advances (ICSEA'12), pp. 179–187.
- Noppen, J., Aksit, M., Tekinerdogan, B., & Nicola, V. (2004). Market-driven approach based on Markov decision theory for optimal use of resources in software development. *IEE Proceedings- Software*, 151(2), 85–94. <https://doi.org/10.1049/ip-sen:20040079>
- Oliveira, N., Pereira, MJV., Henriques, PR., & Cruz, D. (2009). Domain-specific languages: a theoretical survey. In Proceedings of INForum, CoRTA2009 (compilers, programming languages, related technologies and applications, Lisbon, Portugal, pp. 35–46.
- O'Grady, M. J., & O'Hare, G. M. (2017). Modelling the smart farm. *Information Processing in Agriculture*. <https://doi.org/10.1016/j.inpa.2017.05.001>
- Paraforos, D. S., Vassiliadis, V., Kortenbruck, D., Stamkopoulos, K., Ziogas, V., Sapounas, A. A., & Griepentrog, H. W. (2016). A farm management information system using future internet technologies. *IFAC-PapersOnLine*, 49(16), 324–329. <https://doi.org/10.1016/j.ifacol.2016.10.060>
- Pinet, F., Roussey, C., Brun, T., & Vigier, F. (2009). The use of UML as a tool for the formalisation of standards and the design of ontologies in agriculture. In P. Papajogji (Ed.), *Advances in modeling agricultural systems* (pp. 131–147). Bostan: Springer.
- Rains, G. C., & Thomas, D. L. (2009). Modelling farming: An introduction. University of Georgia Cooperative Extension, University of Georgia
- Runeson, P., & Höst, M. (2008). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), 131. <https://doi.org/10.1007/s10664-008-9102-8>
- Sancar Gozukara, S., Tekinerdogan, B., & Catal, C. (2020). Obstacles of on-premise enterprise resource planning systems and solution directions. *Journal of Computer Information Systems*. <https://doi.org/10.1080/08874417.2020.1739579>
- Schmitt, C., Kuckuk, S., Köstler, H., Hannig, F., & Teich, J. (2014). An evaluation of domain-specific language technologies for code generation, in computational science and its applications (ICCSA), 2014 14th International conference on, 2014, pp. 18–26. <https://doi.org/10.1109/ICCSA.2014.16>
- Shaffer, M., Bartling, P., & Ascough, J. (2000). Object-oriented simulation of integrated whole farms: GPFARM framework. *Computers and Electronics in Agriculture*, 28(1), 29–49. [https://doi.org/10.1016/S0168-1699\(00\)00117-4](https://doi.org/10.1016/S0168-1699(00)00117-4)
- Stahl, T., & Voelter, M. (2006). Model-driven software development, Addison-Wesley. ISBN: 978-0470025703
- Strembeck, M., & Zdun, U. (2009). An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15), 1253–1292. <https://doi.org/10.1002/spe.936>
- Tekinerdogan, B., & Demirli, E. (2013). Evaluation framework for software architecture viewpoint languages. in Proc. of ninth International ACM sigsoft conference on the quality of software architectures conference (QoSA 2013), Vancouver, Canada, pp. 89–98, June 17–21.
- Tekinerdogan, B., & Köksal, Ö. (2018). Pattern based integration of internet of things systems. In D. Georgakopoulos & L. J. Zhang (Eds.), *Internet of things—ICIOT 2018*. Lecture notes in computer science. vol. 10972, Cham: Springer.

- Tekinerdogan, B., & Arkin, E. (2019). ParDSL: A domain-specific language framework for supporting deployment of parallel algorithms. *Software & Systems Modeling*, 18, 2907–2935. <https://doi.org/10.1007/s10270-018-00705-w>
- Tekinerdogan, B., Bilir, S., & Abatlevi, C. (2005). Integrating platform selection rules in the model driven architecture approach. In U. Aßmann (Ed.), *Model driven architecture* (pp. 159–173). Berlin: Springer.
- Tummers, J., Kassahun, A., & Tekinerdogan, B. (2019). Obstacles and features of farm management information systems: A systematic literature review. *Elsevier Computers and Electronics in Agriculture*, 157, 189–204. <https://doi.org/10.1016/j.compag.2018.12.044>
- Tummers, J., Kassahun, A., & Tekinerdogan, B. (2020). Reference architecture design for farm management information systems: A multi-case study approach. *Precision Agriculture*. <https://doi.org/10.1007/s11119-020-09728-0>
- Tüzün, E., Tekinerdogan, B., Kalender, M. E., & Bilgen, S. (2015). Empirical evaluation of a decision support model for adopting software product line engineering. *Information and Software Technology*, 60, 77–101. <https://doi.org/10.1016/j.infsof.2014.12.007>
- Van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6), 26–36.
- Verdouw, C., Sundmaeker, H., Tekinerdogan, B., Conzon, D., & Montanaro, T. (2019). Architecture framework of IoT-based food and farm systems: A multiple case study. *Computers and Electronics in Agriculture*, 165, 104939. <https://doi.org/10.1016/j.compag.2019.104939>
- Verdouw, C., Wolfert, J., & Tekinerdogan, B. (2016). Internet of things in agriculture. *CAB Reviews: Perspectives in agriculture, Veterinary Science, Nutrition and Natural Resources*. <https://doi.org/10.1079/PAVSNNR201611035>
- Vaquero-Melchor, D., Palomares, J., Guerra, E. & de Lara, J. (2017). Active domain-specific languages: Making every mobile user a modeller, ACM/IEEE 20th International conference on model driven engineering languages and systems (MODELS), 2017, pp. 75–82. <https://doi.org/10.1109/MODELS.2017.13>
- Voelter, M. (2013). DSL engineering: Designing, implementing and using domain-specific languages, createspace independent publishing platform. ISBN: 97801481218580
- Wang, M., Cao, J., Li, J., & Dasi, S. K. (2008). Middleware for wireless sensor networks: A survey. *Journal of Computer Science and Technology*, 23, 305–326. <https://doi.org/10.1007/s11390-008-9135-x>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.