TIM: A PROTOTYPE TOOL FOR INTEGRATING SYSTEM DEVELOPMENT METHODS

Richard Green
Mike Brough

Department of Computer Science
Keele University
Staffordshire
ST5 5BG
UK

[rich | mdb3] @cs.keele.ac.uk

ABSTRACT
The notion of multiple representations of a system model is becoming more popular, and is a significant motivation behind the drive to integrate system development methods. Some metaCASE tools claim to allow method integration, but typically still make assumptions that impose constraints on the methods integrated, and the way they are integrated. This paper describes a prototype environment for multiple representations, based on a four-level approach to modelling system development knowledge, and that supports several complementary integration mechanisms.

1. INTRODUCTION
System development methods (SDMs) define modelling tools with which to represent system/application models (models). Modelling tools must satisfy two needs: (i) ‘user-friendly’ communication, (ii) rigorous specification, of a model. Many SDMs can be characterised as considering a set of representations to be a model; model semantics are tied to syntax. The syntax(es) used in such pencil-and-paper SDMs (Brough 1988) inherently involve a trade-off between rigour and user-friendliness, otherwise both demands cannot be addressed. CASE tools supporting such SDMs inherit this paradigm.

A modelling paradigm that separates a model from views of the model overcomes the paradox of the pencil-and-paper paradigm, allowing multiple representations of a single model. We call this alternative modelling paradigm the internal model plus views (IMpV) paradigm (Brough 1988). The IMpV
paradigm considers a model to be *internal*, accessible only through the defined *views*. Modelling abstractions that comprise the internal model are independent of any particular *view*; this controlled redundancy yields a *minimal* model. *Views* provide a selective projection of the abstractions, some focusing upon user-friendliness, others upon rigorous specification. *Views* cannot be inconsistent with one another as they are *derived* from the same (internal) model. Note that ‘multiple views’ can be interpreted in two ways: (i) with a single SDM, different *view types* can project alternative dimensions of the same modelling abstraction, (ii) with integrated SDMs, different *view types* can project the same dimension of the same modelling abstraction in different ways. To support *view types* defined by different SDMs, an IMpV CASE tool must allow a significant degree of SDM integration, otherwise a model remains largely dependent on the *view types* of one SDM.

The rest of this paper is organised as follows. The next section discusses the rationale and general principles of method integration, and identifies complementary integration mechanisms, section 3 explains a levelled-knowledge approach to integration, section 4 describes a prototype tool that implements the approach, and finally section 5 summarises the current state of the prototype and discusses future work.

2. **SDM INTEGRATION**

SDM integration is becoming an increasingly important issue (Olle 1991, Kronlof 1993). Common aims of SDM integration are method extension and method strengthening, both resulting in another ‘improved’ method (Kronlof 1993). Our aim is different; to integrate SDMs to an extent that provides SDM-independence for system development knowledge.

SDM-independent models have recently received some attention (Lyytinen 1994). Two systems that allow some SDM independence are MetaEdit (Smolander 1991, Rossi 1995) a commercially-available metaCASE tool, and the object base management system ConceptBase (Jarke, Nissen 1996).

2.1. **The current state of SDM integration**

MetaEdit (Rossi 1995) is a CASE shell that allows SDMs to be customised or developed from scratch. Customisation includes the ability to define different representations for a single concept, e.g. a different shape for each diagram type by which the concept can be represented. A later version of MetaEdit, MetaEdit+ (Kelly 1996), supports three ‘representation paradigms’: graphical diagrams, matrices (Kelly 1994) and tables. But different notations or different ‘paradigms’ are still representing the same abstractions, albeit in different ways; SDM independence is in representational terms only. As it is unlikely that the abstractions defined by a particular SDM are the optimal set for every individual involved in a development project, integration should allow knowledge to be independent of a particular set of abstractions.
The issue of integrating SDMs which use different modelling abstractions is partially addressed in (Kelly 1996), which describes how MetaEdit+ allows properties to be shared by abstractions of different SDMs. Property sharing is a powerful mechanism, e.g. it allows the attributes belonging to a class in an OO-populated model to be shared with an entity type in a structured-populated model. This mechanism therefore allows some degree of integration between different types of SDM. However, because different types of SDM do not simply define alternative configurations of properties, property sharing provides only a partial solution to SDM integration: how can the mechanism be applied when integrating an SDM that supports higher-order relationships with one that supports only binary relationships? how can we organise the operations of a class when we want to view an OO-populated model using a structured SDM?

ConceptBase (Jarke 1996) allows different perspectives of the same real-world feature to be captured and integrated. The process of integration involves abstracting the elements in different perspectives (we call these elements diagram component types, or dcTypes), into a set of generalised elements (we call these elements model component types, or mcTypes). The mcTypes provide ‘interplay’ between perspectives by relating, via abstraction, dcTypes of different perspectives. Thus in ConceptBase integration is at the representational level; different modelling abstractions cannot be handled.

2.2. Integration mechanisms

To attempt to improve the current state of SDM integration, we identify complementary mechanisms which help improve the degree of integration:

• property sharing: if defined at the SDM level allows properties to be configured into different modelling abstractions in different SDMs, e.g. all instances of concept class can automatically be viewed as instances of concept entityType. This is a fine-grained integration mechanism;

• aliasing: allows integration at the modelling abstraction granularity. More than sharing properties, alias concepts share common behaviour and capabilities.

• composite concepts: enables SDM concepts to be built by aggregation of existing concepts, perhaps belonging to other SDMs, e.g. using entityType to define the structural aspect of class. Granularity is the same as with aliasing;

• mapping: allows knowledge represented in one set of concepts to be transformed into knowledge represented in another set of concepts. There is not always an inverse of transformations.

By providing these integration mechanisms, a significant amount of model independence can be achieved, and the IMpV paradigm supported.

3. KNOWLEDGE LEVELS AND DOMAINS

A common strategy for integrating SDMs is to type concepts that are functionally similar into more general higher-level concepts. An SDM concept is an instance of a higher-level concept.
TIM: A Prototype Tool for Integrating System Development Methods

We have attributed knowledge to one of four levels (Brough 1995), each level at a different number of abstractions from the ‘real world’. Each level is defined in terms of those above, except for the top level, and is described below:

- **Level 0**: value-knowledge is attributable to individual things in the real-world, e.g. ‘register Joe Bloggs on Math203’.
- **Level 1**: refers to problem domain concepts, e.g. ‘register student on course’; register, student and course are level 1 concepts. Level 1 concepts are abstractions of level 0 knowledge.
- **Level 2**: comprises knowledge of SDM concepts, e.g. ‘an entityType is an abstraction of things sharing common structure’. Level 2 concepts are instantiated to define level 1 concepts.
- **Level 3**: concepts are abstractions of level 2 concepts, e.g. ‘a model component type is represented in a particular view type by a view component type’. These abstractions help identify SDM commonality.

Data is not the only aspect of system development knowledge that should be defined at each level. Knowledge of the functional dimension (what needs to be done), and the timing dimension (identifying significant events) of system development knowledge are equally important.

A four-level architecture has been used in other system development-related work (Nissen 1996 (ConceptBase), ISO/IEC 10027 (IRDS), Van Assche 1988 (RUBRIC), Kelly 1996 (MetaEdit+)). Although on the surface these all specify the same four levels, this is not the case. The differences can be characterised by the types of relationship between levels: in the case of (Nissen 1996), the relationship between some levels is the is-a relationship, between others it is the a-kind-of relationship. (ISO/IEC 10027), (Van Assche 1988) and (Kelly 1996) use exclusively the is-a relationship. (Nissen 1996) uses both type and specialisation inheritance, whereas (ISO/IEC 10027), (Van Assche 1988) and (Kelly 1996) use only type inheritance. The relationship between our four levels is the is-a relationship, inheritance is type inheritance.

Within each level, there can exist multiple domains, e.g. each SDM represents a level 2 domain. Knowledge can overlap between domains, defined using the integration mechanisms described in section 2. The degree of overlap/integration dictates the degree of model SDM independence.

4. TIM: A PROTOTYPE METHOD INTEGRATION TOOL

TIM (Tool for Integrating Methods) is a prototype tool intended to illustrate the integration mechanisms identified in section 2, using the levelled-knowledge approach to modelling system development knowledge, described in section 3. A brief discussion of TIMs implementation and functionality follows.

Instances of a pre-defined frame-type (Minsky 1975) store knowledge of concepts, concept properties, and concept instances. A frame instance can be used as either a concept definition frame (Figure 1) which names and describes a concept, a property definition frame (Figure 1) which names and describes a property of a concept, or an instance slot-value frame (Figure 3) which holds an
TIM: A Prototype Tool for Integrating System Development Methods

property value for an instance of a concept. Characteristics of the knowledge captured in TIM vary at each level.

4.1. Defining different levels of knowledge

The definition of a level 3 concept is ‘stand-alone’; no parent concept exists. A level 3 concept cannot be an instance of anything else because there are no higher-level concepts to instantiate. (Green 1997) discusses in more detail the issues of defining a ‘top-level’ concept.

To define a level 3 concept, the Concept Creator dialogue (Figure 2) is used. An instance of a concept definition frame (Figure 1) is created, and the concept’s properties can be defined. Details of each property are stored in a property definition frame (Figure 1). Properties define lower-level slots, which can be of different types; we are currently defining these types. Examples include slotName, which take names of lower-level slots, active, taking functional or executable knowledge, and uninterpreted, filled with passive data, e.g. the meaning of a concept. Typing the slots allows TIM to properly handle slot-fillers when the slots are instantiated.

![Figure 1. Concept and property definition frames](image1.png)

![Figure 2. The Concept Creator dialogue](image2.png)
TIM: A Prototype Tool for Integrating System Development Methods

The Components box of Concept Creator allows a concept to be constructed as an aggregation of same-level concepts, supporting the composite concept integration mechanism described in section 2.

A concept at level 2 or 1 is both an instance of a parent concept, and a lower-level concept which itself can be instantiated. Meta-data about a ‘middle-level’ concept is recorded in instance slots when the parent concept is instantiated. As slot types of the parent concept are identified when defined, TIM appropriately handles the values entered, e.g. values in slotName slots are used to name slots of the new concept. Each value entered for an instance slot is stored in an instance slot-value frame (Figure 3). Like a level 3 concept, a middle-level concept can be defined as a composite of same-level concepts.

Figure 3. A modelCompType instance slot-value frame for entityType

Level 0 knowledge is captured by instantiating level 1 concepts; an instance slot-value frame (Figure 3) is created for each value of each instance slot. This knowledge comprises the ‘real-world’ of a particular domain; (Van Assche 1988) calls it the prototyping level, as a system prototype would use this data.

4.2. Defining alias concepts

An alias is a domain-specific name for an existing same-level concept, e.g. the same level 2 concept might be referred to as class in the OMT domain, and objectType in the Shlaer-Mellor domain. Aliasing is one of the integration mechanisms identified in section 2. The semantics and properties of each alias are identical, and an alias can itself be aliased, instantiated and used as a part of a composite concept.

4.3. Browsing the knowledge

TIM provides browsers for accessing the different levels of system development knowledge. The Concept Browser presents meta knowledge (e.g. concept meaning or components), of all concepts at a particular level, or within a specific domain. This browser can be used to view concepts at levels 3, 2 and 1. The Instance Browser presents concept instances and their instance-slot values (e.g. values of the attributes slot for student, an instance of class, might be name, DoB and address). The Instance Browser can be used to view instances at all levels, and provides access to level 1 knowledge, which inappropriate for the Concept Browser.
TIM: A Prototype Tool for Integrating System Development Methods

5. STATUS AND FUTURE WORK

TIM is intended to support several complementary SDM integration mechanisms in an effort to achieve integration to an extent characterised by the IMpV modelling paradigm, with the corollary of improved SDM independence for system models.

Static concepts are not the only components of an SDM; process and heuristics are others. Functional knowledge has potential in supporting more aspects of SDMs, and we intend to investigate this potential.

TIM currently supports the alias and composite concept mechanisms, property sharing will be implemented soon. Mapping, supporting functional knowledge, is partially implemented. The mapping integration mechanism introduces other issues, as it necessitates slots that take active or executable knowledge. There may be several ways such functional knowledge could be fired (manually, automatically upon certain events, or continuously). Developing an appropriate interface for entering such functional knowledge is a difficult issue. In TIM we simply make available the Prolog language for filling active slots, but this would be inappropriate for most users.

The taxonomy of slot-types will be investigated further: current thoughts include active, generalising constraints and rules, slotName, taking names of lower-level slots, uninterpreted, filled with passive data, and state, recording the current state of an instance.

When the implementation of TIM is more complete, we intend to evaluate the effectiveness of the integration mechanisms by building a minimal multi-SDM model consisting of tightly-related SDM concepts. SDM-dependent view types into the model will be defined, and the effectiveness of the integration mechanisms will be measured by the degree to which the view types can access knowledge stored in abstractions of other SDMs.

6. ACKNOWLEDGEMENTS

The authors wish to acknowledge Dr. Pearl Brereton and Prof. David Budgen for their helpful comments on this paper, and the two anonymous reviewers whose feedback also helped improve it. Acknowledgement is also due to the EPSRC and Structured Software Systems Ltd. for funding the project of which this work is a part.

7. REFERENCES

TIM: A Prototype Tool for Integrating System Development Methods


