A Component Abstraction for Localized, Composable, Machine Manipulable Enterprise Specification

Vinay Kulkarni[1], Tony Clark[2] and Balbir Barn[2]
[1]*Tata Research Development and Design Centre, Tata Consultancy Services, 54B Hadapsar Industrial Estate, Pune, India*
[2]*Middlesex University, London, U.K.*
vinay.vkulkarni@tcs.com, {t.n.clark, b.barn}@mdx.ac.uk

Keywords: Enterprise Modelling, Component.

Abstract: Enterprise modelling aims to specify an enterprise in terms of high-level models that address key problems such as business-IT alignment, enterprise transformation and optimal operation. No two situations in real world enterprises are exactly alike but there may be significant overlap. Relative ignorance of such overlaps forces essentially the same problem, albeit in a different context, to be repeatedly solved from scratch. This is a time-, effort- and cost-intensive endeavour. To overcome this problem and facilitate reuse, we propose a model-centric component abstraction that enables specification of the *what*, the *how* and the *why* concerns of enterprise in a localized, composable and machine manipulable manner. We present a meta-model, describe concrete syntax for its textual representation, and discuss the required model processing machinery.

1 INTRODUCTION

Modern enterprises operate in a highly dynamic environment wherein changes due to a variety of external change drivers require rapid responses within a highly constrained setting. This calls for precise understanding of: *what* is the enterprise, *how* it operates and *why* it so operates, the set of change drivers, a set of possible to-be states, and a quantitative and qualitative to-be state evaluation criteria. Understanding is typically required at several levels of granularity: a department, a business unit, the entire enterprise *etc*. The scale of a modern enterprise means this understanding exists only for highly localized parts and typically in the form of documents or descriptive models (Zachmann 1999, TOGAF). Popular enterprise modelling tools, *e.g.*, ArchiMate (http://www.visual-paradigm.com) offer little support for the quantitative or qualitative analysis of enterprise models. As a result, experts are forced to rely solely on their experience when faced with a specific problem. Thus, fractured incomplete knowledge and sole reliance on human expertise emerge as the principal contributing factors leading to change responses that are inaccurate, inefficient and ineffective.

Key decision-makers in enterprises face generic problems: business-IT alignment, optimizing cost of IT to business, transformation with certainty *etc*. These problems manifest differently in different contexts and yet share significant commonality. For instance, the details of wealth management bank merger differs on a case by case basis; however cases have much in common both in terms of problem formulation as well as solution. Current Enterprise Modelling state-of-the-art as well as of-practice completely ignores this commonality. As a result, each problem instance needs to be solved afresh. This is a highly cost-, time-, and effort-intensive endeavour.

The problems we seek to solve are: (i) to reduce excessive dependence on human experts for decision making, (ii) to address the commonality across different instances of a generic problem and, (iii) to address scale and complexity.

Our proposition addresses the problems as follows: (i) the *what*, *why* and *how* perspectives of an enterprise are captured in terms of a core conceptual meta-model, (ii) DSLs and patterns are used to capture reusable knowledge and translated in terms of the core concepts into a kernel language, (iii) the kernel language supports features to address scale and complexity including composition, encapsulation, and higher-order features.

Our approach is extensible through the use of a plug-in architecture supporting DSL-based models translated to an executable kernel language defined

Kulkarni V., Clark T. and Barn B.
A Component Abstraction for Localized, Composable, Machine Manipulable Enterprise Specification.
DOI: 10.5220/0005425801800185
In *Proceedings of the Fourth International Symposium on Business Modeling and Software Design (BMSD 2014)*, pages 180-185
ISBN: 978-989-758-032-1

in terms of a core collection of concepts. Section 2 discusses related work. Section 3 provides an overview of the different aspects of the approach and section 4 concludes by describing our progress to date and our future directions.

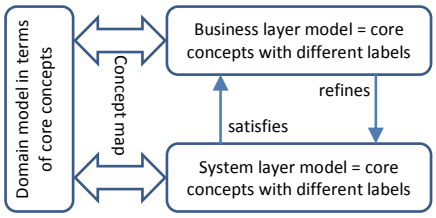

Figure 1: Purposive meta-models.

2 RELATED WORK

The current state-of-the-art of enterprise modelling (or specification) can be broadly classified into two: those that focus on *what* and *how* aspects (Clark et al., 2013, Zachmann 1999, TOGAF, Wisnosky and Vogel, 2002) and those that focus on *why* (Yu et al., 2006, Dardenne et al., 1993, OMG BMM 2010). The supporting infrastructure for the former, with the exception of the ArchiMate tool (bit.ly/1s1WyTv) is best seen as a means to create high level descriptions for human experts to interpret in the light of synthesis of their past experience. The Stock-n-Flow model (Meadows 2008) provides a different paradigm for modelling *what* and *how* aspects and comes with simulation machinery for quantitative analysis (http://bit.ly/1hebMvC). Several BPMN tools providing simulation capability exist but are limited to the *how* aspect (http://bit.ly/PdkqVg). Supporting infrastructure for *why* (http://bit.ly/Oav0Lm) is comparatively more advanced in terms of automated support for analysis. Informed decision making demands taking into account all the three aspects in an integrated manner, however, correlating *what* and *how* with *why* remains a challenge. Given the wide variance in paradigms as well the supporting infrastructure, the only recourse available is the use of a method to string together the relevant set of tools with the objective of answering the questions listed earlier. The non-interoperable nature of these tools further exacerbates automated realization of the method in practice. As a result, enterprises continue to struggle in satisfactorily dealing with critical concerns such as business-IT alignment, IT systems rationalization, and enterprise transformation.

3 PROPOSED SOLUTION

We propose a modelling language engineering solution based on the principles of separation of concerns (Tar et al., 1999) and purposive meta-modelling. We posit a core language defined in terms of generic concepts such as *event*, *property*, *interface*, *component*, *composition,* and *goal*. They constitute a minimal set of concepts necessary and sufficient for enterprise specification. The core language can be seen as a meta-model template where the generic concepts are placeholders. In the proposed approach a template emits the desired purposive meta-model through a process of instantiation wherein the placeholder generic concepts are replaced by purpose-specific concepts. This makes it possible to establish relationships across multiple purposive meta-models as shown in Fig. 1 and also impart consistent semantics.

The meta-modelling approach is suited to the open-ended problem space of enterprise modelling: any number of meta-models can be defined, relationships spanning across the various meta-models specified and the desired semantic meaning imparted *etc*.

3.1 Component Abstraction

A component is a self-contained functional unit with high coherence and low external coupling. A component exposes an interface stating the externally observable goals, expectations from the environment, mechanisms to interact with the environment, and encapsulates an implementation that describes how the exposed goals are met. A component can make use of several contained components in order to meet the promised goals. A component participates in hierarchical composition structure to accomplish wider goals of the enterprise, *e.g.,* larger unit or an enterprise. The expectations of a component from its environment are accompanied by a quality of service guarantee and together both constitute a negotiating lever. Thus, a component is in fact a family of (member) components where all family members have the same goal and interaction specifications but differ only in terms of the quality of service delivered and the expectations from the environment for delivering the promised quality. The core concepts of a component abstraction are depicted in Fig. 2.

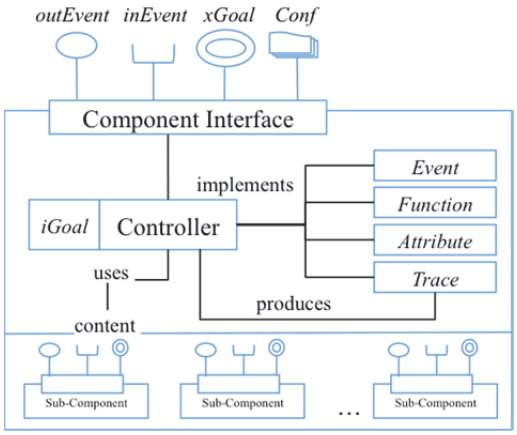

Figure 2: Component abstraction.

We draw from a set of existing concepts to derive the component abstraction. *Modularization*, *reflective component hierarchies* and *interface-implementation separation* are taken from Fractal Component Models (Barros et al., 2009). *Goal-directed active behaviour traces* are taken from Agent Behaviour (Bonabeau, 2002). Defining *component state* in terms of *attributes* and *traces* is borrowed respectively from object oriented design patterns (Gamma et al., 1995) and *event driven* enterprise architecture modeling (Clark and Barn, 2011). The event driven architecture (Michelson 2006) is a means to support flexible interactions protocol between components. Finally the concept of *intentional modelling* (Yu et al., 2006) is adopted to enable specification of component goals.

3.2 Component Meta-model

The proposed component meta-model is depicted in Fig 3. A component has two parts – an interface and implementation: the interface addresses the *what* and *why* aspects, the implementation addresses the *how* aspect. Thus a component (C) is a tuple <*CI*, *Impl*>. A Component Interface(*CI*) is a tuple <*inEvent*, *outEvent*, *xGoal*, *Conf*> where *inEvent* is a set of events of interest to the component, *outEvent* is a set of events generated by the component, *xGoal* is the external observable goal of the component and *Conf* is a set of configuration variants that conform to the *InEvent*, *OutEvent* and *xGoal*. *Conf* is a tuple <*Expect*,*QoS*> where *Expect* is the set of expectations from the environment expressed as name-value pairs, and *QoS* is the set of *QoS* properties to be guaranteed by the component provided the expectations are met. Implementation (*Impl*) is a tuple <*iGoal*,*P*,*F*,*T*,*Content*,*iEvent*> where: *iGoal* is the internal goal of the component, *P* is a set of properties or attributes, *F* is set of functions each encoding a computation, *T* is a trace of past consumed and produced events, *Content* is a set of its sub-components, and *iEvent* is set of internal events used to interact with sub-components.

Component implementation is necessary and sufficient to cater to all its variants as specified in *Conf*. The control unit *CU* represents implementation of a component. It responds to *inEvent* set of events, raises *outEvent* set of events, and orchestrates the *Content* sub-components so as to accomplish the stated *iGoal*. It records the events of interest and changes to component properties (*P*). The control unit captures the behaviour of component *i.e.*, a set of handlers for all *inEvent*.

Event (*E*) is defined as tuple <*Name*,*EP*, *preCond*,*postCond*> where *Name* is the identifying label, *EP* is the set of properties or attributes of the event, *preCond* is the condition that must be fulfilled to recognize the event, and *postCond* is the condition that must hold true after completion of the event. The *preCond* and *postCond* are expressions over events and event properties.

Goal is a tuple <*Name*, *GExpr*> where *Name* is the identifying label and *GExpr* is either property expression (*PExpr*) or event expression (*EExpr*) or goal composition expression (*GCExpr*). Property expression is value expression over properties (*P* and *EP*), event expression is an LTL formula over events, and goal composition expression uses a set of composition operators over goal expressions. The goal composition expression enables specification of limited uncertainty and non-determinism in the goal.

3.3 Language Features

The approach outlined above relies on being able to represent and process an organisation that is expressed in terms of a component-based abstraction. We envisage a product-line approach (Reinhartz-Berger, 2013) whereby a suite of tools based on this abstraction is used to facilitate a collection of different organisation analysis and simulation activities. Each activity will constitute a *domain*, *e.g.*, cost analysis, resource analysis, mergers and acquisition, regulatory compliance. In principle, each new domain will require a new domain specific language to represent the concepts. How should such a proliferation of domains be accommodated by a single component abstraction?
Our proposal is to construct an extensible kernel language that is used as the target of translations from a range of domain specific languages (DSLs).

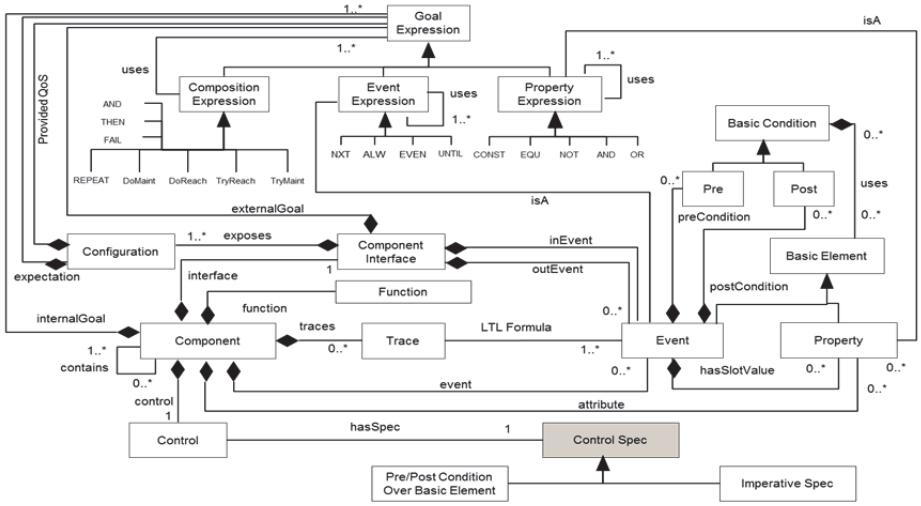

Figure 3: Component meta-model.

Each DSL supports an organization analysis and simulation use-case. We then aim to construct a virtual machine for the kernel language so that it is executable. Model execution supports organisation simulation and some analysis use-cases. Links to external packages such as model-checkers will complete the analysis use-cases.

The use of a single kernel language provides a focus of development effort and can help minimise the problem of point-to-point integration of analysis methods. Our proposal is that the concepts defined by the model in Fig. 3 are a suitable basis for most types of analysis and simulation use-case and therefore the kernel language will be defined in terms of these concepts.

Given its ability to accommodate multiple simulation and analysis use-cases, we envisage the language being the basis of a suite of organisational modelling, simulation and analysis tools, presented in the form of a single integrated extensible meta-tool. Since organisational information is likely to be very large (at least many tens of thousands of model elements) it is important the tool is efficient, scalable, supports distributed development and is flexible in terms of its architecture. To this end we aim that the language should be compiled to a machine language running on a dedicated VM, the language integrates with standard repository technology, and can run equally well on single machines, networked machines and via the cloud.

Organisations consist of many autonomous components that are organized into dynamically changing hierarchical groups, operate concurrently, and manage goals that affect their behaviour. We aim for the kernel language to reflect these features by having an operational semantics based on the Actor Model of Computation (AMC) (Hewitt, 2010) and its relation to organisations, or *iOrgs* (Hewitt, 2009). Actors have an address and manage an internal state that is private and cannot be shared with other actors in the system. Execution proceeds by sending asynchronous messages from a source actor to the address of a target actor. Synchronous messages can be achieved by sending an actor in an asynchronous message to which the result should be sent. Each message is handled in a separate execution thread associated with the target of the message and the message itself (collectively referred to as a *task*). During task-execution an actor may choose to change its state and behaviour (*becoming* a new actor) that is immediately available to process the next message sent to the target address.

Our claim is that the AMC provides a suitable basis for execution and analysis of the concepts defined in Fig. 3 and can be used to represent the features of a component. The rest of this section lists the key features that must be supported by the kernel actor-based language:

[**adaptability**] Organisational components may change dynamically during a simulation. Resources, individuals, and even departments may move location, and have an affect on results. Furthermore, the behaviour of a component may change over time as information changes within the system. Actors can change behaviour as a result of handling a message.

[**modularity**] Each part of an organisation is intended to perform a business function that can be expressed in terms of a collection of operations. The internal organisation in terms of people, IT systems

and the implementation of various business processes is usually hidden. The AMC provides an interface of message handlers for each actor. Both the state and the implementation of the message interface are hidden from the outside. The specification of an actor in terms of its external interface can be expressed in terms of LTL formulas that constitute the external goal for a component.

[**autonomy**] A key feature of an organisation is that the behaviour of each sub-component is autonomous. A particular department is responsible for its own behaviour and can generate output without the need for a stimulus. The AMC is highly concurrent with each actor being able to spawn multiple threads and over which other actors have no control (unless granted by the thread originator).

[**distribution**] An organisation may be distributed and this may be an important feature of its simulation. Furthermore, we have a requirement that the tooling for organisational analysis and simulation should support distributed concurrent development. The AMC is message-based and seamlessly supports execution in the same address space, via a network connection or in the cloud.

[**intent**] In addition to autonomous behaviour, an organisation component exhibits *intent*. This might take the form of an internal goal that guides the behaviour of the component to ensure that it contributes to the overall mission of the organisation. Although actors do not directly provides support for such goals, we intend to use results from the field of Multi-Agent Systems (van der Hoek, 2008) where support for goal-based reasoning is provided within each agent when determining how to handle messages.

[**composition**] An organisation is an assembly of components. As noted above, the topology of an organisation may be static or dynamic. Actors can be nested in more than one way. Actor behaviours are declared and new actors are dynamically created with an initial behaviour (much like Java classes). The scope of actor behaviours can be nested to provide modularity. Adding a dynamically created actor to the state of a parent actor provides composition. Such actors can be sent as part of messages. If the source actor retains the address, then the communicated actor becomes shared between the source and the target of the message.

[**extensibility**] Our aim is to support a number of simulation and analysis use-cases. As such the kernel language will need to support a collection of independent domains. Whilst we expect the DSLs to target the kernel language it is likely that each domain will have its own fundamental concepts and actions (so-called *Therbligs,* (Stanton, 2006)). We envisage such domain-specific features being defined in the kernel language and then pre-loaded to form an augmented target language for DSL translations.

[**event-driven**] Organisational components cannot rely on *when* communications occur and *where* they originate. In addition, a component may simply cause an event to occur without knowing who will consume the event. This is to be contrasted with message-based communication where the target is always known to the source and where sometimes the message carries information about the source that becomes available to the target. The AMC is based on message passing where the source knows the address of the target. Given that the kernel language is the target of DSL transformations, support for event-based communication becomes an architectural issue where events are simply messages that are sent to an actor container that is responsible for delivering event-messages to dynamically changing collections of actors. Providing that the transformation establishes the correct assembly of actors and conforms to an appropriate message passing protocol then component events are supported without needing to make them an intrinsic part of the kernel.

```
1  module ::=
2    export(number) [id']; binding' in command
3
4  binding ::=
5    id [ (id') ] = exp
6  | act id [(id')] {((pattern') -> command)*}
7
8  exp ::=
9  | exp (exp')
10 | fun (id') exp
11 | proc (id') command
12 | (let | letrec) binding' in exp
13 | const
14 | id
15 | new id [ (exp') ]
16 | case exp { (pattern -> exp)* }
17 | [exp']
18 | functor [ ( [ exps ] ) ]
19 | { (id = exp)' }
20
21 command ::=
22   { command' }
23 | (let | letrec) binding* in command
24 | become id [ ( exp' ) ]
25 | case exp { (pattern -> exp)* }
26 | send exp ( exp' )
27 | exp ( exp' )
```

Figure 4: The ESL Kernel.

Fig. 4 shows the kernel features of a language called ESL that is currently under development. It has been designed to support the features that are

discussed in this section and, as a result, address the problems outlined in section 1. The syntax definition assumes const and id for constants and identifiers, underlines terminals and uses x^y to denote x(yx)*. In overview, a system consists of a collection of modules (line 1) that exports a means of communication. An actor behaviour is introduced as a binding (line 6) that could be nested as a local definition (line 12). A behaviour has state and message handling rules (line 6) and a new actor is created (line 15) by supplying values for the state variables. Pattern matching is used to process messages (`pattern` is not defined) and to dispatch to a command. A command can change the behaviour of the receiver (line 24) or send further messages (line 26). Data values are constants (line 13), lists (line 17), terms (line 18), actors, functions (line 10) and procedures (line 11).

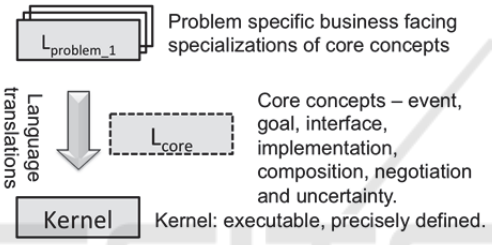

Figure 5: Translation to the Kernel.

Fig. 5 shows the proposed process where many different problem-oriented DSLs are translated in terms of the core concepts to the kernel language where simulation and analysis can be applied.

4 SUMMARY AND NEXT STEPS

We have identified 3 key problems and proposed an approach to solving these in order to make decision making in organisations more effective. Our approach is based on a domain analysis of the core concepts and involves supporting multiple DSLs that can be understood in terms of these concepts and realised in terms of a kernel language used for simulation and analysis. To date we have analysed several EA use-cases in terms of the core concepts and have started to prototype the kernel language. Currently we are developing real-world case-studies to validate and illustrate the proposed approach. For each case-study we will construct: a problem specific specialization of the core concepts, a business facing language constituting concrete syntax for the specialized meta model, and a mapping from this language to the kernel language.

REFERENCES

Tarr, P., Ossher, H., Harrison, W., and Sutton, S.(1999). N degrees of separation: multi-dimensional separation of concerns. Proceedings of the 21st Int. Conf. on Software Engineering, pp. 107-119.

Barros, T., Ameur-Boulifa, R., Cansado, A., Ludovic, H. and MadelaineBarros, E (2009). Behavioural models for distributed Fractal components. Annales des Télécommunications 64(1-2).

Bonabeau, Eric (2002). Agent-based modeling: Methods and techniques for simulating human systems. Proc. of National Academy of Science, USA, 99(Suppl 3).

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Addison Wesley.

Clark, A., and Barn, B (2011). Event driven architecture modelling and simulation. SOSE 2011.

Michelson Brenda (2006). Event-Driven Architecture Overview. Patricia Seybold Group, February 2, 2006.

Yu, E., Strohmaier, M. and Xiaoxue Deng (2006). Exploring Intentional Modeling and Analysis for Enterprise Architecture. Enterprise Distributed Object Computing Conference Workshops. EDOCW '06.

Clark, A., Frank, U., Kulkarni, V., Barn, B. and Turk, D (2013). Domain specific languages for the model driven organization. In Proc. of the 1st Workshop on the Globalization of Domain Specific Languages.

Zachman, J (1999). A framework for information systems architecture. IBM Systems Journal, vol. 38(2/3), 1999.

The Open Group, TOGAF 9.1 White Paper On Intro. to TOGAF Version 9.1 http://www.opengroup.org/togaf/

Wisnosky, D. and Vogel J. (2004). DoDAF Wizdom: A Practical Guide to Planning, Managing and Executing Projects to Build Enterprise Architectures Using the Department of Defense Architecture Framework (DoDAF).

Dardenne, A., Lamsweerde, A. and Fickas, S (1993). Goal-directed requirements acquisition. Science of Computer Programming, Volume 20(1–2).

Object Modeling Group, Business Motivation Model (BMM), v. 1.1, 2010, http://www.omg.org/spec/BMM/1.1/

Donella Meadows (2008). Thinking in systems: a primer. Chelsea Green Publishing.

Reinhartz-Berger, I., Cohen, S., Bettin, J., Clark, T., & Sturm, A. (2013) Domain Engineering: Product Lines, Languages and Conceptual Models. Springer.

Hewitt, C. (2010). Actor model of computation: scalable robust information systems. arXiv:1008.1459.

Hewitt, C. (2009). Norms and Commitment for iOrgs (TM) Information Systems: Direct Logic (TM) and Participatory Grounding Checking. arXiv:0906.2756.

van der Hoek, W., & Wooldridge, M. (2008) Multi-agent systems. *Handbook of Knowledge Representation*, 887-928.

Stanton, N. A. (2006) Hierarchical task analysis: Developments, applications, and extensions. *Applied ergonomics*, *37*(1), 55-79.