Dear Author,

Please check your proof carefully and mark all corrections at the appropriate place in the proof (e.g., by using on-screen annotation in the PDF file) or compile them in a separate list. Note: if you opt to annotate the file with software other than Adobe Reader then please also highlight the appropriate place in the PDF file. To ensure fast publication of your paper please return your corrections within 48 hours.

For correction or revision of any artwork, please consult http://www.elsevier.com/artworkinstructions.

Any queries or remarks that have arisen during the processing of your manuscript are listed below and highlighted by flags in the proof. Click on the ‘Q’ link to go to the location in the proof.

<table>
<thead>
<tr>
<th>Location in article</th>
<th>Query / Remark: click on the Q link to go</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>The reference given here is cited in the text but is missing from the reference list – please make the list complete or remove the reference from the text: Fogli et al. (2012) and Repenning and Ioannidou (2006), Lieberman et al. (2004), Freeman (1978), Fischer (2009).</td>
</tr>
<tr>
<td>Q2</td>
<td>Please confirm that given names and surnames have been identified correctly.</td>
</tr>
<tr>
<td>Q3</td>
<td>The number of keywords provided exceeds the maximum allowed by this journal. Please delete three keywords.</td>
</tr>
<tr>
<td>Q4</td>
<td>Ref. Fischer (2009) is cited in the text but not provided in the reference list. Please provide it in the reference list or delete this citation from the text.</td>
</tr>
<tr>
<td>Q5</td>
<td>Ref. Freeman (1978) is cited in the text but not provided in the reference list. Please provide it in the reference list or delete this citation from the text.</td>
</tr>
<tr>
<td>Q6</td>
<td>Ref. Lieberman et al. (2004) is cited in the text but not provided in the reference list. Please provide it in the reference list or delete this citation from the text.</td>
</tr>
<tr>
<td>Q7</td>
<td>Refs. Fogli et al. (2012) and Repenning and Ioannidou (2006) are cited in the text but not provided in the reference list. Please provide them in the reference list or delete these citations from the text.</td>
</tr>
</tbody>
</table>
| Q8                  | Please supply the name of the city of publication for reference “Smith (1990)”.

Thank you for your assistance.
End-user development by application-domain configuration

Alistair Sutcliffe*, George Papamargaritis

- Software tool helps end users develop applications without knowledge of programming languages.
- System provides 'seeding' of domain-oriented design environments for end-user development.
- Novel graphical user interface facilitates specification of temporal and spatial constraints for reservation types of applications.
- Two-phase approach provides a generic architecture which can be configured for domains, which then support end-user customisation and application generation.
- Semi-automatic generation of systems for reservation/allocation applications by component composition and reuse.
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, Markopoulo Avenue, Athens GR 19002, Greece

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.

Keywords: End-user development, Application generation, Domain-oriented design, Graphical user interface, Reservation applications

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 "an end-user tool that allows software development to be performed by non-developers," 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 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 ([Zhisheg 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-free 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 [Seifah and Ashraf (2007)]. While these systems enable relatively easy development by menu/diagram-based interfaces, they 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 rely 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 graphical environment ease of use while extending generation to a wider range of domains.

3. Domain-oriented end-user development

The approach we investigate is to develop the domain-oriented approach to support end-user development (Fischer, 1998; Fischer et al., 2009), but overcome the ‘seedling’ problem by a new configuration tool based on generic conceptual models of problem domains. We explore how the domain configuration limitations of DODEs might be overcome in a two-phase approach:

(i) Configure a generic application generation environment with high-level conceptual models.

(ii) Specialise a selected high-level model with more domain-specific information to enable generation of a concrete application.

Given the two-phase approach, we address two types of end user: first, end users who have some knowledge of computing, even though programming and software development is not their main professional role; and secondly, end users who have little or no knowledge of computing. The latter group are true end users, while the first group may be regarded as intermediary end users. The development strategy was to create tools so intermediary users could generate applications from software templates and then true end users could customise the generated applications without any specialist knowledge. This contrasts with the end-user software engineering view of users as technical specialists who create programmes for their own use, where software tools are intended to improve software quality (Ko et al., 2011; Burnett, 2009).

Our approach to the seeding problem is to first configure the environment with generic models covering a wide range of possible applications, in this paper, allocation/reservation-type problems, although we have researched generic conceptual models for other domains such as hiring, sales transactions, etc. (Sutcliffe, 2002; Sutcliffe et al., 2006). In the second phase a domain expert elaborates the seed with more domain-specific detail by specialising classes and adding attributes. The seeded environment is then ready for end users to generate a range of applications within the scope of the generic models.

Following our aim of hiding the complexities of programming from end users, application generation is based on component reuse, with user interface facilities for component search and tailoring that combine natural language processing and programming by demonstration in graphical user interfaces.

The process can be summarised in the following research question:

Can the gap between programming and end-user development be bridged in two phases: configuring high-level conceptual models, followed by domain customisation and automated application development?

3.1. Process overview

This section introduces the EUD process based on the two-phase approach. The process sequence is summarised in Fig. 1, which illustrates the steps and tool support in configuring the EUD environment leading to application development. An interactive configuration/customisation approach is adopted so steps 3–9 are manual tasks supported by editor tools, leading to full automatic application generation in step 10.

The process starts by the domain expert/configurer deciding the terms of reference or scope of the applications to be generated. In the case study this is reservation/allocation genre applications. The select domain model process (step 1) is supported by the end-user model library and a simple browser-editor tool which helps the domain expert find the most appropriate domain model family for the target range of applications. In step 2 the domain expert specialises the selected generic domain model by adding high-level attributes from an analysis of existing domain languages. Step 3 involves configuring the database and algorithms for the application generator, by selecting algorithms from the appropriate repository and customising them for the database schema, i.e. range of data types, expected values, etc. This step is supported by editor tools for configuring the database schema and specialising generic algorithms from the algorithm repository. In the current version of the EATT tool the choice of algorithms is constrained by the OOpt library to variety of heuristic search and genetic algorithms. In the case study this produces an application generator for a domain genre (allocation/reservation problems), with assumptions about the range of data types that can be addressed. In an alternative development a generator for a range of hiring/loans applications might be produced. At this stage the domain expert hands over the configured application generator to the end user. The boundary of role responsibilities in the process is flexible: however, the subsequent steps are driven by user interface wizards so the user does not require knowledge of algorithm scripts, component code or database schema.

The end user’s role commences with analysis for the specific range of applications intended from the generator, i.e. either a range of booking reservations/applications (hotel rooms, theatre, train tickets, etc.) or allocation applications (allocating contractors to jobs, tools to tasks, etc.). Domain analysis (step 5) captures terminology in natural language, any graphical/spatial details which are relevant, with the business rules and constraints to generate the application logic (e.g. in room booking, the size, cost, facilities, days available; for theatre bookings the performance ID is added; while for train tickets the time and route will be necessary). These specific attributes will be edited into generic placeholders created in step 3 during the database configuration. There is no specific support for data analysis activities where word processors and graphical drawing packages can be employed. Specialising the domain model (step 6) is supported by the constraint editor for specialising algorithms for business rules which work in conjunction with the domain properties mapper to create (condition, action) rules to deliver the desired range of reservation applications.

The user interface for the specific application is specified in step 7, which involves two sub-processes; step 8 adds the commands and input fields for the specific application (theatre performances, times, and costs) with location and view points of the seats displayed on a theatre layout diagram, so the end user can select performance dates and times with the desired seats. A form-filling/menu selection tool, add UI terms and commands follows (step 8), while graphical drawing tools (e.g. Photoshop) are used for constructing the graphical user interface (step 9), with an editor to designate which graphical objects are active for the application’s operational dialogues. Graphical objects from the Photoshop image...
file are designated as active selectable objects using the Windows GUI icon classes.

At this stage configuration of the EEAT generator is complete, so the user can execute the fully automated step 10 to generate the specific application. The support tools, query matcher and search/allocation controller, build the application UI and link the customised algorithms to user input commands and the back-end database. In step 11 the specific application is run by entering values into a form-filling user interface with menu lists to specify user requirements for (in a theatre booking application) a play, at a specific date/in date range, with seat reservations, etc. When the specification is complete is a matter of user judgement in light of the two phases of domain analysis. In the first generic configuration phase the initial domain model starts with a limited number of generic attributes and method templates for each object. More generic methods and attributes are added, and the seed ones specialised to cover the target range of applications. At this stage over-specification is advised. In the second customisation phase all, or more usually a sub-set, of the generic attributes and methods are specialised according to the requirements of the applications to be generated. The end user has to judge when the functional specification meets the requirements, e.g. is a seat reservation service part
of theatre booking, train ticket purchase, etc.? If generic objects for seats had not been added in the first configuration phase, the process iterates until the requirements for all the target applications are met.

In the following section the process and support tools are described in more detail with a scenario of use illustrating how the domain expert and end user configure then customise and generate an application.

### 3.2. Phase A: configuring the system architecture

The starting point for our approach was to develop a generic architecture which could be configured to generate wide-ranging applications within domains similar to ‘abstract’ product lines. The architectural concept is related to domain-oriented design environments (DODEs: Fischer, 1994; Fischer et al., 2004) which provide a software framework containing generic components for managing a reuse library, component retrieval, explanation facilities and critics to guide the end-user development. DODEs have to be configured with design examples and domain knowledge to power the advisors and critics (Fischer, 1994). However, they do not actually generate working applications, and user developers have to author code to add any functionality beyond the given reusable components. In contrast, EATT is intended to generate executable applications directly via algorithm selection from a reuse library. The generic architecture is illustrated in Fig. 2.

Three main components have to be configured to create a working application generator: first, the component retriever has to be provided with component descriptions for algorithms and other reusable components; secondly, meta-data for databases to be used in the application has to be supplied; and finally, models of the domains are necessary.

A wizard configuration editor guides the configuration process to populate the architecture with appropriate domain models, the database with meta-data, and the reuse component library with properties and other details. The design wizard consists of a dialogue management template which has to be populated with domain models and rules to map the user’s language to search parameters. Similarly, the application generator template has to be configured with appropriate domain and solution knowledge to enable composition rules to select, order and integrate search/matching algorithms, data retrieval processes and end-user interface components. Finally, there is a template for restricted natural language interpretation which has to be specialised for the target domains.

END-user application generation takes place in two phases:

1. **Domain configuration**: In this phase the end user acts as a domain expert intermediary with some software engineering background, who selects the domain models and then configures the architecture for the range of applications to be generated. This requires specialising the domain models with a domain sub-language.

2. **Application customisation and generation**: In this phase further detail is added by the end user then the application is generated resulting in an end-user query interface or requirements input dialogue.

The first phase of domain configuration is aimed at ‘intermediary users’ since although no programming or software engineering is involved, some computing knowledge (i.e. requirements analysis, specification and modelling) is advisable for domain configuration. Domain models are taken from the domain theory library (Sutcliffe and Maidan, 1998; Sutcliffe, 2002), which contains conceptual models that are sufficiently abstract to cover a large number of application variations, while being sufficiently concrete to be comprehensible to end users. The models are essentially sets of collaborating classes motivated by a high-level functional goal, such as handling hiring and rental transactions. The family of object hiring ‘object system models’ (OSMs) contains classes representing the resources, the requesting agent, with generic methods for handling loans, returns, rentals, reservations, etc. Other examples of OSM ‘domain families’ are object sensing for monitoring applications, object aggregation for compositional transactions, and agent control for command and control systems. In total 11 families with 2–3 sub-family specialisations are described. However, the domain theory models are paper-based descriptions so they have to be converted to an XML machine-processable format.

In the case study reported in this paper, the object allocation (OA) domain family was selected; this has the potential to generate any application with goals involving reservation, booking, matching or selection of resources. The OA domain consists of a set of resources, resource sub-classes, a requesting agent and an allocator agent with generic matching method; see Fig. 3.

Model specialisations within this family are for conceptual objects with sub-classes for financial information, resources, tasks, time-slots and goal-responsibilities; and physical object allocation, with lower-level sub-families for human agents, machines (non-human agents), physical and spatial resources. Models can be aggregated to describe problem domains; for example, allocating courses to rooms in training/educational domains involves conceptual resources being allocated to spatial physical resources (rooms), and if time scheduling is needed, temporal slot allocation can be added. OA applications have two operational modes: selecting resources for user requirements, and matching one type of resource to another (e.g. allocate people to tasks). All domain models are linked to generic requirements which describe design issues and goals relevant to the problem domain, in the allocation case consideration of the complexity of the resource and client requirements, conflicts in demands between stakeholders, optimisation, constraints relaxation, etc. The generic requirements indicate the complexity dimensions of the problem space and other factors, such as the possibility of relaxing constraints in matching and search which might influence the solution. Domain analysis is carried out to extend the OSM generic domain models to add further entities, methods and attributes which may be necessary for applications in the expect scope of generation. At this stage methods are added as placeholder stubs or outline scripts rather than detailed specifications.

Other operations in the configuration phase are selection of any necessary databases and the search/matching algorithms to
be used. The British Telecom iOpt algorithm component library was selected to supply the application logic and search/matching processes. This choice was driven by pragmatic, industrial considerations: British Telecom wished to enhance the reuse of its algorithm library. iOpt is an object-oriented framework for solving constraint-based satisfaction or optimisation problems with a configurable library of search algorithms ranging from simple hill climbing to heuristic search and genetic algorithms for evolutionary computing solutions. It supplies constraint-based problem solving and a variety of heuristic search algorithms, with built-in data types (e.g. integer, real, Boolean, string, object, and set data types), and operators (e.g. arithmetic, logical, set) to build a constraint satisfaction problem solver. The library of search components is used to instantiate heuristic search algorithms which start with a basic solution to a search problem and then perform iterative (breadth, depth) moves to improve the solution. iOpt models allocation requirements as constraints and provides heuristic search algorithms to search for feasible solutions which satisfy the constraints. Meta-heuristics (e.g. simulated annealing, taboo list) guide the algorithm in the search space towards optimal solutions. At this stage the configured generator is populated with Object Allocation conceptual models from the domain model library, databases of searchable resources and iOpt constraint-based search algorithms.

The domain model configuration dialogue is driven by the generator wizard which guides the user through the design process. After the architecture has been populated with the desired domain models from the domain theory OSM library, the generic properties for each model class are specialised with the domain model/properties editor. Then mappings between resources and related terms in the users’ requirements language are captured with the domain properties mapper so the iOpt algorithms can search resource databases. The constraint editor enables the user-designer to add constraints to prevent the end user from selecting invalid matches between resource classes, e.g. machine agents to training resources, and to specify domain-specific constraints. The results of configuration are written to XML files which are used in the generation process. The last configuration task is to add interpreters to handle ambiguous natural language terms in end-user queries. Rules are specified for spatial and temporal terms by interacting with timeline or layout diagrams to denote precise interpretation of ambiguous terms. Then the wizard prompts the user to generate the specific application by picking the mode of operation (requirements matching or inter-resource allocation), followed by the types of resource and their properties. If the requirements mode has been selected then an end-user query interface is generated and queries composed of property (attribute) values and Boolean operators can be entered to select resources. Alternatively, for the matching mode the end-user interface is generated to request input of constraints for the specific resource-matching problem. In both cases the iOpt search algorithm library is invoked to select the appropriate resources from the database and to find the optimal allocation.

The domain model selection/configuration dialogue is illustrated in Fig. 4. In this implementation the generator architecture has been configured with the OA family of domain models for matching, booking and resource allocation problems. First, the intermediary user selects the domain model sub-family; in the illustration, human agents and tasks-jobs have been selected for the matching mode of operation. The wizard supplies concrete examples of the selected domain models to guide the user.

3.3. Phase 2: customisation and application generation

This phase starts with conversion of the domain models into a machine-processable XML format, and analysis of the domain sub-language. The domain theory models provide a high-level set of abstract properties for classes, for instance the OA family lists properties for physical resource objects as: identifier, size, shape, mass, colour and type. However, these can be specialised with a specific domain lexicon to provide more comprehensible variable names.
for the end-user dialogue. A domain language analysis was carried out to collect a corpus of terms used in scheduling and matching applications within British Telecom and by searching the literature for resource allocation case studies. This created a domain lexicon for reservation, booking and matching applications. Some examples of the generic properties used to populate the design wizard are:

**Physical resource/human agent**: Age, height, weight, role, qualifications, skills, knowledge, generic attributes.

**Physical resource/machine agent**: Type, function, inputs, outputs, production capacity, mass, production rate, power, resource requirements, generic attributes.

**Conceptual resource/task**: Type, complexity, goal, inputs, outputs, pre- and post-conditions, state, average duration, generic attributes.

Generic attributes are placeholders which can be elaborated as new property types, while term instances and all synonyms can be added to property types during the design configuration phase of application generation. The components of the generator architecture are illustrated in Fig. 5.

### 3.3.1. Application customisation

Application customisation is intended to be carried out by true end users rather than intermediaries, since all operations are driven by graphical user interfaces.

Properties for the selected domain models are specialised for the range of applications anticipated in the domain. The end user may either refine the generic properties or new properties can be added, as illustrated in Fig. 6. This shows how an generic property in the OA **human agent** model ‘skill’ can be specialised with more specific skills which will be present in the databases that are expected to be used in application generation, e.g. engineer skills in customer service, line and server installation, etc. Properties can be refined into sub-classes and instances, e.g. for human agents, the specific types of skills expected in a range of domains can be given or qualifications refined into academic and vocational, etc. Each property is specified either as an enumerated set or value domains with data types and ranges.

Once the domain sub-language and models have been loaded into the generator, the next step is to supply matching rules to link the user’s requirements language to resource descriptions, or guide the mappings between two sets of resources in the matching mode of operation. Matching rules are specified either manually by the user indicating mappings on the displayed lists of properties, or automatically by the system searching resource attribute lists for exact or partial lexical matches. This is augmented by stemming...
algorithms (University of Lancaster, 2005) and WordNet synonym lists, although the user has to confirm partial matches. The matching algorithm is:

While requirements terms (input)
For each resource class
Search Resource properties for exact lexical match
Write exact match mapping rule
Search Resource properties for partial match
Apply Stemming algorithm
Assess match
If successful
Write mapping rule
Else
Search Wordnet for synonyms
Assess synonym match
Write mapping rule
Else
Report no match
End-if
End-if

The automatic process establishes mapping and then requests the user to supplement these with other requirement-resource property mappings that need domain knowledge to construct the rule; see Fig. 7.

The constraint specification dialogue has a similar structure to the matching rules and allows the user to specify compatibility constraints between resource and requirement properties which are stored in an XML constraints file for the application schema. First, the user specifies one or more property values in the source resource class, and selects property values to specify incompatible requirements for the target resource class.

3.3.2. User interface customisation

The generated end-user interface for the allocation application is a simple query interface composed of menu lists of properties with form-filling dialogues to complete the queries with value constraints and limited Boolean operators. However, the application may need further customising to fit end users’ needs. For example, precise lexical matching requires users to be precise in their specification of queries and matching constraints, which does not always come naturally. The application customising phase therefore includes a facility to generate interpreters for ambiguous and fuzzy queries in natural language rather than enforcing input of precise values and use of Boolean operators. Of course there is no escape from the need to be precise, so the user-designer has to configure natural language interpreters with semantic definitions of ambiguous natural language terms.

Application customisation also supports specialisation for expression of temporal and spatial requirements. Allocation problems often involve spatial resources and temporal constraints, so interpreter configuration modules have been developed for a range of terms such as ‘nearby’, ‘adjacent’, ‘close to’ for spatial allocation, and ‘before’, ‘after’, ‘immediately after’, etc., for temporal constraints. The configuration facilities allow end users to add as many ambiguous terms as they wish and specify the precise interpretation by graphical editors and timeline specification dialogue, as illustrated in Fig. 8. The spatial interpreters are more restricted since they rely on a domain-specific topography, e.g. layout of seats in an aircraft, theatre, etc. These also support end-user customisation since configuration of the interpreter is driven by user interfaces of familiar graphical drawing tools with annotations on diagrams of resources for a particular domain. Once an interpreter has been implemented for set of spatially organised resources (e.g. seating plans, room layouts), it can be rapidly reconfigured for different application versions.

Temporal ranges are selected for the sliders (minutes-hour, hours-day, days-week, etc.), then the natural language term is entered on the left-hand side of the screen and the sliders are manipulated to specify the precise temporal interpretation. A simple lexicon allows the system to infer definitions from the slider position (e.g. ‘before’, ‘after’, ‘map’ to left or right of the slider.
Fig. 7. Matching rules dialogue in the constraint editor tool.

Fig. 8. Temporal interpreter UI customisation dialogue, in which the user enters an ambiguous temporal term and then specifies its interpretation by moving sliders to denote time ranges.

Please cite this article in press as: Sutcliffe, A., Papamargaritis, G., End-user development by application-domain configuration. J. Syst. Software (2014). http://dx.doi.org/10.1016/j.jss.2013.11.1121
position); durations (e.g. ‘mid-morning’) expect two sliders to be set, while some terms such as ‘early’ may be defined as points with one slider or as a duration with two. Two sliders can be selected to express concurrency, ‘at the same time as’, although the current implementation has limited ability to infer more complex temporal semantics such as ‘start within a short while after’.

To initialise the spatial interpreters the designer/user has to describe the topography of the domain with standard graphical drawing tools (e.g. PowerPoint, Photoshop). A dialogue then guides the user to specify the semantics of ambiguous natural language terms, such as ‘nearby’, by pointing and annotating the domain sketch; see Fig. 9. The system’s object management system infers a spatial rule from the annotations, for example, objects falling either side of a boundary on the diagram will be interpreted as ‘nearby’ or ‘not nearby’.

The user enters the term then selects objects on the diagram to define a precise meaning. As with the temporal interpreters the system users a lexicon to control the dialogue and inference process. Spatial definitions can be either specified by user action alone with multiple points, e.g. for ‘nearby’ the system takes the first selection as the target and all subsequent selections as proximal objects. Alternatively the user can label objects as composites, as shown in Fig. 9, where blocks and rows have been specified for seat reservation applications. ‘Nearby’ is then defined by selecting seats within that block. Similarly, ‘not too close to’, is inferred from gaps between the target and distal objects in a structured topology. In layouts without a structured topography the system provides a ‘bullseye’ metaphor so proximity can be defined by manipulating the bullseye rings which are linked to customisable distance scales (mm, cm, metres). The system infers the meaning of ‘nearby’ or ‘away from’ as the position of objects falling within or outside the ring measured from the target object. Other interpreters can be configured in a similar manner by the system making inferences from lines drawn on topographic sketches, e.g. directions and connections between objects are inferred from the coordinates of lines and object locations.

4. Implementation

The generated system architecture consists of the query/requirements capture interface (see Fig. 10) which employs form-filling dialogues to capture either the user’s query for resources or constraints when matching two sets of entities. The end-user interface also calls the customised natural language interpreters as required by the query dialogue. The user’s query is then sent to the Allocation search controller which calls the appropriate iOpt APIs, and supplies parameters to search the appropriate resource databases. The search results are returned to the user interface to display lists of resources, partially matched resources or error messages if no matches were found.

In this example the user has been prompted to select their target location and has enquired about the availability of free places close to the target. The system will highlight all contiguous available places; however, if the user has requested ‘nearby’ following the definition shown in Fig. 9, only the places to the left and right would be highlighted, since the system inferred nearby to be left, right and in front of the target within the same block.

The generated application architecture (see Fig. 11) consists of three packages: user interface classes which implement the UI query/matching and results display; spatial and temporal interpreters called by UI classes to handle natural language input; and the Search controller class which calls and controls iOpt library components to execute database searches.

EATT was implemented in Java using JavaSwing UI components. The lexical matching algorithms use the Java API to access the WordNet dictionary (Java WordNet Library, 2013), while the lexical matching process uses the Paice/Husk stemming algorithm (University of Lancaster, 2005). The spatial editor for building
spatial models and spatial interpreters is based on the JHotDraw tool (JHotDraw, 2013) which generates two-dimensional graphics. The Arch4 library (Arch4, 2004) open-source library generated the source code for Java classes, while JOpt supplied constraint-based problem solving and a variety of heuristic search algorithms. JOpt provides built-in data types (e.g., integer, real, Boolean, string, object, and set data types), and operators (e.g., arithmetic, logical, and set) to build a constraint satisfaction problem solver. A library of search components is used to instantiate heuristic search algorithms which start with a basic solution to a search problem and then perform iterative (breadth and depth) moves to improve the solution. Evaluation heuristics assess the efficiency of the solution in comparison to the objective function of the input problem. The heuristic search template binds generative (constructive) heuristics, neighbourhood search, neighbourhood move functions, evaluation functions, etc. In the same way, genetic algorithm templates assemble generative heuristics with crossover, mutation, and selection operators. The complexity of the input problem can be classified by the complexity of integrity constraints, and data dependencies expressed in the database and in the user's requirements. Complexity metrics facilitate semi-automatic selection of the appropriate heuristic search or evolutionary computing processes according to the complexity of the domain. If many data dependencies restrict generate-and-test style search then heuristic search is preferred; alternatively, genetic algorithms or simulated annealing can be used to generate new potential solutions. In the resource allocation case study, heuristic search templates were used as most resource allocation problems implied a high level of data dependencies and integrity constraints.

5. Evaluation

The first versions of EATT were evaluated in the domain configuration phase with intermediary users (British Telecom software engineers), then examples of the generated applications were tested with end users. Initial tests demonstrated that the application generation and customisation could be successfully completed once the system library had been loaded with the appropriate domain models and algorithms. End users found the generated query requirement interfaces easy to use with few usability problems.

Two iterations of further usability evaluation were carried out on an improved prototype with six software engineers (who performed the intermediary role) and seven novice end users drawn from British Telecom and University of Manchester computer science undergraduates respectively. The six British Telecom software engineers (domain expert/intermediary role) all had several years (>5) of programming experience in Java and other languages; the novice users were first-year students in the School of Informatics, University of Manchester, who had limited programming experience as they had not completed the initial course in Java. Both groups were introduced to the EAT system in a 30-min training session when the purpose and operation of all features were explained, with hands-on demonstration of editor-customisation.
tools. Participants were encouraged to ask questions to clarify any concepts or operations they did not understand.

In these evaluations both user groups were asked to test the whole system, so this represented a more severe stress test for the configuration/generation part of the system with novice end users. Both groups were asked to configure the generic architecture for resource allocation domains; to generate end-user applications for two scenarios, room booking and service engineering call allocation; and then to operate the generated applications to reserve a specific room for a class within a time period (query mode), and allocate a set of engineers to a database of outstanding jobs (matching mode). The test system was configured with databases and the iOpt library so these aspects of the first configuration phases were not tested; however, all users did have to select the appropriate domain model from the library. A lexicon of domain properties was provided to enable the domain customisation phase to begin. Both novice and expert users completed both tasks in reasonable time (range from 22 to 45 min) although the novices required more help from the researcher when problems were encountered. The problems encountered during the evaluation are shown in Table 1.

Usage problems encountered by novice and expert users are categorised into function areas: specialisation of properties, mapping rules, and user interaction with spatial interpreters, as well as usability problems with operating the generated query user interface.

Usability problems were defined as critical incidents when the users needed help in operating the user interface and experienced difficulties in understanding the system so they had to ask the evaluator to explain the meaning of terms. Not surprisingly, novices encountered more usability problems than did intermediary users, although the distribution was similar with specialisation of domain properties accounting for most problems (48%). For both sets of users, terminology comprehension problems were most frequent (60% of total usability problems) and the majority of these were concentrated in the specialised properties task; in contrast, the remaining 40% of usability problems were spread more evenly with a concentration in the operation of the end-user query interface. One reason for the high number of problems in the query interface of the generated application was the provision of both form-fill/natural language queries and the option of using Boolean operators. This made the user interface too complex and confusing for novice users. When the complexity was reduced by removing Boolean operators option and simplifying the natural language queries as sets of selections from menu lists, the error rate was reduced to low levels. Other parts of the configuration interface (mapping rules and configuration interpreters) performed well with few usability problems. Evaluations of the configuration facilities of the generator demonstrated that intermediaries and novice end users could use the system and successfully customise the generator architecture for resource allocation problems, and subsequently generate room-booking and service engineer-job allocation applications.

A usability survey captured the attitudes towards the EATT system of both novice and intermediary users after they had completed the configuration, generation and use of the test applications. The post-test questionnaires, indicated a positive view of EATT with the mean rating of the configuration ad customisation interfaces for novices and intermediaries 4.5 and 4.8 (on a 1–7 scale where 7 = very easy to use). The most commonly reported problems in open-ended questions were comprehension of domain terms in property selection and specialisation, creation of lexical matching rules and the formulation of matching queries. Users encountered difficulties when specialising generic properties (e.g. task, goals and human roles) to more specialised concepts (e.g. job categories). Terminology such as sub-classes, measured property values and enumeration types were too abstract and need to be replaced by simpler expressions. When creating matching rules some irrelevant properties were chosen and users failed to specialise temporal and spatial expressions. The few comprehension difficulties with natural language expressions in the temporal and spatial components were resolved when the configuration diagrams were used to explain how ambiguous terms were being defined.

In the customisation phase 2, novice users experienced difficulties when selecting properties associated to enumeration classes (e.g. skills). The graphical user interface controls for moving property values on the search panels was considered too complex. The number of usability problems declined by over 50% (novice total 22 and intermediary total 9) in the second round of testing after design improvements had been made to the configuration/customisation wizard dialogues and end-user interface. Most of the residual problems concerned comprehension of terms in the configuration dialogue which could be alleviated with improved training. All the residual usability problems could be solved with modest effort and addition of explanation facilities to guide end users. In general, the wizard-based dialogue successfully assisted the users in both phases with step-by-step guidance, demonstrating that the generator was suitable for end users who were novices, as well as intermediary users with some computing expertise.

6. Assessment of reusability

The usability evaluation demonstrated that EATT showed some promise as an end-user application development tool; however, the development assumed the presence of specifications for allocation domain problems and the iOpt search algorithms library to provide the generated solution. To investigate wider-ranging application of the EATT architecture, a paper-based validation study was undertaken to assess the redesign of EATT necessary for a new domain. This involved the paper’s authors and two colleagues at the University of Manchester who were experienced software engineers. Hiring, loans and rental applications were selected since these are all specified as another domain theory OSM family (Sutcliffe, 2002). Specific application scenarios of library loans, bicycle rental and fancy dress hire were used to test the customisation process. Table 2 summarises the main changes for the new domain. The configuration wizard enables a new set of generic models to be loaded; however, the object system models and their associated generic properties have to be coded as XML files. The design wizard then controls the specialisation process as in the resource allocation domain. The domain properties editor, rule mappers, and interpreter configuration can be reused as they stand. The iOpt algorithm library can also be reused to search for objects for hire in response to users’ requirements. Some changes are necessary so the generator can create the user interface dialogue for loans, hiring operations (request loan, reserve, loan, return, etc.) with new methods for database updating when loaned resources are returned. The effort to configure EATT for the hiring/loans domain was estimated to be approximately five days, consisting of a domain analysis for the hiring/loans lexicon, and modification of four methods in the generator and database update classes.
To test more wide-ranging reuse, a second generic domain of monitoring applications was selected (object sensing domain model family). The domain properties editors and interpreters could be reused without change, although a more sophisticated temporal interpreter may be required. Mapping rule classes were not directly applicable, although they might be reused in event interpreters to map signals to higher-level events. The specific example used to test customisation was monitoring patients (heart rate, EEG, etc.) in long-term healthcare. Configuration of EATT to this domain necessitated more extensive changes to the generator to implement methods for detecting and interpreting patterns of events and then displaying alerts for unusual events as well as normal patterns. The iOpt solution library was no longer appropriate so a new reuse library would have to be found, e.g. digital signal processing and data-mining algorithms. Estimating the effort required for this configuration was not possible without further specification; however, it was at least two or three times the resource for the hiring control domain.

The reuse testing demonstrated that the EATT architecture could be easily transferred to domains with similar transaction properties and requirements for searching and matching, for example sales order processing, financial transactions, as well as loans and hiring. However, for domains not related to information system–style applications, more effort was required to configure the generator, although the basic architectural template and customisation wizards were still reusable.

7. Discussion

The development of the EATT tool has demonstrated the feasibility and proof of concept of a two-phase approach for end-user development, and extended the DODE approach with a novel means of assisting ‘seeding’ design environments with high-level conceptual models. Although we did not add critics and explanation components of DODEs (Fischer, 2009), the domain models act as design patterns; furthermore, the generic requirements and issues associated with the models provide reusable knowledge that could be implemented as knowledge-based assistant tools. The two-phase approach may empower end-user development with simple user interfaces, coupled with powerful configuration tools, enabling end users to generate, tailor and use a wide range of related applications. It combines a natural language approach for components search with specification-based user demonstrations and manipulations in a graphical environment, following the interaction style of AgentSheets (Ioannidou et al., 2009). We extended the graphical EUI approach into business-oriented applications such as booking reservation systems. Usability testing demonstrated the effectiveness of the approach, although the range of specific applications tested was limited, and large-scale system testing was not carried out. Nevertheless, the EATT tool proved the viability of end-user application generation/customising tools based on models of problem domains. XML-based customisation for generating financial applications (Emmerich et al., 2001) is similar to our approach, although the range of applications which could be generated by TIGRA was more limited.

The user interface contribution of EATT extends work on UIDE (user interface development environments) which have used task models to specify the application structure and logic, combined with design patterns to semi-automatically generate the user interface (Ahmed and Ashraf, 2007). The application of search allocation tasks in a hotel room reservations system was demonstrated in Ahmed and Ashraf; we have extended this with more complex spatial and temporal configuration support for such applications using user-centric language, e.g. ‘nearby’, and graphical tools to develop reservation/allocation interfaces. A similar task-based approach for end user UI development has been applied to service composition (Paternò et al., 2011), with search-browse tasks in media resource selection applications. In this case the UI and application were generated semi-automatically from service components with embedded UI design knowledge in the tool; however, only simple list-based search was supported, rather than the richer graphical world interaction provided in our system.

Unlike many application generators (Freeman, 1978; Smith, 1990; Markus and Fromherz, 1994; Batory et al., 2000), no knowledge of a formal specification language is necessary in our two-phase approach. The model-driven dialogue allows users to customise the generators to a range of applications, resembling product-line configuration of variation points. The approach is related to model-driven architecture (Mellor and Balcer, 2002), but extends the concept with ‘problem domain MDA’ by using models of application requirements which are closer to the users’ perspective. Models from the domain theory (Sutcliffe, 2002) provide a bridge which proved to be sufficiently abstract to enable customisation of a wide range of applications, while being comprehensible to end users. We argue that these domain abstractions are closer to the end-user perspective than GenVoca (Batory and Geraci, 1997; Batory et al., 2000) generators which are focused on technical specialists in database applications. However, even end-user oriented domain models are not easy to understand as most of the user problems in testing the EATT tool involved misunderstanding of terminology.

Our work builds on the DODEs approach (Fischer, 1994, 1998; Fischer et al., 2009), in that both are component-reuse based, end users interact via simple dialogues, and both require seeding with models, components and knowledge pertaining to a domain. EATT provides a seeding assistant tool for DODEs since configuration requires domain models, solution algorithms and a domain lexicon. DODEs provide a generic architecture for end user development to build design tools, critics and explanation facilities, but the architecture needs more specific, design knowledge to power explanation and critic facilities. EATT and the domain theory models provide such knowledge, while the combination of natural language and graphical demonstration also enables end-user configuration of user interfaces to specify rules generally applicable to temporal and spatial problems. This aspect of EATT is similar to AgentSheets (Repenning and Ioannidou, 2004), although we have not implemented an explicit rule-based system; instead, rules are inferred by graphical annotations on diagrams, adopting a programming-by-example approach (Lieberman et al., 2004); however, more complex ‘drag and drop’ style manipulations would require new interpreters to be implemented.

EATT extends previous two-phase approaches to end-user development by providing a library of generic conceptual models as a starting point and a range of tools to support the process of development via progressive refinement of these models and integration with appropriate algorithms to automatically generate solutions via component based reuse. Fogli and Parasiliti Provenza (2012) adopt a similar two-phase approach to EUD for e-government.
services, although their development support environment is tailored more directly to services for data entry administration and reservation/search functions in e-government with a six-step task model, so it is not clear how their tools could be adapted for other application domains. Another phased but domain-centric approach proposed by Drey and Consel (2012) supports EUD for sensors and device control in home-based services. However, their approach in the Pantraguel programming language provides a more general meta-model of sensor-controlled actuator which can be adapted to a wide variety of monitor-control oriented applications. The Pantraguel meta-model is similar to generic problem models in the domain theory (object sensing and agent control) (Sutcliffe, 2002, 2014) which can be used in EATT. In contrast, Stav et al. (2013) use three phases/roles from domain developer to service composer (app level), then service end user with support for specifying and implementing domain-specific languages to drive a menu/form filling EUD tool set. Although they illustrate application of their approach in the development of mobile apps for navigation and location-based services, no evaluation with end users was reported; also, transfer to another domain appears to require development of a new domain-specific toolkit.

Application generators with natural language front ends (Mellor and Balcer, 2002; Smith et al., 2000; Pane and Myers, 2006) are similar to EATT, but they require a more comprehensive configuration with a restricted natural language ontology and grammar, to customise parsers and the generator. In contrast, EATT provides an initial domain ontology and requires less customisation via a limited domain analysis, and simple extension of the domain lexicon. The mapping rules component avoids the complexity of natural language parsing. Although EATT is limited to information system transactional domains, we believe the model-driven generator approach may involve less configuration than building new natural language parsers. Another contribution of EATT is support for specify rules by graphical metaphors, e.g. timeline sliders, rather than ambiguous natural language terms. Although use of diagrams to express specification concepts is well known in visual programming languages, we believe the diagram-based temporal and spatial interpreters provide a novel extension to natural language interfaces. Diagram-based interfaces capture user interactions to infer computational semantics, such as relationships between components in web applications (Newman et al., 2003).

Testing inter-domain transformation of EATT showed that core customisation wizards and generation components can be reused across many different domains, while most of the architecture transferred easily within information system transaction-style domains. Our future research will test the prototype generator introduced in this paper for scaling with larger databases, and applications with detailed requirements, e.g. flight booking and seat reservation with constraint relaxation so alternatives can be suggested for user choices which cannot be initially satisfied. Another possible improvement is using natural language similarity matching algorithms such as latent semantic indexing (LSI) to automate the mapping rules between the user's requirements language and database terms; however, these algorithms work more effectively with phrases and longer texts than with single keywords. Also, the interpreter package will be extended to add more natural language terms and more complex expression of queries and requirements; in this case, LSI might be more productive.

The EATT approach could be incorporated within Software Shaping Workshops — VIS (visual interactive systems end user development) approach (Costabile et al., 2007) which proposes a series of workshops facilitated by HCI (human computer interaction) and Software Engineering experts, who help end users specify computational semantics and processes by discussion of images, diagrams and processes. In Costabile et al.'s example of an MRI scan analysis application, computational objects are annotated on images and associated with operations denoted by user-crafted icons. EATT users created diagrams and images in a similar manner, but combines this customising with component-based reuse. We agree that a participatory design approach is necessary, which we tested in iterations of systems evaluation to improve terminology and tool usability. Participatory design involving both end users and intermediaries could also help to mitigate terminology problems we discovered during the evaluation.

8. Conclusions

In conclusion, we have described a proof of concept end-user application generator-customising tool, validated its use with end users, and demonstrated the reusability of this ‘problem model’-driven architecture approach. The contribution of the EATT application generator has been to extend previous two-phase EUD approaches with configuration based on a library of generic conceptual models that provide a sound starting point to guide further specialisation. Previous approaches have proposed modelling languages or conceptual schema; however, modelling has been left to the user developer with limited guidance. Our two-phase approach still relies on an intermediary domain expert role which assumes some software engineering knowledge to interpret computing terminology and select appropriate algorithms; however, domain experts with such computing/development knowledge are not unusual in some end-user domains, such as medical informatics (Driedger et al., 2007; Thew et al., 2009). Furthermore, we contend that the intermediary role could be achieved by end users with reasonable training, i.e., introduction to concepts.

There are several limitations to our work so far, which has only demonstrated the feasibility of the two-phase approach to EUD tools. EEAT can be configured to generate applications in a wide range of domains but it is dependent on the range of the models in the domain theory library (Sutcliffe, 2002). The applications generated depend on choice of algorithm library and user interface toolkit which limited generation to Java environments using the British Telecom iOpt library. Further work is necessary to reuse other algorithm libraries. The tools are prototypes which would need development to address the usability problems identified during the evaluation, as well as further testing for robustness, so further development would depend on the skills of the domain expert intermediary. The evaluation of EEAT was conducted with a limited number of users and tasks; furthermore, we did not test the effectiveness of the generated applications, although informal evaluations did demonstrate their usability. Further evaluations will be carried out on the generated tools and applications with a range of end users with skills varying from no computer knowledge, domain experts with limited computer knowledge and more skilled software developers. Only one application area, allocation/reservation, was tested, and the range of applications problems was relatively simple, so in our future work we will stress-test generation with more complex constraint-based matching in other reservation problems. Assessment of the generalisability of the architecture was limited to a single paper-based exercise for configuration to monitoring applications, so while this provided insight into the modest extent of additional work to transfer EEAT to another problem class, realistic software customising needs to be undertaken. In spite of these limitations, the work we have reported suggests that our approach merits further investigation. However, the ultimate test will involve examining the trade-off between the effort expended in our two-phase approach which addresses a wide range of application areas, versus the configuration effort and ease of use pay-off in more constrained DODEnvironments (Fischer et al., 2009). Future work could also
investigate the economics of different architectural approaches (Sutcliffe, 2008, 2014) to compare the usability and utility of more domain specific EUD tools (e.g. Fogli et al., 2012) with generic high-level specification languages (Repenning and Ioannidou, 2006; Pane & Myers, 2006) for EUD development.

Acknowledgement

The authors wish to thank British Telecom for use of the iOpt library and support for this research.

References


Archpath, A. Available online at http://archpath.sourceforge.net/ (accessed 06.02.13).


Alistair Sutcliffe retired in October 2011 from the University of Manchester, where he is now Emeritus professor. However, he continues his research as a visiting professor at University College London and the University of Lancaster. He has been a principle investigator on numerous EPSRC and European Union projects. His research interests in HCI are theory and models of user experience, interaction design, socio-technical media, and design of complex socio-technical systems. In software engineering he specialises in requirements engineering methods and tools, scenario-based design, and theories of domain knowledge. He has over 250 publications including five books and was awarded the IFIP Silver Core in 2000.

George Papamargaritis gained his PhD on domain modelling for software architecture development from the University of Manchester Institute of Science and Technology in 2007. He worked as a software engineer for British Telecom in the UK from 2000 to 2006, when he moved to Greece to take up his current position as softwarearchitect at Intracom IT Services in Athens.

Please cite this article in press as: Sutcliffe, A., Papamargaritis, G., End-user development by application-domain configuration. J. Syst. Software (2014). http://dx.doi.org/10.1016/j.jss.2013.11.1121