Building abstractions

Alan Bundy
Dept. of Artificial Intelligence, Edinburgh

Fausto Giunchiglia
IRST, Trento

Toby Walsh
Dept. of Artificial Intelligence, Edinburgh

Abstract

The use of abstraction has been largely informal. As a consequence, it has often been difficult to see how or why a particular abstraction works. This paper attempts to help correct this trend by presenting a formal *theory of abstraction*. We use this theory to characterise the different types of abstraction that can be built; the different classes of abstractions we identify capture the majority of abstractions of which we are aware. We end by proposing a method for automatically building one very common type of abstraction, that used in Abstrips; our proposal is motivated by consideration of the various formal properties that such a method should possess.

1 Introduction

We see abstraction as the mapping of one representation of a problem, the "*ground*" representation onto a new representation, the "*abstract*" representation. The overall aim might be to improve the efficiency of reasoning; for example, an abstract solution might be used to guide the solution to the ground problem. Or it might be to increase the number of facts derivable; for example, a learning system might construct abstract versions of rules that are learnt so that they can fire in new situations.

Unfortunately the use of abstraction has in general lacked sound and theoretical foundations. This has led to many problems. For example, abstractions have in general been constructed by hand; without a comprehensive theory of abstraction, it is very difficult to determine automatically what to abstract for a given problem representation. Additionally there has been a lack of understanding of why abstraction works, of the different ways abstraction can be used, and of the relationships between the different types of abstractions. These factors have greatly restricted the usefulness of abstraction. The aim of this paper is: (1) to describe a *theory of abstraction*; (2) to use this theory to identify the different types of abstractions that can be built; (3) to look in detail at how we can automatically build one (rather famous) type of abstraction.

2 A theory of abstraction

We begin by outlining the formal framework presented in [**?**, **?**]. Since we see abstraction as a mapping between problem representations, we need a formal method for describing problem representations. One very general method is with a **formal system**, Σ.

> **Definition 1 (Formal System)** *: A* **formal system***, Σ is a pair, $\langle \Lambda, TH(\Sigma) \rangle$ where Λ is the language, $TH(\Sigma)$ is the set of theorems and $TH(\Sigma) \subset \Lambda$.*

The language, Λ provides the basic tools for writing down formulae. Usually the language is defined by an alphabet, the set of (well formed) terms and the set of well formed formulae (wffs from now on). To simplify matters, we just take it to be a set of well formed formulae (leaving the alphabet, and the set of well formed terms implicit). The set of theorems, $TH(\Sigma)$ is the set of statements which are taken to be true. Thus $TH(\Sigma) \subset \Lambda$. In fact, we nearly always deal with **axiomatic** formal system, in which the theorems are given by the transitive closure of a set of inference rules applied to a set of axioms.

Since we view abstraction as a mapping between problem representations, we formally define an **abstraction** as a mapping from one formal system to another, more abstract one:

Definition 2 (Abstraction) *: An* **abstraction***, $f : \Sigma_1 \Rightarrow \Sigma_2$ is a pair of formal systems $\langle \Sigma_1, \Sigma_2 \rangle$ with languages Λ_1 and Λ_2 respectively, and an effective total function $f : \Lambda_1 \rightarrow \Lambda_2$.*

Following historical convention, Σ_1 is called the **ground space** and Σ_2 the **abstract space**. f, the function that maps wffs from Λ_1 onto wffs in Λ_2 is called the **mapping function**. This is a very general definition of abstraction. Indeed it gives no guarantee that the abstraction is useful; what is true in the abstract space may bear no relationship to what is true in the ground space. An interesting and general restriction is therefore to abstractions that preserve provability in some way. In [**?**] we identified the very important class of **truthful abstractions** for which any theorem of the ground space is mapped onto a theorem of the abstract space.

Definition 3 (Truthfulness) *An abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ is* **truthful** *iff, if $\varphi \in TH(\Sigma_1)$ then $f(\varphi) \in TH(\Sigma_2)$.*

Truthful abstractions are complete (the abstraction of *any* ground theorem is a theorem in the abstract space) but may not be correct (there may be theorems in the abstract space which do not correspond to any theorem in the ground space). A large number of abstractions proposed in the past are truthful. For example, Hobbs' granularity [**?**] can be described as a truthful abstraction. In [**?**], we also considered the dual class of abstractions which are correct but not complete. In any sufficiently interesting theory, undecidability ensures that is not possible to have both completeness and correctness. Preserving provability is actually only a very weak property to demand – in [**?**] we also consider stronger properties like the preservation of the *structure* of proofs; abstract proofs can then be used to guide the search for ground proofs. Of course, the preservation of the structure of proofs usually entails some type of preservation of provability. What is perhaps most important is that we are able to prove some very interesting results just from the weak assumption that provability is preserved.

3 Types of abstractions

Many different (truthful) abstractions have been proposed. Unfortunately, there have been few attempts at classifying the different types of abstractions.

A first step therefore in building abstractions automatically is to identify the different ways we can abstract a problem representation. This section attempts to provide such a classification.

With axiomatic formal systems, the abstract space often uses the same inference rules (or a subset of them) as the ground space. This makes implementation easier, and allows for the use of hierarchies of abstractions. Additionally, the axioms of the abstract space are frequently given by applying the mapping function to the axioms of the ground space. From now on we will restrict our attention to such abstractions. We can therefore classify abstractions simply by the properties of their mapping function.

An important class of abstractions – and one that formalises nearly *all previous work in abstraction* – is the class of **atomic abstractions**; that is, abstractions which map the atomic formulae, and not their logical structure. For example, $f(p \;\&\; q)$ is often the same as $f(p) \;\&\; f(q)$. Note that we have overloaded the symbol "&" since it is used to represent both conjunction in the ground and in the abstract languages. Strictly speaking we should distinguish between these two uses. However, since it is given the same meaning in both the ground and the abstract spaces, we introduce no ambiguity by this overloading.

Abstracting the logic is very dangerous as the consistency of a formal system is usually very finely balanced. In general, the logic itself is well behaved and it is the theory that needs to be simplified. Additionally, an abstraction cannot change the logical structure of formulae too radically if it is to preserve provability (or, more strongly, the structure of proofs). [**?**] provides a longer discussion of these issues.

The recursive definition of atomic formulae suggests a systematic classification of atomic abstractions:

1. **term** abstractions where we map the terms. Further subdivided into:
   - **domain** abstractions where we map the constants together;
   - **function** abstractions where we map the function names together, or reduce their arity;
2. **predicate** abstractions where we map the predicate names together, reduce their arity or (a special case) map the whole atomic formula onto $\top/\bot$.

This classification describes *all* atomic abstractions that can be decomposed into mappings on the individual parts of the atomic formulae. All these different types of abstractions can be found in the literature. Most, in fact, can be found in the work of Plaisted [?, ?]. Both Hobbs [?] and Imielinski [?] have proposed domain abstractions. Plaisted [?, ?] has used function abstractions on some interesting problems. Tenenberg [?] has looked at predicate abstractions which collapse predicate names together. And the abstraction used in Abstrips [?] can be seen as predicate abstraction that maps certain preconditions onto $\top$.

This section has identified some of the major types of abstraction. Each maps a different part of the atomic formula (*eg.* the predicate names, the constants, ...) or alternatively the whole atomic formula. It will depend on the problem, and the choice of representation exactly what it is worthwhile mapping together. The next section considers this problem in detail for one particular class of abstraction.

4 Building Abstrips abstractions

An important class of atomic abstractions is the class of predicate abstractions which map some of the atomic formulae onto $\top$. Abstrips [?] used such abstractions to simplify its Strips-like planning domain. This section explores how to build such abstractions automatically.

The problem domain is determined by a set of operators. We assume each has a set of preconditions, and a conclusion described by an implication in a situation calculus. Abstrips constructed a hierarchy of abstraction levels by mapping preconditions onto true, $\top$ according to their *criticality*; this is a measure of how difficult the precondition is to achieve. Those preconditions which it is impossible to change are given the highest criticality. Those preconditions which it is very hard to change are given the next highest criticality. And those preconditions which it is very easy to change are given the lowest criticality. This notion of criticality is, in fact, very general and need not be restricted just to preconditions – for instance, when using a domain abstraction, we can define the criticality of the name of a constant.

Abstrips constructed plans in a hierarchy of abstract spaces, each of which ignored the preconditions below a certain criticality. We propose a method to calculate such criticalities *automatically* which improves upon the *semi-*

automatic method used in Abstrips. As in the original method used by Abstrips, the criticality of a precondition is independent of its arguments; that is, it is a measure of how difficult *on average* it is to achieve a precondition. To this end, we ignore the arguments to the preconditions.

We envisage an iterative process, whereby we assign the preconditions some default starting value and then by a series of iterations calculate their criticality. We shall represent by $C(Ops, p, n)$ the criticality of the predicate, p at the n-th iteration given a set of operators, Ops. Note that, unlike Abstrips, we assign a criticality to every predicate, irrespective of whether it appears as a precondition to an operator or not. We will not be particularly interested in the absolute values $C(Ops, p, n)$ returns, just their relative ordering. We motivate the choice of our criticality assignment function by identifying some desirable properties it should possess. The first is that the assignment function should converge to an unique answer.

Property 1 (Convergence)

$$\lim_{n\to\infty} C(Ops, p, n) = a_p$$

We might sometimes allow a_p to be infinite (in which case, we also want to be able to know that the criticality can be calculated with arbitrary precision). Second, we would like the assignment function to be **fair**. Initially, we have no information to distinguish between the different predicates. The assignment function should not therefore discriminate between them. Every predicate should be given a uniform starting value, a_0.

Property 2 (Fairness)

$$C(Ops, p, 0) = a_0$$

Third, we want the function to be **monotonic**. Given a set of operators, if we add another operator with conclusion, p then as p is easier to satisfy (we have another way of proving it) the criticality of p should go down (or, at least, stay the same). And if we add a precondition, q to an operator with conclusion p then as p is more difficult to satisfy (we have to prove another precondition) the criticality of p should go up (or, at least, stay the same).

Property 3 (Monotonicity)

$$C(Ops \cup \{q \to p\}, p, n) \leq C(Ops, p, n)$$

$$C(Ops \cup \{(q\&r) \to p\}, p, n) \geq C(Ops \cup \{r \to p\}, p, n)$$

There are other properties, like **selectivity** (the assignment function should return a range of values), which are less precise in their definition and whose truth may depend very precisely upon the given operators. This is, by no means, an exhaustive list of properties that a criticality assignment function should possess; for example, we might also want the function to respect **impossibility** – those preconditions which cannot be changed should be given the largest criticality. Nevertheless, we would argue that these are all *necessary* properties.

5 A solution

We propose a criticality assignment function that satisfies all the properties identified in the last section. It is based upon interpreting $C(Ops, p, n)$ as the difficulty of finding a proof of p of up to depth n. $C(Ops, p, n)$ is defined recursively as a function of the difficulty of finding a proof of up to depth $n-1$ plus the difficulty of finding a proof of *exactly* depth n (which we represent by $D(Ops, p, n)$). We combine these difficulties by analogy to the calculation of parallel resistance. With two resistors in parallel, the current can go through either the first resistor or the second, thereby reducing the total resistance. Similarly, we can find either a proof of up to depth $n-1$, or we can find a proof of exactly depth n. Thus the difficulty of finding a proof of up to depth n is the parallel sum of the difficulties of finding a proof of up to depth $n-1$ and of finding a proof of exactly depth n (equation **??**). This method of combining difficulties guarantees the monotonicity property. The starting values for the calculation of criticalities are given by equation **??**; this equation gauarantees the *fairness* of our solution.

$$C(Ops, p, 0) = a_0 \tag{1}$$

$$\frac{1}{C(Ops, p, n)} = \frac{1}{C(Ops, p, n-1)} + \frac{1}{D(Ops, p, n)} \tag{2}$$

This still leaves us to decide how to calculate $D(Ops, p, n)$, the difficulty of finding a proof of exactly depth n. This is also defined recursively.

The boundary conditions are easy. $D(Ops, p, 0)$ is the difficulty of finding a proof of depth 0; this must equal $C(Ops, p, 0)$ (equation **??**). The other boundary condition is when no operator has p as a conclusion; since it will be impossible to change this predicate, we set this difficulty to ∞ (equation **??**).

The step equations are more complicated. The difficulty of finding a proof of p of exactly depth n, that is $D(Ops, p, n)$ is the parallel sum of the difficulties of finding a proof of depth n that ends with an operator with p as conclusion (equation **??**). $D(Ops, q \rightarrow p, n)$ represents the difficulty of finding a proof of depth n that ends with the application of the operator, $q \rightarrow p$. Note that q itself might be a conjunction of preconditions, p_i. Finally the difficulty of ending a proof of depth n with the operator, $\wedge p_i \rightarrow p$, that is $D(Ops, \wedge p_i \rightarrow p, n)$ is at least as difficult as finding a proof of depth $n-1$ of the most difficult precondition to that operator, that is $\max\{D(Ops, q_i, n-1)\}$ (equation **??**). In calculating the difficulty of proving the preconditions to an operator, we have made the simplifying assumption that the difficulty is dominated by the most difficult precondition; a more thorough analysis would also consider the difficulties of proving the less difficult preconditions. However, such a calculation could be very expensive; taking the maximum should give a good lower bound.

$$D(Ops, p, 0) = a_0 \tag{3}$$
$$D(Ops, r, m) = \infty \tag{4}$$
$$\frac{1}{D(Ops, p, n)} = \sum_{q \rightarrow p \in Ops} \frac{1}{D(Ops, q \rightarrow p, n)} \tag{5}$$
$$D(Ops, \wedge q_i \rightarrow p, n) = \max\{D(Ops, q_i, n-1)\} \tag{6}$$

where Ops contains no operator with r as conclusion, and $m > 0$

6 A worked example

We have tested this solution on some examples, taken from both planning domains (*eg.* the original Abstrips operators [**?**]) and from theorem proving domains (*eg.* the operators devised by Green for the famous Monkey and Bananas problem [**?**]). For illustration, we give a table containing the calculation of criticalities for the Monkey and Bananas problem. The operators for this problem are given in Appendix **??**.

$C\ /a_0$	$n = 0$	1	2	3	∞
at	1.00	0.25	0.25	0.25	0.25
has	1.00	0.50	0.33	0.25	0.25
reachable	1.00	0.50	0.33	0.33	0.33
on	1.00	0.50	0.50	0.50	0.50
movable	1.00	1.00	1.00	1.00	1.00
empty	1.00	1.00	1.00	1.00	1.00
climbable	1.00	1.00	1.00	1.00	1.00

Our criticality assignment function converges to an answer rapidly. Indeed, by the third iteration it reaches its final values. The relative order of criticalities calculated by our method is almost identical to that assigned by the original semi-automatic method used in Abstrips. The method used in Abstrips also needed an initial partial order on the preconditions. For the Monkey and the Bananas problem, we assume the partial ordering $\{(climbable, movable, empty),\ (on, reachable, has, at)\}$. The first set of preconditions represent type preconditions that cannot be changed, whilst the second set are all conclusions of operators (and so can be changed). The only difference between the criticalities calculated by our method and those calculated by the method used in Abstrips is that the latter method gives *at* a lower criticality than *has*. However, this difference seems more a criticism of the need to supply a partial order as a different partial order would produce different criticality assignments.

7 Conclusions

We have presented a powerful *theory of abstraction*. We have used this theory to identify the different ways we can abstract a new problem. We then concentrated on one particular type of abstraction, that used in Abstrips. We explored how to build such abstractions automatically, motivating our proposed method by defining the formal properties we want it to possess.

Acknowledgements

Alan Bundy is supported by a SERC Senior Fellowship. Fausto Giunchiglia started this work at the Department of Artificial Intelligence at Edinburgh supported by SERC grant GR/E/4459.8. He is now supported at IRST by the Istituto Trentino di Cultura. Toby Walsh is supported by a SERC studentship. All the members of the Mathematical Reasoning group in Edinburgh and the Mechanized Reasoning group in Trento are thanked for their many contributions to this work.

Appendix

A Monkey and Bananas' Operators

$$
\begin{aligned}
at(z, x1, s) \wedge movable(z) \wedge empty(x2, s) &\rightarrow at(Monkey, x2, m(Monkey, z, x2, s)) \\
at(z, x1, s) \wedge movable(z) \wedge empty(x2, s) &\rightarrow at(z, x2, m(Monkey, z, x2, s)) \\
at(z, x, s) \wedge climbable(y, z, s) &\rightarrow at(z, x, c(y, z, s)) \\
at(z, x, s) \wedge climbable(y, z, s) &\rightarrow on(y, z, c(y, z, s)) \\
at(Box, Under, s) \wedge on(Monkey, Box, s) &\rightarrow reachable(Monkey, Bananas, s) \\
reachable(z, x, s) &\rightarrow has(z, x, r(z, x, s))
\end{aligned}
$$