Applications of Abduction: A Unified Framework for Software and Knowledge Engineering

Tim Menzies[1]

Artificial Intelligence Department, School of Computer Science & Engineering, University of NSW Sydney, Australia, 2052;
Email: `timm@cse.unsw.edu.au`; Url: `www.cse.unsw.edu.au/~timm`

Abstract. A new framework is proposed in which software engineering (SE) is the construction of a search space and knowledge engineering (KE) is the constructing the intelligence to control the traversal of that space. Conventional information systems and object-oriented notations can specify the search space. An abductive inference engine can implement the intelligent control. This unified framework supports conventional SE/KE approaches, plus automatic screen generation, conflict resolution in requirements capture, code library management, optimised code generation, and automatic testing tools.

5000 words. Submitted to the 10th European Workshop on Knowledge Acquisition, Modeling, and Management Sant Feliu de Guixols, Catalonia, Spain October 15 - 18, 1997

1 Introduction

Conventional software descriptions can be divided into several perspectives that include the *events* and *activities* in which *data* is processed (§2.2). Knowledge-level modeling [20, 21] adds an extra perspective; i.e. the *goal* of the program and the *options* which must be *reflected on* while exploring that goal.

This article argues that these perspectives can be connected as follows. Roughly speaking, conventional software notations will be used to define some "road map" and knowledge engineering tools will be used to control how we drive over that road map. More precisely:

- Conventional software engineering perspectives will define a search space. We will model this search space as a directed graph.
- Knowledge-level tools will define the search space traversal control knowledge. We will model this search space traversal control using a graph-theoretic abduction inference engine (§2.1) which processing graphs generated from information systems (IS) process models (§2.2) and object models (§2.3).

Such a single framework would be useful for many applications since they are often a mixture of SE and KE. For example, consider the following MIS application:

> We want a few screens to let students enroll themselves into their tutorials. If they can't get the tutorials they want, we can offer them next-best options. Lecturers should be able to check the current enrollments.

The case to be made here is that a single framework can offer a non-trivial level of support for the following SE/KE tasks associated with the processing of these requirements:

- *Object modelling* (§2.3): For RAM-based constructs, an OO notation might be more succinct for modelling purposes. OO also allows us to model procedures along with their associated data structures.
- *Business process modelling* (§2.2, §3.1): The requirements beg the question "what does the lecturer do once they have checked the enrollments?". They might realise they need to hire more tutors or even abandon the subject due to low enrollments. An analysis of the business processing might detect new requirements (e.g. an email button to let the lecturer forward the current enrollment figures to the head of school).
- *Data modelling* (§3.2): Assuming the use of a relational persistence mechanism, the student records must be mapped to disc in a normalised form.
- *Requirements modelling* (§3.3): The problem owners (students, lecturers developers) must be consulted with a view to creating a consistent specification. Note that the requirements of different owners may conflict.
- *Dialogue modelling* (§3.4): The mental model of the users must be mapped into the screen design. Related and common actions must be mapped into the same screen.
- *Planning* (§3.5): In the case of students not being able to get their desired tutorial, they may need to either get another time slot or drop the subject. An intelligent planning system could assist in the search through these options, if it had knowledge of course structures (including prerequisites), student goals, and known constraints. Planning is one example of the kind of knowledge-level tasks which may extend a standard software engineering implementation.
- *Code Library Management* (§3.6): Developers building this system would like an intelligent assistant to propose what existing software modules may be relevant to the current problem.
- *Testing* (§3.7).
- *Code generation* (§3.8): Ideally, the final system executes quickly.
- *Maintenance.* We have discuss maintenance in this abductive framework elsewhere [17].

Note that the view of knowledge-level modeling taken in this paper is closer to Newell's SOAR approach than the KADS problem-solving methods approach [26]. For more on this approach to knowledge-level modeling, see [15].

2 The Framework

This framework is based around the HT4 graph-theoretic abduction engine (§2.1). HT4 searches for consistent portions of some background theory which are relevant to some task. The approach to abduction was originally defined for the validation of qualitative neuroendocrinological theories [16] (§3.7). Once developed, it was noted that the internal data structures of the algorithm made little commitment to qualitative reasoning. Indeed, the algorithm could execute over any representation reduced to a directed graph of literals; e.g. business process graphs (§2.2) connecting objects (§2.3).

2.1 Graph-Theoretic Abduction

Abduction is the search for assumptions $\mathcal{A}$ which, when combined with some theory $\mathcal{T}$ achieves some set of goals $\mathcal{OUT}$ without causing some contradiction [4]. That is:

- EQ_1: $\mathcal{T} \cup \mathcal{A} \vdash \mathcal{OUT}$;
- EQ_2: $\mathcal{T} \cup \mathcal{A} \not\vdash \perp$.

HT4 caches the proof trees used to satisfy EQ_1 and EQ_2. These are then sorted into *worlds* $\mathcal{W}$: maximal consistent subsets (maximal with respect to size). Each world condones a set of inferences. A world's *cover* is the size of the overlap between $\mathcal{OUT}$ and that world. In the case of multiple worlds being generated, a customisable assessment operator is used to select the preferred world(s). The "best" assessment operator is domain dependent. We return to this point below (§3.5).

For example, consider the task of achieving certain $\mathcal{OUT}$puts using some $\mathcal{IN}$puts across the knowledge shown in Figure 1. In that figure:

- `x` $\overset{++}{\rightarrow}$ `y` denotes that `y` being `up` or `down` can be explained by `x` being `up` or `down` respectively;
- `x` $\overset{--}{\rightarrow}$ `y` denotes that `y` being `up` or `down` could be explained by `x` being `down` or `up` respectively.

Each edge in Figure 1 is augmented with two pieces of meta-information which are explored subsequently. However, the notation is introduced now:

1. Each edge is annotated with a heuristic weight representing how expensive it is to make that inference. Most edges have cost 10, but the edge `corporateSpending` $\overset{++}{\rightarrow}$ `investorConfidence` requires a large amount of book-keeping by associate accountancy packages to measure the subjective measure `investorConfidence`. Hence, this edge has a weight of 100[1]. We will discuss the use of this edge weight in §3.8.

[1] It is easier to measure `publicConfidence` via simple telephone surveys. Hence, the cost of its input-edge is only 10.

2. Figure 1 is a combination of the opinions of two authors: *Dr. Thick* (whose contribution is drawn with thick lines) and *Dr. Thin* (whose contribution is drawn with thin lines). Observe the apparent conflict in the middle of Figure 1 on the left-hand-side. *Dr. Thick* believes `foriegnSales` $\overset{++}{\rightarrow}$ `companyProfits` while *Dr. Thin* believes `foriegnSales` $\overset{--}{\rightarrow}$ `companyProfits`. We will discuss the resolution of this conflict in §3.3. 110

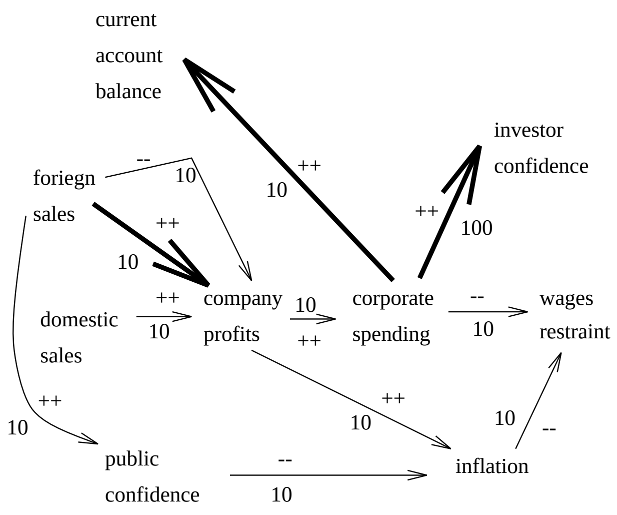

Fig. 1. Some economics knowledge.

$\mathcal{P}$[1]: `domesticSalesDown, inflationDown`
$\mathcal{P}$[2]: `foriegnSalesUp, publicConfidenceUp, inflationDown`
$\mathcal{P}$[3]: `domesticSalesDown, companyProfitsDown, corporateSpendingDown, wagesRestraintUp`
$\mathcal{P}$[4]: `domesticSalesDown, inflationDown, wagesRestraintUp`
$\mathcal{P}$[5]: `foriegnSalesUp, publicConfidenceUp, inflationDown, wagesRestraintUp`
$\mathcal{P}$[6]: `foriegnSalesUp, companyProfitsUp, corporateSpendingUp, investorConfidenceUp`

Fig. 2. Proofs from Figure 1 connecting $\mathcal{OUT}$= {`investorConfidenceUp, wagesRestraintUp, inflationDown`} back to $\mathcal{IN}$puts= {`foriegnSalesUp, domesticSalesDown`}.

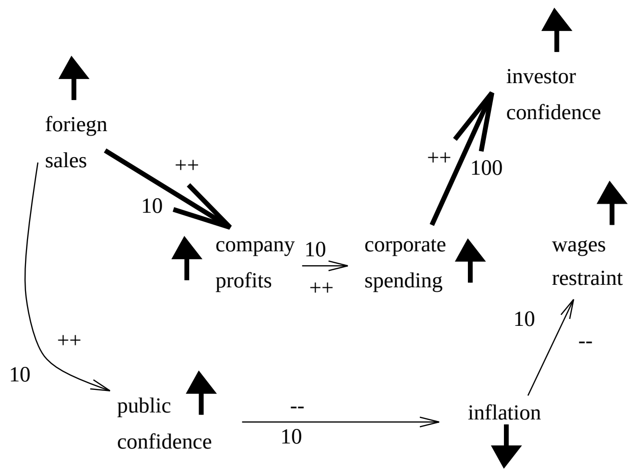

Fig. 3. World #1 is generated from Figure 1 by combining $\mathcal{P}[2]$, $\mathcal{P}[5]$, and $\mathcal{P}[6]$. World #1 assumes `companyProfitsUp` and covers 100% of the known $\mathcal{OUT}$puts.

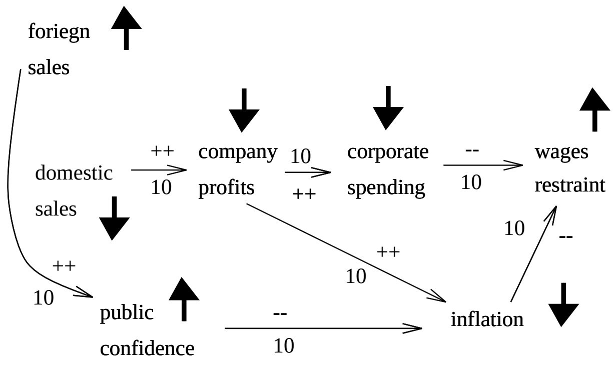

Fig. 4. World #2 is generated from Figure 1 by combining $\mathcal{P}[1]$, $\mathcal{P}[2]$, $\mathcal{P}[3]$, and $\mathcal{P}[4]$. World #2 assumes `companyProfitsDown` and covers 67% of the known $\mathcal{OUT}$puts.

In the case of the observed $\mathcal{OUT}$puts being {`investorConfidenceUp, wagesRestraintUp, inflationDown`}, and the observed $\mathcal{IN}$puts being {`foriegnSalesUp, domesticSalesDown`}, HT4 can connect $\mathcal{OUT}$puts back to $\mathcal{IN}$puts using the proofs of Figure 2. These proofs may contain controversial assumptions; i.e. if we can't believe that a variable can go `up` and `down` simultaneously, then we can declare the known values for `companyProfits` and `corporateSpending` to be controversial. Since `corporateSpending` is fully dependent on `companyProfits` (see Figure 1), the key conflicting assumptions are {`companyProfitsUp, companyProfitsDown`} (de-noted *base controversial assumptions* or $\mathcal{A}_b$). We can used $\mathcal{A}_b$ to find consistent belief sets called worlds $\mathcal{W}$ using an approach inspired by the ATMS [2]. A proof $\mathcal{P}[i]$ is in $\mathcal{W}[j]$ if that proof does not conflict with the environment $\mathcal{ENV}[j]$. In our example, $\mathcal{ENV}[1]$={`companyProfitsUp`} and $\mathcal{ENV}[2]$={`companyProfitsDown`}. Hence, $\mathcal{W}[1]=\{\mathcal{P}[2], \mathcal{P}[5], \mathcal{P}[6]\}$ and $\mathcal{W}[2]=\{\mathcal{P}[1]\ \mathcal{P}[2]\ \mathcal{P}[3], \mathcal{P}[4]\}$ (see Figure 3 and Figure 4). Note that while the background theory (Figure 1) may be inconsistent, the generated worlds are guaranteed to be consistent.

HT4 places few restrictions on the representations it can process. The above process is defined for any representation that can be mapped into a directed graph of literals (the internal data structure of HT4). Many common knowledge representations can be mapped into such graphs. Propositional rule bases can be viewed as graphs connecting literals from the rule left-hand-side to the rule right-hand-side. Horn clauses can be viewed as a graph where the conjunction of sub-goals leads to the head goal. In the special (but common) case where the range of all variables is known (e.g. propositional rule bases, strongly-typed variables), this graph can be converted into a ground form where each vertex is a literal. We describe below (§2.2, §2.3) translators for knowledge expressed in standard information systems (IS) notations.

2.2 Process Modelling

The "Olle-126" is a review of common IS notations. T.W. Olle chaired the "Design and Evaluation of Information Systems" technical committee (TC8.1) of the International Federation of Information Processing [22]. This committee offers a coherent overview of 33 commercial methodologies for planning, business analysis, and design of information systems. Their report was developed via an extensive committee evaluation process, plus a special workshop to discuss and review its framework. That report found that beneath the studied methodologies are 126 common components divided into three categories: *data*, *process*, and *behaviour*. The *data* perspective describes the entities connected up by the other two perspectives. *Processes* connect activities which users may perform for some time. *Behaviours* connect instantaneous events.

For our purposes, we make certain changes to the "Olle-126" (see Figure 5). The Olle IS methodologies were developed before the wide-spread use of object technology. Therefore, we replace the Olle *data perspective* with an *object perspective* (§2.3). The Olle-*process* and *behaviour* perspectives have a similar meta-level structure (e.g. pre-conditions to activities/events). Hence, we will use the term Process to denote the representations of the Scenarios elicited during requirements capture [25]. A Process is a Graph containing Flows between Actions (which may be either Activities or Events). Flows have pre Conditions and post Conditions (which must just be True). All components of a Process are Concepts (recall Figure 7) and may use services defined in BusinessClasses; i.e. the *object* perspective.

Note the Concept abstract superclass at top of Figure 5. This will be explained later (§3.4).

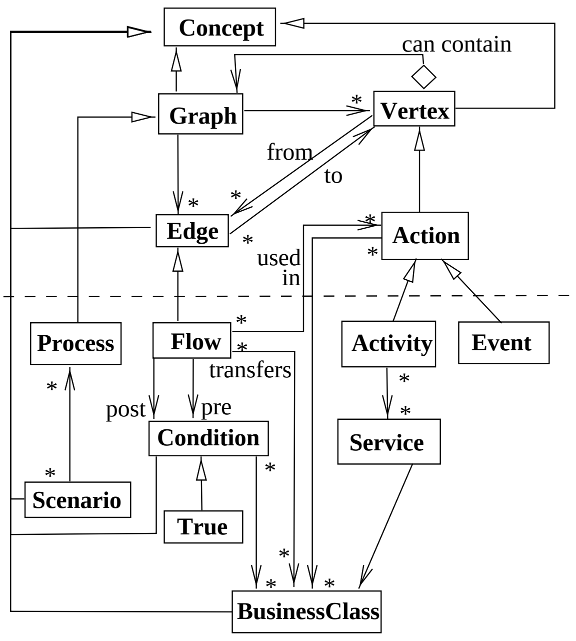

Fig. 5. Business process graphs (drawn using the notation of §5). See Figure 8 for more details on the region above the dotted line.

2.3 Object Modelling

The *object* perspective of this framework (Figure 6) is constrained by what structures can be converted into the directed graphs of HT4. Hence, it is much simpler than some OO meta-models (e.g. UML [1]). However, within those limits, we can still represent many common OO constructs. Each BusinessClass handles a set of Responsibilities which are managed by a set of Attributes and Operations, both of which are defined by their returnType. Operations may be defined by some Pseudo-code. Responsibilities may Collaborate with other BusinessClasses. Each Collaboration (be it an Association or an Aggregation) is a named Relationship specifying Cardinality ranges (e.g. 1 to 1, 1 to many, 2 to 4, etc). Note that apart from IS business process modeling, Figure 5 could also be used as a meta-model for other common flow types such as Harel statecharts [9] and KADS interpretation models [26].

The key feature of Figure 6 is the usedValues of the Attributes. This is generated by an analysis of the Process graphs. If we only ever test or set (e.g.) `1:weekDay=monday, 2:weekDay=tuesday`, then we need not define seven vertices for weekday in the graph (two will suffice). Similarly, if we only ever test or set `temperature>7, temperature=23`, then we only need `temperature` vertices for `1:temperature<7`, `2:temperature=7`, `3:7<temperature<23`,

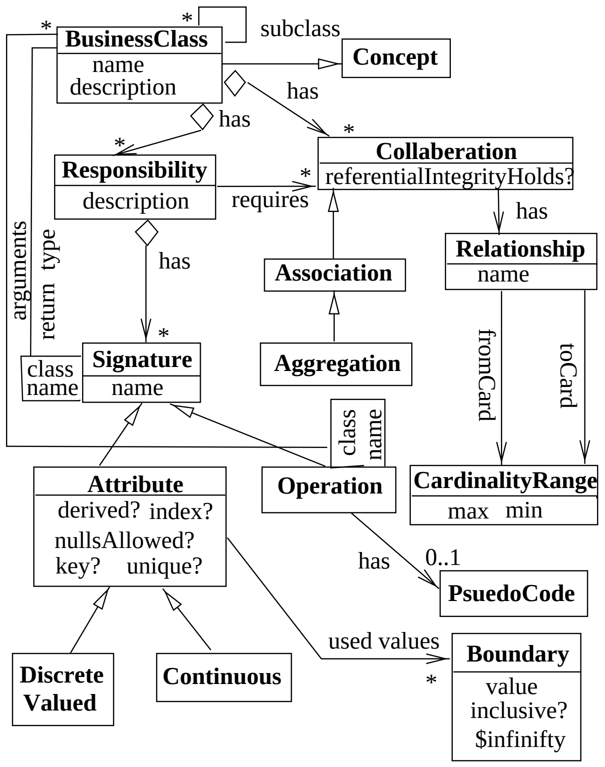

Fig. 6. The *object* perspective (drawn using the notation of §5).

`4:temperature=23`, `5:temperature>23`; i.e we can divide continuous variables into a finite number of discrete ranges. A test in the code for (e.g.) `temperature>7` translates into a test for graph vertices `3 or 4 or 5`.

An argument could be made that Process Graphs are not necessary since these connections could be deduced from the Collaborations and the Operation source code of BusinessClasses. We do not take this approach for two reasons. Firstly, while OO Collaborations describe how a class can traverse to another class, they do not say which traversals are made in the context of a particular business goal. That is, Process Graphs give an extra level of information above that of a Collaboration. Secondly, it is a difficult task to automatically infer calling graphs between Operation source code. Haynes & Menzies [13] had to use an approximate method for an untyped OO language (Smalltalk). Murphy shows that call graphs from different generators can vary widely [19]. Hence, we prefer explicit connection knowledge to be provided in the conceptual modelling.

3 Using the Framework

This section argues that the above framework could handle the SE and KE tasks listed in the introduction.

3.1 Software Engineering Modelling

The base notations of this framework are deliberately reversed-engineered from standard OO and IS methodologies (§2.2,§2.3). This allows us to argue that a tool based on this framework could process much of standard software engineering. For example, the business process modelling of our tutorial assignment program could be performed using IS notations (recall Figure 5).

3.2 Data Modeling

The attributes of Attribute (Figure 6), plus the Collaboration.referentialIntegrityHolds? provides enough information to support an auto-conversion of our system into an underlying relational engine. The conversion of OO models to a relational schema is not a research issue for this project. Commercial tools already exist that can automate this task (e.g. [23]). Lastly, we see no conflict of the conceptual model offered here (Figures 5&6) and common OO models. Process Graphs, for example, are a subset of the statecharts of UML [1].

3.3 Requirements Modelling

Requirements modeling (RM) is the process of assisting a community of business users to move to a common position. Our worlds-generation approach offers RM support. Recall the apparent conflict between *Dr. Thick* and *Dr. Thin* in Figure 1. The worlds of Figure 3 and Figure 4 tell us:

- *Dr. Thin*'s contributions can be found in two worlds; i.e. with respect to the problem of $\mathcal{OUT}$puts= {`investorConfidenceUp, wagesRestraintUp, inflationDown`}, and $\mathcal{IN}$puts= {`foriegnSalesUp, domesticSalesDown`}, a single author's opinions are inconsistent.
- Both authors contributions exist in the same consistent world ($\mathcal{W}$[1]); i.e. the apparent conflict of *Dr. Thick* and *Dr. Thin* did not matter for the analysed problem. If this was true for all the analysed problems, then we could declare that for all practical purposed, *Dr. Thin* and *Dr. Thick* are not really disagreeing.
- *Dr. Thin* may wish to review their opinion that `foriegnSales` $\overset{--}{\rightarrow}$ `companyProfits` since, in terms of the studied problem, this proved to explain less of the required behaviour that *Dr. Thick*'s option that `foriegnSales` $\overset{++}{\rightarrow}$ `companyProfits`.

HT4 has technical advantages over other conflict resolution approaches:

- Easterbrook [3] lets users enter their requirements into an explicitly labeled viewpoints. He makes the simplifying assumption that all such viewpoints are internally consistent. HT4 has no need for this, potentially, overly-restrictive assumption. HT4 can handle inconsistencies within the opinions of a single user. That is, HT4 can analyse conflicts at a finer granularity than approaches based on manually-entered viewpoints (e.g Easterbrook or Finkelstein *et. al.* [5]).
- Easterbrook's Synoptic tool only permits comparisons of two viewpoints [3, p113]. HT4 can compare N viewpoints.
- We have found that it easier to build efficient implementations [14, 15] using the above graph-theoretic approach that using purely logical approaches (e.g. [10]).
- HT4 places few restrictions on the representations it can process (§2.1).

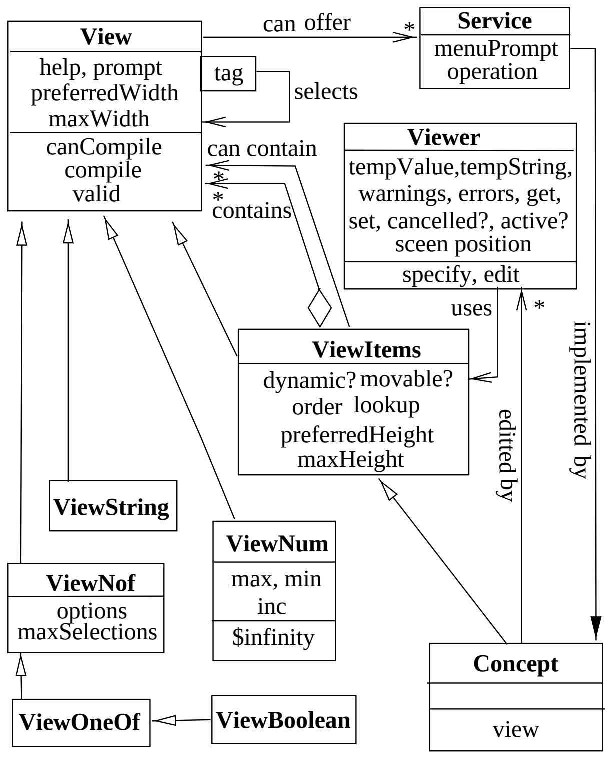

Fig. 7. The View hierarchy (drawn using the notation of §5).

3.4 Dialogue Modelling

Menzies and Spurret discuss the MYLE prototype for automating the generation of the interface [18] for object-oriented systems. In MYLE, each business object reported its contents and editing rules as an aggregation of View instances (see Figure 7). For example, a Date instance could report itself as a ViewItems instance containing a ViewNum instance for the number of the day in the month, a ViewOneOf instance for the name of the month, and a ViewNum instance for the year. When interacting with the user, a Viewer instance handles the screen activity; e.g. screenPosition and temporary copies of the current value and the string on the screen. When the user entered a string, a quick peek at the string (View.canCompile) looked for any gross errors before View.compile converted the string into an internal value. This value was then checked by View.valid without errors and warnings being stored back in the Viewer. A View can also offer a set of Services back to the user; i.e. actions that the user can initiate from the screen. These Services are implemented by the instance from which the screen is generated.

MYLE auto-generated one window for each business object. Each window displayed the Views from that object and was controlled by Viewers. The resulting screens were cluttered with numerous small editors. This was inappropriate for many users since they were always switching from window to window to perform common tasks. A better style of interface would be to auto-generate screens which clump together related processing on the same screen. Given that (i) we have knowledge of the business processes that use objects and (ii) each object can offer a set of View objects, then, we could auto-configure the dialogue layer in a superior manner to MYLE. The knowledge for this clumping can come from the Process Graphs.

3.5 Planning and Knowledge-Level Tasks

We have argued elsewhere [15] that a wide-range of knowledge-level tasks map into a variant of HT4 that supports customisable inference assessment operators called best. For example:

- *Single fault diagnosis* favours favours worlds with only one input and greatest number of outputs.
- *Explanation* can be characterised as the process of favouring the Worlds which contain the most number of things that the user has seen before.
- *Tutoring* is an extension to explanation. If the best explainable Worlds were somehow sub-optimum, then we could then make a entry is some log of teaching goals that we need to educate our user about the Edges which are not in their best explainable Worlds but are in other, more optimum, Worlds.
- *Planning* is the search for a set of operators that convert some current state into a goal state. Given a set of operators, we could partially evaluate them into an and-or Graph they propose between literals. For planning, we could favour the World with the simplest Proofs. One application of least-cost planning is optimised code generation (§3.8).

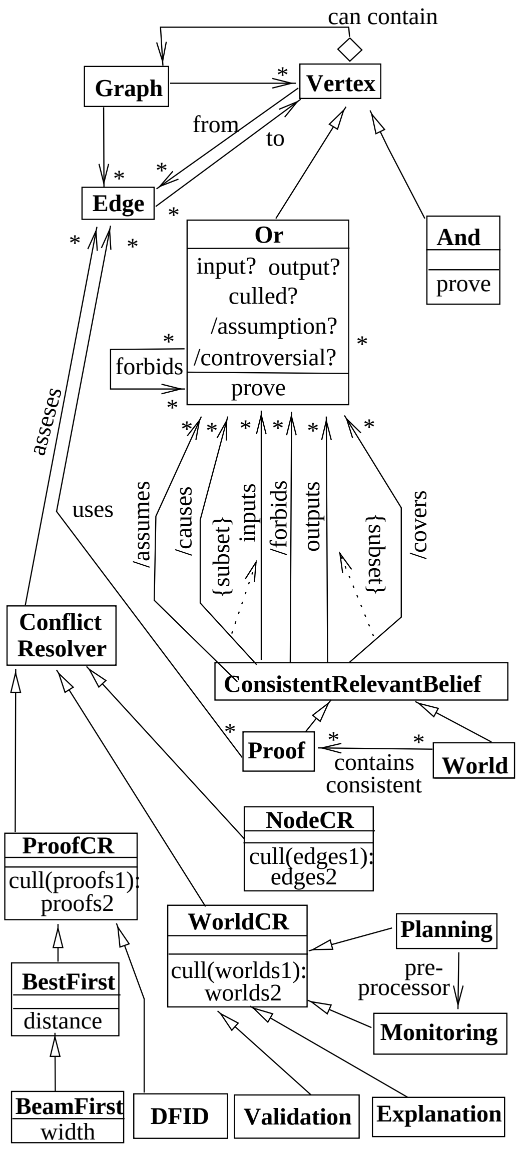

Fig. 8. HT4 (drawn using the notation of §5).

- Once generated, the best planning Worlds could be passed to a *monitoring* system. As new information comes to light, we could reject the plans (Worlds) which contradict the new information.
- *Validation.* See §3.7.

The ConflictResolver (CR) hierarchy of Figure 8 characterises bests into the information they require before they can run:

- *Vertex-level* NodeCR assessment operators can execute at the local-propagation level; e.g. use the edges with the highest probability.
- *Proof-level* ProofCR assessment operators can execute when some proofs or partial proofs are known; e.g. BestFirst, BeamSearch, and DFID (depth-first iterative deepening).
- *Worlds-level* WorldCR assessment operators can execute when the worlds are known; e.g. Validation (§3.7), Planning, Monitoring, and Explanation.

```
proof(2,
      % assumes
      [publicConfidenceUp],
      % causes
      [foriegnSalesUp],
      % inputs
      [foriegnSalesUp],
      % path
      [foriegnSalesUp, publicConfidenceUp,inflationDown],
      % forbids
      [foriegnSalesDown, publicConfidenceDown,inflationUp],
      % outputs
      [inflationDown],
      % covers
      [inflationDown]
      ).
```

Fig. 9. An instance of Proof (Figure 8) for $\mathcal{P}[2]$ of Figure 2.

An example of a Proof instance is shown in Figure 9. Proofs store where they start (Proof.inputs) and where they finish (Proof.outputs), and the path taken between them. For reasons of efficiency, a forbids set is maintained showing what is blocked by a Proof. Proofs also store what was assumed along the path.

3.6 Code Library Management

Given that we can support an explanation strategy (§3.5), we should also be able to support intelligent browsing of source code libraries. When faced with a new problem, and a source code library that the programmer has only limited experience with, the programmer could ask the source code library to offer explanations of how it might achieve the programmer's goals. If the source code was available as a directed graph connecting functions and the programmer's goals were $\mathcal{IN}$put, $\mathcal{OUT}$put pairs, then the "explanation" would be a list of modules within the library that were relevant to the programmer's current project.

3.7 Testing

Verification is the detection of syntactic anomalies in the structure of a program [24]. Validation is the testing of a program with reference to some external semantic criteria [29]. Some kinds of validation can be automated (e.g. test suite coverage) while others require extensive user involvement (e.g. assessments of screen "usability").

Our approach could be used to automatically perform certain automatic verification tests. For example:

- *Circularities* could be detected by computing the transitive closure of the and-or graph. If a vertex can be found in its own transitive closure, then it is in a loop.
- *Ambivalence* (a.k.a. inconsistency) could be reported if more than one world can be generated. That is, given a set of $\mathcal{IN}$puts, mutually exclusive conclusions can be made.
- *Un-usable rules* could be detected if the edges from the same part of the conceptual model touch vertices that are incompatible (defined by $\mathcal{I}$).

Given a library of known behaviours (i.e. a set of pairs $< \mathcal{IN}, \mathcal{OUT} >$), this approach could perform an automatic validation check. Abductive validation uses a World assessment operator that favours the Worlds with largest number of covered outputs. For example, returning to the worlds shown in Figure 3 and Figure 4, the validation algorithm would note that the overlap of $\mathcal{W}[1]$ and $\mathcal{OUT}$ is all of $\mathcal{OUT}$ and the overlap $\mathcal{W}[2]$ and $\mathcal{OUT}$ does not include `investorConfidenceUp`; i.e. $\mathcal{W}[1]$ explains 100% of the desired $\mathcal{OUT}$puts while $\mathcal{W}[2]$ only explains 67%. The maximum cover is 100%; i.e. (i) their exist a set of assumptions ({`cUp`}) which let us explain all of $\mathcal{OUT}$; and (ii) this model has based abductive validation.

Note that this process can be summarised as: "can a model of X explain known behaviour of X?". We have argued elsewhere [16] that this is the non-naive implementation of KBS validation since it handles certain interesting cases:

- If a model is globally inconsistent, but contains local portions that are consistent and useful for explaining some behaviour, HT4 will find those portions.
- In the situation where no current model explains all known behaviour, competing models can be assessed by the extent to which they cover known behaviour. Model X is definitely better than model Y if model X explains far more behaviour than model Y.

Elsewhere, we have shown examples where this validation approach has faulted theories published in the international peer-reviewed neuroendocrinological literature. Interesting, the detected faults were found using the data published to support those theories [16].

An interesting variant on our validation approach are the automatic test suite generation procedures offered by the dependency-network approaches of Ginsberg [7, 8] and Zlatereva [27, 28]. The dependencies between rules/conclusions are computed and divided into mutually consistent subsets. The root dependencies of these subsets represent the space of all reasonable tests. If these root dependencies are not represented as inputs within a test suite, then the test suite is incomplete. Test cases can then be automatically proposed to fill any gaps. Such test cases can be generated in HT4 be setting $\mathcal{IN}$ to the roots of the graphs within HT4 and $\mathcal{OUT}$ to be all the graph vertices that are not $\mathcal{IN}$puts (but this will be a slow inference process).

3.8 Optimised Code Generation

Let the *cost* of a world be the maximum cost of the proofs within it. Consider a best assessment operator which maximises the covered? inputs while minimising the cost of the Worlds. A world so-generated would be a least-cost plan for achieving the maximum desired goals in the least time. If this least-cost plan was reported in (e.g.) C code, then this "plan" would be become an automatically generated optimised program.

Kanovich [11] uses exactly this scheme to to optimise functional evaluation. Kanovich's implementation of his approach runs over Pascal source code to extract the headers of each source procedure. The inputs and outputs of the procedures are studied and each procedure header is added to a dependency network that is an and-or graph. Pre-conditions to executing the procedure are stored as conjunctions upstream of proposition representing the procedure call. Possible outputs from each procedure are out-edges from the procedure proposition. Kanovich's planner runs over this network to return the world with minimum cost. This world is then compiled into the main procedure of a Pascal program which calls the procedures in the original source code.

In a similar approach, Freeman-Benson *et. al.* [6] discuss code extraction from a constraint solver. Such a "solution" is a directed acyclic data-flow graph (which is very similar to an HT4 World) whose nodes are constraint methods. Optimised code-generation is just a depth-first traversal of this network and the printing of the method calls in each node.

Returning to our example (Figure 1), the maximum cost of a proof in $\mathcal{W}[1]$ is 120 (from $\mathcal{P}[6]$) and the maximum cost of a proof in $\mathcal{W}[2]$ is 30. Hence, we may prefer to generate code from $\mathcal{W}[2]$ (Figure 4) since this will run significantly faster than $\mathcal{W}[1]$. However, this fastest program would not be able to achieve all the program goals (the $\mathcal{OUT}$put set) of a program generated from $\mathcal{W}[1]$; i.e. in this example, we have have to make a choice between runtime speed and completeness. Note that our approach supplies the information required to intelligently make this choice.

4 Conclusion

We have offered a unified abductive architecture for unifying software engineering and knowledge engineering approaches. Conventional SE notations define a search space which we can intelligently search using KE tools. One interesting feature of this architecture is the application of a knowledge engineering technique (abduction) to standard SE. The advantages of this approach is that a range of tasks can be implemented in the single framework: requirements modelling; business process modelling; OO modelling; data modelling; dialogue modelling and automatic screen generation; various knowledge-level tasks including planning; code library management (which we view as a synonym for explanation); testing; and optimised code generation (which we view as a synonym for least-cost planning).

Currently, HT4 is operational and the translator between the Process Graphs and the BusinessClasses is being implemented. Once implemented, we hope to have an SE/KE tool that covers nearly the whole life cycle of MIS applications from requirements capture during analysis, through to screen generation, coding, and testing. To cover the whole life cycle, this abductive framework would have to be extended to maintenance (and this is discussed elsewhere [12, 17]).

5 Appendix: OO Notation

.

This section describes our OO notation. In diagrams like Figure 5, boxes denote classes and contain class name (top), attributes (middle, optionally) and operations (bottom, optional). Boolean attributes are followed by a "?". Arrows denote relationships: diamonds denote aggregation relationship; triangles denote inheritance; otherwise it is a general association. Boxes on the sides of classes denote qualified associations; i.e. lookup tables on some field, e.g. View.tag. Relationships can be augmented with multiplicities: "*" means "many"; default is 1 to 1. In the text, X denotes a class and X.y denotes the operation/attribute y of class X. Class names are always shown with a Leading capital letter while operation/attribute names have a leading lowerCase letter. A leading "/" before an attribute or a relationship denotes an attribute or relationship that can be derived from other attributes or relationships.

References

1. G. Booch, I. Jacobsen, and J. Rumbaugh. *Version 1.0 of the Unified Modeling Language*. Rational, 1997. http://www.rational.com/ot/uml/1.0/index.html.
2. J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.
3. S. Easterbrook. *Elicitation of Requirements from Multiple Perspectives.* PhD thesis, Imperial College of Science Technology and Medicine, University of London, 1991. Available from *http://research.ivv.nasa.gov/~steve/papers/index.html.*

4. K. Eshghi. A Tractable Class of Abductive Problems. In *IJCAI '93*, volume 1, pages 3–8, 1993.
5. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibe. Inconsistency Handling In Multi-Perspective Specification. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.
6. B.N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33:54–63, 1 1990.
7. A. Ginsberg. A new Approach to Checking Knowledge Bases for Inconsistentcy and Redundancy. In *Proc. 3rd Annual Expert Systems in Government Conference*, pages 102–111, 1987.
8. A. Ginsberg. Theory Reduction, Theory Revision, and Retranslation. In *AAAI '90*, pages 777–782, 1990.
9. D. Harel. On Visual Formalisms. In J. Glasgow and B. Chandrasekaran N.H. Narayanan, editors, *Diagrammatic Reasoning*, pages 235–271. The AAAI Press, 1995.
10. A. Hunter and B. Nuseibeh. Analysing Inconsistent Specifications. In *International Symposium on Requirements Engineering*, pages 78–86, 1997.
11. M.I. Kanovich. Effecient Program Synthesis: Semantics, Logic, Complexity. In *Theoretical Aspects of Comptuer Software, September, 1991, Sendai, Japan*, 1991.
12. T. Menzies and D. Fensel. 42 Kinds of Knowledge Mainteance, 1997. In preperation.
13. T. Menzies and P. Haynes. Empirical Observations of Class-level Encapsulation and Inheritance. Technical report, Department of Software Development, Monash University, 1996.
14. T.J. Menzies. On the Practicality of Abductive Validation. In *ECAI '96*, 1996.
15. T.J. Menzies. Applications of Abduction: Knowledge Level Modeling. *International Journal of Human Computer Studies*, 45:305–355, September, 1996.
16. T.J. Menzies and P. Compton. Applications of Abduction: Hypothesis Testing of Neuroendocrinological Qualitative Compartmental Models. *Artificial Intelligence in Medicine*, 1997. To appear.
17. T.J. Menzies and A. Mahidadia. Ripple-Down Rationality: A Framework for Maintaining PSMs. In *Workshop on Problem-Solving Methods for Knowledge-based Systems, IJCAI '97, August 23.*, 1997.
18. T.J Menzies and R Spurret. How to Edit "it"; or a Black-Box Constraint Based Framework for User Interaction with Arbitrary Structures. In *Tools Pacific 12*, pages 213–224. Prentice Hall, 1993.
19. G.C. Murphy, D. Notkin, and E.S.C. Lan. An Empirical Study of Static Call Graph Extractors. Technical Report TR95-8-01, Department of Computer Science & Engineering, University of Washington, 1995.
20. A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.
21. A. Newell. Reflections on the Knowledge Level. *Artificial Intelligence*, 59:31–38, Feburary 1993.
22. T.W. Olle, J. Hagelstein, I.G. MacDonald, C. Rolland, H.K. Sol, F.J.M. Van Assche, and A.A. Verrijn-Stuart. *Information Systems Methodologies: A Framework for Understanding.* Addison-Wesley, 1991.
23. Persistence Software Inc., http://www.persistence.com/. *Persistence: A Relational Mapping and Caching Tool.*
24. A.D. Preece. Principles and Practice in Verifying Rule-based Systems. *The Knowledge Engineering Review*, 7:115–141, 2 1992.
25. J. Rumbaugh. Getting Started: Using Use Cases to Capture Requirements. *JOOP*,

pages 8–23, 1994.

26. B.J. Wielinga, A.T. Schreiber, and J.A. Breuker. KADS: a Modeling Approach to Knowledge Engineering. *Knowledge Acquisition*, 4:1–162, 1 1992.
27. N. Zlatareva. CTMS: A General Framework for Plausible Reasoning. *International Journal of Expert Systems*, 5:229–247, 3 1992.
28. N. Zlatareva. Distributed Verification and Automated Generation of Test Cases. In *IJCAI '93 workshop on Validation, Verification and Test of KBs Chambery, France*, pages 67–77, 1993.
29. N. Zlatereva and A. Preece. State of the Art in Automated Validation of Knowledge-Based Systems. *Expert Systems with Applications*, 7:151–167, 2 1994.

Some of the Menzies papers can be found at *http:// www.cse.unsw.edu.au/~timm/pub/ docs/papersonly.html.*

This article was processed using the LaTeX macro package with LLNCS style