A Generic Framework for Approximate Simulation in Commonsense Reasoning Systems

Benjamin Johnston, Mary-Anne Williams

University of Technology, Sydney
Faculty of Information Technology
johnston@it.uts.edu.au, mary-anne@it.uts.edu.au

Abstract

This paper introduces the *Slick* architecture and outlines how it may be applied to solve the well known Egg-Cracking Problem. In contrast to other solutions to this problem that are based on formal logics, the *Slick* architecture is based on general-purpose and low-resolution quantitative simulations. On this benchmark problem, the *Slick* architecture offers greater elaboration tolerance and allows for faster elicitation of more general axioms.

Introduction

In his influential Naïve Physics Manifesto papers, Patrick Hayes (1985) called for AI researchers to end their preoccupation with 'toy worlds' and attempt to build large scale formalisms. He observed that models of commonsense knowledge are orders of magnitude larger and more complex in size than any 'toy worlds', and that the construction of large models will require new idioms and development methodologies to manage the scale of the problem. Addressing this criticism, Morgenstern and Miller (2006) have published a web-page collecting a set of benchmark problems. One of these benchmark problems was contributed by Ernie Davis and is known as the 'egg cracking problem'. While the problem is far removed from any kind of commercial pressures, it nevertheless provides a useful benchmark in which sophisticated commonsense reasoning techniques may be proved. The problem is to characterize the following situation:

> A cook is cracking a raw egg against a glass bowl. Properly performed, the impact of the egg against the edge of the bowl will crack the eggshell in half. Holding the egg over the bowl, the cook will then separate the two halves of the shell with his fingers, enlarging the crack, and the contents of the egg will fall gently into the bowl. The end result is that the entire contents of the egg will be in the bowl, with the yolk unbroken, and that the two halves of the shell are held in the cook's fingers.

Various elaborations are also proposed that can be used to test a given solution; elaborations such as the question of what would happen if the egg is hard-boiled, if the egg is not from a chicken, if the bowl is upside down or made from different materials, or if the procedure is performed very fast or very slow.

The egg cracking problem is interesting because a successful characterization of the egg cracking problem requires a relatively sophisticated theory of naïve physics, of which rich elaborations make use of many commonsense phenomena such as position, shape, solids, liquids, viscosity, cracking, gravity, containment, expiration and even the social roles, capabilities and desires of the cook.

While the egg-cracking problem does not mandate a characterization based on formal logic, the three proposals in the literature (Morgenstern 2001; Shanahan 2004; Lifschitz 1998) have used action calculi. Other techniques of commonsense-reasoning might also be used, such as (though, not limited to) Cyc (Lenat *et al.* 1990), Open Mind Common Sense (Singh *et al.* 2004), qualitative reasoning or a combination of several techniques.

The objective of this paper is to introduce a novel approach to solving the egg cracking problem. We have created *Slick*, an architecture that allows for cost-effective construction of commonsense knowledge bases but still yields many of the benefits of logical methods. The *Slick* architecture is based on a generalized approach to constructing simulations: it is therefore based on a fundamentally non-logical and sub-symbolic method of deduction. *Slick*, however, is accessed via a symbolic layer and can therefore be meaningfully integrated with or as a component of a symbolic system—systems can be developed in a hybrid manner, playing to the strengths of both simulation and formal methods such as action calculi.

Methods of Commonsense Reasoning

Constructing an exhaustive formal model of the egg-cracking problem at every level of granularity is obviously beyond the capabilities of techniques (or human resources) available today. However, given the finite capabilities of a research laboratory (or a commercial environment) it is possible to sacrifice some of the depth, breadth or expressiveness in order to create a functionally useful system. In fact, many of the existing approaches in the literature can be seen to either emphasise depth and expressiveness or emphasise breadth: by focusing on rich representation it becomes costly to create broad knowledge bases, whereas techniques that are used to efficiently create broad knowledge bases tend to require shallow or informal representations.

Logical and Qualitative Methods. The usage of logics of action and of qualitative reasoning are largely inspired by Hayes (1985). These approaches are fraught with many difficulties: the underlying logics and formalisms offer little guidance for distinguishing superior representations from inferior representations; it is often difficult to validate the developed theories for inconsistencies; different knowledge engineers are likely to make different simplifying assumptions that increase the complexity of integrating multiple knowl-

edge bases; and finally, effective use of logics and qualitative methods typically requires a great deal of aptitude and education. It comes as no surprise that the logical approaches have enjoyed their greatest successes in domains where a small set of engineers can grasp the problem, and ensure that the formalization remains logically and philosophically consistent.

Semi-logical and Informal Methods. In constrast to purely logical methods, projects such as Cyc (Lenat *et al.* 1990) and the DARPA High-Performance Knowledge Bases (HPKB) (Pease *et al.* 2000) emphasise logic-*inspired* representations that support and embrace inconsistency so as to more naturally reflect the manner in which humans inconsistently perceive and reason about the world. At the extreme are projects such as Open Mind Common Sense that use very simple word associations and logical structures to provide very general forms of commonsense 'intelligence' (such as word associations) that can be constructed at low cost and give the appearance of intelligence without requiring deep reasoning capabilities (Singh *et al.* 2004). While these methods may be suitable for reasoning about general patterns of action, or possible tools and actions that may lead to a cracked or broken egg, such work is currently simply not intended for or capable of deep reasoning about specific scenarios.

Multi-paradigm Approaches. In *The Society of Mind*, Minsky (1986) presents the view that a mind consists of a society of interacting processes, for which no single representation suffices. In this vein, there have been ambitious proposals for creating powerful general purpose systems with commonsense intelligence based on diverse multi-paradigm architectures (McCarthy *et al.* 2002; Sowa 2002). However, these broad proposals are aimed at supporting very rich forms of commonsense reasoning, and are far beyond the intended scope of this work.

A compelling concrete example of the multi-paradigm approach is Mueller's (1998) ThoughtTreasure that combines factual logical knowledge and reasoning (similar to Cyc), grids (2D maps of stereotypical settings), simulated agent-based planning and procedural 'rules of thumb'.

Multi-paradigm architectures may ultimately be necessary to create rich commonsense reasoning capabilities, however this does not preclude the use of *Slick* or any other approaches to commonsense reasoning. Each such approach could be reasonably subsumed as a component providing a specific reasoning capability within a multi-paradigm architecture: in fact, the development of successful multi-paradigm architectures requires the creation of components (such as *Slick*) that provide powerful reasoning mechanisms.

Simulation

Aside from the simplistic agent-based simulations of plans used in ThoughtTreasure (Mueller, 1998), direct simulation as an approach to commonsense reasoning is not often mentioned in the literature. Simulation, however, is not an unusual approach to commonsense reasoning—qualitative reasoning often makes use of qualitative simulations, and deduction with formal methods such as action calculi could be considered to be a kind of simulation. However, it is interesting to observe that a great deal of extremely rich commonsense 'know-how' can be seen in the simulations that appear in modern computer animations and games. Realistic and computationally efficient models of solids, liquids, physical forces, natural environments, human needs and even relationships can be found in games and physics libraries (e.g., ODE (Smith 2006)), and a compelling argument could be made for incorporating these simulations as a component in an intelligent system. While we are unaware of any games that involve a task of cracking an egg; it is not hard to imagine that one might occur in some future game.

Morgenstern (2001) reflected on the difficulty of using logic to adequately describe the motion of a liquid as it flows from an egg to a bowl or the motion of a falling object. In contrast, under a simulation such effects follow directly from the Newtonian equations of motion, and the force of gravity. Since the intent of simulation is to accurately and directly emulate real world behaviors, the knowledge engineering process is conceptually simplified to the familiar task of implementing (typically in a procedural programming language) the fundamental commonsense laws of a problem domain. Furthermore, the representations used in implementing simulations are often highly conducive to 3D visualizations, making it easier to observe, refine and validate their output.

While commercial simulation libraries can produce realistic simulations when appropriately configured, they are unfortunately not designed with general purpose commonsense reasoning in mind. Simulation libraries are typically optimized for limited classes of physical simulation (such as rigid-body physics), and do not include flexible mechanisms for associating elements within a simulation with symbolic representations that may be used in the high-level meta-reasoning of an intelligent system. Nevertheless, the principle of using simulation as a form of commonsense reasoning is attractive because it offers a rich and natural approach to conceiving and expressing commonsense knowledge, and because it simplifies knowledge elicitation and knowledge maintenance. We have therefore developed *Slick* as a generic framework for simulating a wide range of commonsense phenomena: not just rigid-body physics, but liquids, materials and even abstract concepts such as human relationships and fatigue in a straightforward way.

Slick Overview

The *Slick* architecture is designed to be a component for commonsense reasoning within a larger intelligent system. Sophisticated reasoning and problem solving is beyond the scope of the Slick architecture: it is intended that the host architecture be responsible for planning, symbolic reasoning,

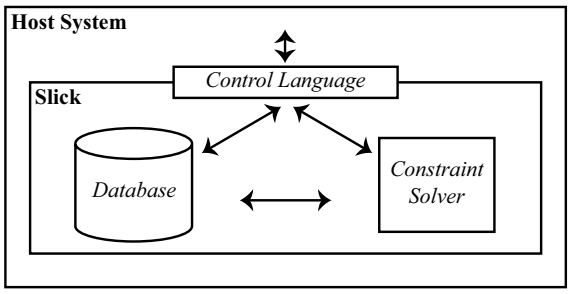

Figure 1: Slick Architecture

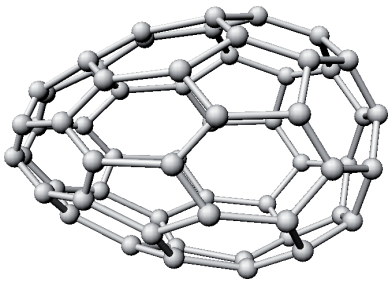

Figure 2: Modelling an egg with *entities* (balls) and *joins* (sticks) (Screenshot from Java3D visualization for Slick prototype)

and the resolution of high level goals and contextual information. The host system uses Slick to solve commonsense problems posed either symbolically, by way of constraints derived from sensors, or by some combination of the two. From a computational perspective, the Slick architecture is a simple constraint solver over multidimensional domains, however Slick is not presented here as a technical or algorithmic contribution: its benefits lie in the *structural architecture* that is motivated by the need to ground the sub-symbolic processes of a simulation to the symbolic representations of the host intelligent system or to the real world via sensors.

Slick consists of the following three subcomponents: a generic relaxation-based *constraint solver*; a *database* of generic objects and constraints for simulation; and a symbolic *control language* that is used for both instantiating objects from the database into the constraint solver and for testing constraints and alternative scenarios (See Figure 1).

Slick 'Constraint Solver'

In order to simulate a wide range of behavior, simulations are constructed from abstract structures that we refer to as *entities* and *joins*. These structures are similar to frames and represent the fundamental particles and relations of a simulation. *Entities* and *joins* do not have any implementation or 'active' character, their purpose is to give structure to a simulated environment, and to store the ongoing state of that simulation. Their naming is intentionally generic, as they may be given any interpretation by the knowledge engineer. For example, *entities* and *joins* may be used to denote people and social regard, atoms and intramolecular bonds, balls and sticks (as in ball-and-stick models) or even e-mails and discussion-threads.

In simulating physical environments, *entities* and *joins* have a natural interpretation as over-sized particles of matter and bonds that can be used to approximate the shape and behavior of an object (this particular physical interpretation corresponds to a 3D generalization of the 2D 'molecules' used by Gardin and Meltzer (1989)). For example, to represent an egg; 60 *entities* can be used to approximate the mass distribution of an egg and 90 *joins* to structure the egg's shape as per Figure 2. While each *entity* and *join* has an independent identity and existence within a simulation, individual *entities* and *joins* do not necessarily denote any particular real-world concept, it is rather a given set of *entities* and *joins* that as a coherent whole could be regarded as denoting an egg shell.

Entities and joins are similar to frames in that they have a set of *attributes* which are used to store the state of a simulation, however these *attributes* need not store a single unique value but can have a different value for each instant and scenario of a simulation. That is, a particular entity of an eggshell might have *attributes* for its Cartesian coordinates (x, y, z), temperature and color: the value of these *attributes* may vary for every instant in time. Some of these attributes are informational and necessarily constant over a simulation (such as the type and role of the entity or join) so are referred to as *labels*. In our simulations, we have assumed that mass and rigidity remain constant for a given entity and so have defined these using *labels*, however it is conceivable that in more sophisticated simulations these would be mutable *attributes* (e.g., rigidity could be made to vary depending on the temperature or age of the egg). In our naïve implementation, every value of every *attribute* at every instant is retained; however greater space efficiency can be achieved by saving only those values that are required.

Figure 3 provides an example of the *attributes* and corresponding values at a given instant, for a sample *entity* and a sample *join* from an egg-cracking simulation. *Labels* are indicated by bold type.

Values of attributes are assigned and updated in the simulation by way of *constraints*. These *constraints* are solved in an iterative process, calculating values for attributes over one or more discrete *simulation dimensions*. In our egg-cracking simulation there is just one *simulation dimension*, namely time, modelled as a series of discrete instants separated by short constant intervals. In other situations these simulation dimensions might be used to represent the three dimensions of space or more abstract concepts such as database records. Furthermore, while our naïve implementation uses equally spaced instants in time, this spacing could be adaptive so as to perform more accurate simulations at critical moments in the simulation (such as when objects are colliding).

A *constraint* is simply a parameterizeable function that calculates the value of an attribute in a given instant and scenario from its current values. In an physical simulation there are separate *constraints* for phenomena such as persistence, momentum, gravity, shape-holding rigidity, bond forces, liquid

```
Entity 34 at instant [time:1.35s, scenario:1]
  identifier: world/egg#1/shell/e/shape#4
  type: entity
  role: shape
  number: 4
  shape-rigidity: 0.4
  fragment-mass: 0.2
  color: cream
  temperature: 25°
  x: -0.744
  y: -0.3336788
  z: 1.5337192

Join 41 at instant [time: 1.35s, scenario:1]
  identifier: world/egg#1/shell/j/shape#3
  type: join
  entity-a: <entity 34>
  entity-b: <entity 35>        } the 'joined' entities
  role: shape
  number: 3
  spring-constant: 0.7
  snap-length: 1.01
  snapped: false
```

Figure 3: Example of an Entity and a Join

Constraint: Persistence
$\{$<x: x_t', y: y_t', z: z_t'>$\}$
where, $x_t' = x_{t-1}$
$y_t' = y_{t-1}$
$z_t' = z_{t-1}$

Constraint: Momentum
$\{$<x: x_t', y: y_t', z: z_t'>$\}$
where, $x_t' = x_t + (x_{t-1} - x_{t-2})$
$y_t' = y_t + (y_{t-1} - y_{t-2})$
$z_t' = z_t + (z_{t-1} - z_{t-2})$

Constraint: Gravity
$\{$<z: z_t'>$\}$
where, $z_t' = z_t + \frac{1}{2} g (t_{interval})^2$

Figure 4: Example Specifications of Constraints

flow, impermeability and the immobile floor. Each of these constraints is given a *ranking* which determines the order it is to be applied in the calculation of the value for an attribute. The ranking can be used to ensure that the 'persistence' constraint is applied first (corresponding to the default that things tend to stay as they are); followed by constraints such as gravity; then finally, hard constraints that the user should never see violated such as the impermeability and immobility of the floor.

Scenarios are used to specify multiple executions of similar but related situations. For example, an egg-cracking simulation can be set up using several *scenarios* each having the bowl in a different orientation: each *scenario* is completely isolated as though run in an entirely separate simulation.

While a *constraint* ordinarily returns a single value, other return-values are possible: a set of values, a distinguished 'inconsistent' token or a distinguished 'underspecified' token. If a single value is returned, then the relevant *attribute* is set appropriately. However, a set of values can be returned to indicate uncertainty or multiple outcomes, so that Slick will fork the current *scenario* into a corresponding set of new *scenarios* each identical except for the values of uncertain *attribute*. If the distinguished values are returned then slick will either mark the entire *scenario* as inconsistent (in the case of the 'inconsistent' token) or delay the evaluation of the *constraint* as long as possible (in the case of the 'underspecified' token).

An example of the *constraints* for persistence, momentum and gravity appear in Figure 4. Note that while the gravitational *constraint* only causes a fixed displacement at each time instant, it is the persistence and momentum constraints that cause these constant effects to accumulate and result in gravitational acceleration.

This approach to simulation has several major advantages:

1. **Ease of specification.** Constraints are analogous to point-wise axioms in a logical approach. The advantage is that there is a direct correspondence to the fundamental laws of Newtonian physics—macro-scale effects such as the nature of a crack and how it may increase in size are emergent behaviors that do not need to be explicitly specified.
2. **Constraints are reusable.** An implementation of mass-spring constraints for entities and joins can be reused for any solid: a ceramic bowl is subject to the same physical constraints as an egg shell; it merely has a different shape and rigidity.
3. **The frame problem is elegantly solved.** Persistence and friction limit the changes that are possible between one moment and the next, and also propagate the state of unchanged attributes.

Slick Database

While the general purpose simulation framework described above grants an intelligent system the ability to simulate a diverse variety of commonsense problems, it offers no mechanism for relating the primitives of the simulation to abstract symbols or for interpreting the outcomes of a simulation. The Slick *database* is used for associating symbols to the corresponding entities, joins and constraints that must be instantiated into the simulator. We have taken a pragmatic approach for the time being by allowing only simple hierarchical structures: the database is a forest of trees whose leaf nodes are *schema* for the instantiation of the simulation primitives, and whose internal nodes are labelled to provide a decomposition structure. For example, to instantiate an egg, the database is searched for a tree whose root node is labelled with the egg symbol. This tree will have internal child nodes with labels such as shell, yolk and white. The 'shell' branch of the tree will in turn have leaf nodes that are schema for the construction of 60 entities whose position, mass, color and brittleness are set to appropriate values for realistic simulation of an egg shell. More abstract concepts such as 'cracked' would be defined to accept a parameter during instantiation and are represented in the database as trees whose leaf nodes are only schemas for *constraints* (the symbol 'cracked', when instantiated for an egg shell, would create *constraints* that report the inconsistent value in any scenario where the egg is not cracked).

Slick Control Language

The Slick *control language* is very simple; it consists of operators to perform the following tasks:

1. Creation of new scenarios for simulation,
2. Instantiation, from symbols, of schema within the database,
3. Querying and manipulating trees in both the database and simulation scenarios (e.g., a chocolate egg could be created by grafting the chocolate texture onto an egg shell shape, and deleting the branches for the yolk and white),
4. Directly instantiating simulation primitives (entities, joins and constraints),
5. Setting and querying values of attributes in the simulation.

Our concrete implementation of Slick currently uses a simple stack-based language with a declarative style of usage both for populating the database and as the control language. The syntactic details of this language are unimportant as the control language could equivalently be implemented using, for example, XML or as an API for an existing language.

In practice, the control language is one of the most powerful benefits of Slick. The ability to interact with the Slick database and readily produce new and meaningful concept combinations (such as 'chocolate eggs') is an outcome that is difficult to achieve using formal logics. Yet, in Slick, these benefits are realized with just a small set of obvious operators for querying, deletion and grafting of trees.

Results

Without any standard quantitative method for evaluating the elaboration tolerance of solutions to the egg cracking problem, it is difficult to comprehensively evaluate Slick within the space constraints of this paper. However, we believe that Figure 5 speaks volumes: it depicts the internal state of a simple simulation in which an egg shell has cracked after being dropped onto a table. A crack can be observed in Figure 5 as broken joins along the sides of the shell. In our real-time visualization, this crack can be seen progressing along the edge of the shell, cracking in a manner consistent with our human intuitions. Most importantly, this simulation does not require the explicit *a priori* definition of abstract constraints such as a 'crack' because the cracking behavior simply emerges from the physical constraints that apply to the entities and joins.

In richer simulations that we have tested, Slick is able to correctly answer more difficult questions such as whether the egg or bowl remains unbroken and whether the bowl or table is 'wet'. While these abstract properties (such as 'cracked', 'broken' and 'wet') will emerge from the fundamental laws governing the simulation, it is still useful to create additional constraints that correspond to these abstract concepts so that the outcome of a simulation may be analyzed symbolically. For example, when 'cracked' is defined as having at least one join broken, the host intelligent system can test if an egg has been cracked simply by testing to see if the constraint holds.

Various elaborations and what-if-scenarios can be entertained by adjusting the parameters of the simulation. In fact, it is in the comparative ease by which elaborations are accommodated, that the benefit of Slick becomes most evident:

Speed and Force. By virtue of simulation, such elaborations present no difficulty. If there is insufficient force, the egg shell's rigidity will not be overcome, and conversely if there is too much force it may even exceed the rigidity and snap-length of the bowl (breaking the bowl).

Shape and Orientation. Again, once the shape and position has been configured in the simulation's initial conditions, such elaborations present no difficulty. For example, if the bowl is too small then the simulated egg yolk and white will simply overflow the bowl without any prior need to model concepts such as 'fullness'.

Texture. A wide range of solids, liquids and gasses can be simulated by adjusting parameters for rigidity, breaking points, viscosity and permeability. The different behaviors of chocolate eggs, coconuts, hard-boiled eggs, duck eggs, soccer balls and of paper, clay and ceramic bowls can all be accommodated (to a usable degree of approximation) simply by appropriate configuration of these material parameters.

Detail. Finer-grained approximations require further specification, but this is readily done by elaborating the initial conditions (rather than the laws of simulation). For example, in a real chicken egg there is a membrane between the white and the shell that holds a slightly-cracked egg together: the simulation could be enriched by inserting such a membrane (a pliable surface) into the initial conditions of the egg.

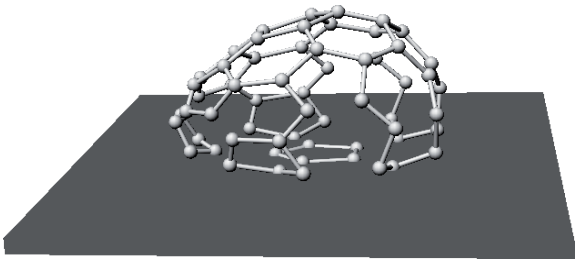

Figure 5: Simulation of an egg-shell cracking after being dropped onto a table

Scope. Additional constraints would be required to address elaborations far beyond the problem's original scope. For example, additional constraints would be required for 'rotting' and 'cooking' if the action is to be performed over months or at high temperatures. However, it is important to note that the addition of these constraints does not require revision of existing constraints: gravity and momentum continues to work on all matter. 'Cooking' would be simulated just by adding a constraint to allow a subset of bonds to 'harden' after a certain temperature has been reached.

The Slick framework is therefore able to evaluate an extremely rich range of questions and elaborations without exhibiting the same brittleness evident in the elaborations of solutions using action calculi as proposed by Morgenstern (2001), Shanahan (2004) and Lifschitz (1998).

An implementation of the Slick framework is a relatively complex software product that requires an up-front investment of developer time, but once implemented provides an extremely flexible basis for experimenting with variants on the egg cracking problem: for example, constraints for gravity, friction and mass-spring kinetics are virtually all that is required for simulation of most commonsense situations involving solids. While we have not conducted a formal experiment to quantify the cost savings of the Slick architecture, we can report our personal experiences with implementing constraints in the Slick architecture:

- Identification of necessary constraints was trivial as missing constraints became painfully obvious when watching the simulations (for example, the need for friction was evident when an egg slid across the floor 'forever').
- Specification of these constraints was trivial—suitable specifications can be found in any high-school physics text-book.
- Implementation of these constraints is routine and straightforward: the mathematics required to convert commonsense-based physical laws into Java code in the Slick framework is possessed by most high-school graduates.
- Validation was significantly simplified because the visualizations could be used to gauge whether the model accurately reflects intuition.

- Populating the database with 3D objects required minimal effort—we used a chemist's tool originally designed to create 3D wire-frame models of molecules.
- The most tedious aspect of using Slick was tuning parameters so that, for example, the rigidity and snap-length on the egg shell is such that it cracks as expected when dropped from a suitable height. We anticipate that this task could be automated in future by a simple machine learning or optimization algorithm.

These experiences are in contrast to those we have observed when we have attempted to create and update equivalently rich formalizations of similar problems using action calculi: our impression is that within logical formalisms the tasks of identification, specification and implementation/validation are orders of magnitude more difficult, time consuming and intellectually challenging than with Slick. Furthermore, with appropriate tools that integrate modelling with automatic parameter optimization, it should be possible to further simplify the knowledge engineering process with Slick. In fact, we eventually intend to populate the database semi-automatically based on experimentation and sensor readings collected by a robot from its situated environment (i.e., perceiving the 3D shape and fitting parameters automatically).

Limitations and Future work

While the egg cracking problem is useful for exposition and comparison with other techniques, in future we intend to test Slick with more abstract applications of simulation such as the creation of commonsense models of business processes, market behavior, professional human relationships and schedule feasibility (both from a physical perspective, but also from the perspective of modelling factors such as human fatigue and productivity). The Slick architecture is ideal for domains that are governed largely by a set of fundamental laws from which most observable behavior emerges. This is clearly the case with simple physical scenarios that are governed by Newton's laws of motion, however useful and usable approximations can be found in other domains—for example, simple game theoretic models may be used for modelling economic and social behavior.

We have mentioned that Slick can be readily integrated with other approaches, including logics of action: in future we intend to explore this in greater depth. A naïve integration is possible by exposing Slick to a theorem prover as an 'external' predicate that is invoked as isolated simulations. The Slick database makes it easy to instantiate simulations from symbolic references, so predicates such as ***can-crack***(`chocolate-egg`, `bowl`, 3N) can be readily exposed to a logical theorem prover. Deeper integration may be possible with, theorem provers based on proof methods such as analytic tableaux or SLD resolution, by exploiting the similarity between *scenarios* and branches of a proof tree or tableau. For example, a theorem prover encountering ***drop-when***(0 seconds, `egg`) would configure Slick's initial conditions, while ***is-cracked***(3seconds, `egg`) would subsequently invoke Slick to evaluate the outcome. In a language such as Prolog, this integration could be done in a similar way to CHR (Frühwirth 1998).

Conclusion

Slick is a pragmatic mechanism for commonsense reasoning. Slick can provide a useful facility to an intelligent system for performing efficient and cost-effective commonsense reasoning about a diverse range of concepts:

- Slick simplifies and reduces the cost of knowledge engineering (and knowledge maintenance) in domains that are principally governed by a set of fundamental laws,
- Slick's commonsense 'theories' can be readily validated using tools such as 3D visualizations,
- Slick is readily implemented and computationally efficient (and will always terminate),
- Slick's database allows internal representations to be grounded to high level symbols or sensor input,
- Slick's database allows for elegant integration with other formalisms (such as action calculi), allowing for hybrid reasoning that draws on the strengths of each.

References

Frühwirth, T. 1998, 'Theory and Practice of Constraint Handling Rules', *J. of Logic Programming*, vol. 37, no. 1, pp. 95-138.

Gardin, G. & Meltzer, B. 1989, 'Analogical representations of naïve physics', *Artificial Intelligence*, vol. 38, no. 2, pp. 139-159.

Hayes, P.J. 1985, 'The second naïve physics manifesto', in J.R. Hobbs and R.C. Moore (eds), *Formal Theories of the Commonsense World*, Ablex, pp. 1-36.

Lenat, D.B., Guha, R.V., Pittman, K., Pratt, D. & Shepherd, M. 1990, 'Cyc - toward Programs with Common-Sense', *Communications of the ACM*, vol. 33, no. 8, pp. 30-49.

Lifschitz, V. 1998, 'Cracking an Egg: An Exercise in Commonsense Reasoning', Presented at the 4th Symposium on Logical Formalizations of Commonsense Reasoning.

McCarthy, J., Minsky, M., Sloman, A., Gong, L., Lau, T., Morgenstern, L., Mueller, E.T., Riecken, D., Singh, M. and Singh, P. 2002, 'An architecture of diversity for commonsense reasoning', *IBM Systems Journal*, vol. 41, no. 3, pp. 530-539.

Minsky, M. 1986, *The Society of Mind*, Simon & Schuster, NY.

Mueller, E.T. 1998, *Natural language processing with ThoughtTreasure*, Signiform, New York, <http://www.signiform.com/tt/book/>.

Mueller, E.T. 2000, 'A calendar with common sense', *The 2000 International Conference on Intelligent User Interfaces*, ACM Press, New York, pp. 198-201.

Morgenstern, L. 2001, 'Mid-Sized Axiomatizations of Common-Sense Problems: A Case Study in Egg-Cracking', *Studia Logica*, vol. 67, no. 3, pp. 333-384.

Morgenstern, L. and Miller, R. 2006, *The Commonsense Problem Page*, <http://www-formal.stanford.edu/leora/commonsense/>.

Pease, A., Chaudhri, V., Lehmann, F. & Farquhar, A. 2000, 'Practical Knowledge Representation and the DARPA High Performance Knowledge Bases Project', *Principles of Knowledge Representation and Reasoning*, pp. 717-724.

Shanahan, M. 2004, 'An attempt to formalize a non-trivial benchmark problem in common sense reasoning', *Artificial Intelligence*, vol. 153, no. 1-2, pp. 141-165.

Singh, P., Barry, B. and Liu, H. 2004, 'Teaching machines about everyday life', *BT Technology*, vol. 22, no. 4, pp. 227-240.

Smith, R. 2006, Open Dynamics Engine, <http://www.ode.org/>.

Sowa, J.F. 2002, 'Architectures for intelligent systems', *IBM Systems Journal*, vol. 41, no. 3, pp. 331-349.