Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

End-user development by application-domain configuration

Alistair Sutcliffe[a,*], George Papamargaritis[b]

[a] Manchester Business School, University of Manchester, Booth Street West, Manchester M15 6PB, UK
[b] Intracom IT Services, Markopolou Avenue, Athens GR 19002, Greece

ARTICLE INFO

Article history:
Received 8 February 2013
Received in revised form 7 October 2013
Accepted 30 November 2013
Available online 8 January 2014

Keywords:
End-user development
Application generation
Domain-oriented design

ABSTRACT

An application generator/tailoring tool aimed at end users is described. It employs conceptual models of problem domains to drive configuration of an application generator suitable for a related set of applications, such as reservation and resource allocation. The tool supports a two-phase approach of configuring the general architecture for a domain, such as reservation-booking problems, then customisation and generation of specific applications. The tool also provides customisable natural language-style queries for spatial and temporal terms. Development and use of the tool to generate two applications, service engineer call allocation, and airline seat reservation, are reported with a specification exercise to configure the generic architecture to a new problem domain for monitoring-sensing applications. The application generator/tailoring tool is evaluated with novice end users and experts to demonstrate its effectiveness.

1. Introduction

End-user software development is receiving increasing attention as the number of end users practising software development is predicted to rise (Ko et al., 2011). While tools to generate user interfaces, such as screen painters and report writers, are familiar facilities for end users, creation of software applications has remained in the realm of expert programmers. More advanced end-user development (EUD) tools have ranged from visual programming languages to enhanced spreadsheets and graphical design environments (Fischer, 1994). Another approach (Repenning and Ioannidou, 2004) has been to facilitate end-user development of interactive simulations with rule-driven agents. While these tools can help end users to develop educational and entertainment-oriented applications, they do not appear to scale to business domains or complex software engineering problems. While some genres of domain-specific tools with programmable scripting languages have been successful, notably spreadsheet (Burnett, 2009) and database (SQL) programming (Batory and Geraci, 1997), end users in most domains still have to learn conventional programming languages.

Model-driven architectures have produced a range of tools which could potentially empower EUD. However, these tools (e.g. executable UML: Mellor and Balcer, 2002) rely on expert knowledge for specification in semi-formal notations and an action specification language for procedural detail. Action specification languages follow the syntax and semantics of conventional programming languages and present a considerable barrier for end users (Ko et al., 2011). Similar problems are encountered with component-based software engineering, where glue code has to be written to interface components, or components have to be customised with a scripting language, as in ERPs (Keller and Teufel, 1998).

EUD has been defined as "a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact" (Lieberman et al., 2006). However, the boundary between language-oriented development (Batory et al., 2002; Myers et al., 2004) and customising, configuring (Eagan and Stasko, 2008), and tailoring (Pipek and Kahler, 2006) software is blurred, where the latter tend to involve parameterisation of existing programmes, rather than direct modification of a program's source code. In this paper we describe application generation and tailoring with an end-user interface that does not require any specialist knowledge of programming concepts. We therefore describe our EUD approach as an application generator-tailoring tool.

In contrast to end-user programming and end-user software engineering (Burnett, 2009; Ko et al., 2011) we aim to hide the complexities of programming and scripting from the users, and instead facilitate application development via user-friendly graphical interfaces. Question and answer dialogues with form filling are used to capture user requirements in two phases, first by customising a generic architecture with components selected from related domains; then end-user requirements are expressed directly in a dialogue with the generated application. We explore two approaches to bridge the communication gap in EUD: (a) use of generic conceptual models of application domains, and (b) application customisation using diagrams representing real-world

* Corresponding author. Tel.: +44 0161 306 3315.
E-mail addresses: ags@manchester.ac.uk, ags@man.ac.uk (A. Sutcliffe), George.Papamargaritis@intracom-it.com (G. Papamargaritis).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.11.1121

domains. In following sections we describe related work; the design and software architecture of an end-user oriented application tailoring tool (EATT); its implementation; usability evaluation with novice end users; and configuration of the architecture to new domains. The paper concludes with a discussion comparing our approach with other EUD tools.

2. Related work

End-user development has followed many paths, ranging from domain-specific high-level languages (Batory et al., 2002; Freeman, 1987; Neighbors, 1984) to high-level domain-oriented design environments (DODEs) (Fischer, 1994) that take a reuse component configuration approach; and hybrid environments composed of graphical objects, simple rule scripting and interaction to specify programmes (Lieberman et al., 2006; Repenning and Ioannidou, 2004; Ioannidou et al., 2009).

EUD via domain-specific languages was supported by high-level compilers or application generators, such as Draco (Freeman, 1987; Neighbors, 1984), that produced applications by enabling designers to create domain-tailored specification languages which could then be used to generate software systems within the same domain. The reusable Draco domains consisted of an abstract language containing objects and functions with alternative implementation routes, and a transformation engine to select optimal implementations. However, the scope of Draco was limited by the hierarchy of implemented domain languages and mappings to executable components. KIDS (Smith, 1990) generated applications from formal specification of functions and high-level transformations. High-level requirements were expressed in a set-theoretic formalism which required programming expertise.

GenVoca (Batory and Geraci, 1997; Batory et al., 2000, 2002) used domain analysis to specify a grammar consisting of realms and components, which was used to generate libraries of data structures, databases and graphical components. Realms modelled the problem, organising systems into parameterised layers. Components implemented alternative functional refinements, so the generator configured a multi-layer architecture by extending component templates where extensions of components were formulated as type equations. This required adoption of a template-based programming style (Batory et al., 2000) which excludes end users, who have to understand the syntax and semantics of the GenVoca grammar.

Restricted natural language has been advocated as one way of escaping from the formal language trap for end users. The unified approach (Zhisheng et al., 2002) supported a query-based approach to application generation. End users submitted their requests for new applications as queries, using an SQL-like natural language which combined a domain-specific sub-language for expressing requirements (e.g. banking) and quality of service constraints (e.g. end-to-end delay, throughput). However, reuse was limited since a new domain-specific sub-language had to be constructed for each application.

A similar approach used a question–answer (QA) agent (Yoshida et al., 2004) to capture the user requirements and translate them into abstract classes and generation rules. The QA agent matched the users' answers to design rules that generated specific applications from a product-line component library. Although the QA interface was natural language-based it assumed knowledge of a formal domain ontology. Explore/L (Markus and Fromherz, 1994) provided a natural language interface with a restricted vocabulary and syntax consisting of simple declarative sentences (i.e. subject–verb–object) for expressing requirements. Although the templates for expressing classes of requirements were general they had to be customised with a semantic lexicon of the domain, and interpreters developed to map user requirements to components for application generation. Furthermore, the range of requirements expressions was limited by the set of templates and interpreters provided.

More recently, service-oriented EUD approaches have proposed domain-specific languages for e-government services (Fogli and Parasiliti Provenza, 2012) and navigation with location-based services in mobile applications (Stav et al., 2013). While the domain languages and form-filling menu-driven interfaces of these systems are more end-user friendly, they do rely on domain experts to specify domain components and configure development environments for end-user customisation in relatively restricted domains. General EUD environments based on task-tree diagram specifications have also been produced for customising service applications for web and mobile platforms (Paternò et al., 2011); a case study for genres of reservation applications is reported. Similarly, EUD languages and customisation support for a reservation/allocation task based on design patterns has been proposed by Seffah and Ashraf (2007). While these systems enable relatively easy development by menu/diagram-based interfaces, they do rely on extensive, prior configuration by domain experts or service component developers.

Spreadsheets as an exemplar end-user tool have been extensively researched by Burnett et al. (2002), whose aim was to improve the correctness of user-generated script by algorithms that detected user errors from their interaction with spreadsheets. This approach is limited to configuration and validation of spreadsheets, rather than a wider range of applications. Domain-oriented design environments (DODEs: Fischer, 1994; Fischer et al., 2004) eliminate the need for high-level languages by taking a component tailoring and reuse approach. Components and design patterns are represented in graphical forms mapped to the domain, then end-user development is supported by explanation facilities and critics which guide the users' tailoring activities. Fischer argues that the DODE concept transcends end-user development towards 'meta design' where the user is engaged in a domain-oriented design activity, insulated from coding and computational semantics (Fischer et al., 2009). However, DODEs do require configuration with components and knowledge within a restricted range of applications (or a domain) in a seeding cycle (Fischer, 1998).

Graphical EUD environments generally require less specialist programming knowledge from end users, since they reply on interaction with graphical objects to specify computation semantics (Lieberman et al., 2006). A wide variety of notations (e.g. entity relationship, data flow, flowchart diagrams) have been proposed for graphical EUD as well as domain specific notations for CAD (computer aided design) environments (Hale et al., 2012). Graphical EUD has been based on agent-based paradigms and graphic templates for constructing rules (Repenning and Ioannidou, 2004), while Myers et al. (2004) made extensive use of graphical metaphors to suggest natural programming concepts (ALICE). End users create and designate graphical objects with editors and then produce behaviour by a combination of rule-based specification and user interface manipulations from which the environment infers rules, as 'programming by example' However, these agent rule-based programming environments have mainly been targeted at educational applications, games and applications where physical objects are present. The graphical approach may be more difficult to apply to information systems that involve transactions with virtual objects.

Although previous research has moved some way towards end-user development tools which hide programming from end users, there is a contrast between application generators that span a wide range of applications but employ either restricted natural language or formal specification languages; and graphical EUD environments which have concentrated on education and games-style environments. The challenge we address is to preserve the