Cohesion, coupling and the meta-theory of actions

Andreas Herzig and **Ivan Varzinczak**[*]
IRIT – Université Paul Sabatier
118 route de Narbonne – 31062 Toulouse Cedex 04 France
e-mail: {herzig,ivan}@irit.fr

Abstract

In this work we recast some design principles commonly used in software engineering and adapt them to the design and analysis of domain descriptions in reasoning about actions. We show how the informal requirements of cohesion and coupling can be turned into consistency tests of several different arrangements of modules. This gives us new criteria for domain description evaluation and clarifies the link between software and knowledge engineering in what concerns the meta-theory of actions.

1 Introduction

Among the principles of the object-oriented paradigm are the following:

1. Work with modules (or components, functions, etc.).
2. Minimize interactions between such modules.
3. Organize the modules into well-defined layers to help minimize interactions. The goal is to have components of one layer using only components from immediate neighbors, wherever possible.
4. Anticipate what kind of extensions or modifications might be made in the future, and support this at design time so that one can extend the system with minimal disruption later.

There seems to be an agreement that such principles for object-oriented programming or design are the same as for knowledge representation. To witness, the design of domain descriptions in reasoning about actions has much more in common with software engineering than one might think: in the same way as for software projects, one can talk about consistency, evolution and correctness of domain descriptions.

All the principles above can be applied to the design of domain descriptions, too. We argue that a good domain description should be one whose consistency check and maintenance complexities are minimized, so that any further modification is localized, with a bounded scope.

[*]Supported by a fellowship from the government of the FEDERATIVE REPUBLIC OF BRAZIL. Grant: CAPES BEX 1389/01-7.

With this in mind, one can see the specification of domain descriptions as a task similar to project development in software engineering: Item 4 above is what has been called *elaboration tolerance* [McCarthy, 1988]. In this way a representation is elaboration tolerant to the extent that the effort required to add new information (a new action or effect) to the representation is proportional to the complexity of that information [Shanahan, 1997]. Items 1, 2 and 3 reflect the concept of *modularity*, which means that different modules have no elements in common. Such a notion of modularity is going to lead us along the present work.

This paper is an elaboration of the results we have presented in [Herzig and Varzinczak, 2004]. Here we pursue the following plan: in Section 2 we recall some important concepts from software engineering; after discussing the ontology of dynamic domains (Section 3) we apply the concepts of Section 2 to the design of domain descriptions (Sections 4 and 5) making a step towards formal criteria for domain description evaluation. In Section 6 we present the main results that follow from our approach, and before concluding we address related work found in the literature on this subject.

2 Some principles of software engineering

One of the first steps in software development is that of *abstraction*. Abstraction consists mainly in rendering lower-level details invisible to upper levels in order to facilitate the understanding and design of complex systems. As an example, a specification of a data or knowledge base query does not need to take into account the algorithmic process that will be carried out in order to answer the query.

In parallel to abstraction, one of the most important guidelines in project design is that of *modularity*: dividing the software into modules, based on their functionality or on the similarity of the information they handle. This means that instead of having a “jack of all trades” program, it is preferable to split it up into specialized subprograms. For instance, a program made of a module for querying a database and a module for checking its integrity is more modular than a single module that does these two tasks at the same time.

Among the major benefits of modular systems are reusability, scalability and better management of complexity.

There is more than one way to split up a program. One of the most used techniques is that of forcing functional independence of its modules. One ensures functional independence in a project by defining modules with only one purpose and “aversion” to excessive interaction with other modules [Pressman, 1992].

Among the criteria commonly used for evaluating functional independence of modules (and thus how modular a piece of software is) are the informal notions of cohesion and coupling.

Cohesion is how closely related pieces of a single component are to each other. A module is cohesive when at the high level of abstraction it does only a single task. The more a module is focused on a precise goal the more it is cohesive.

A highly cohesive module will be simpler to understand, having to do only a single task, while a lowly cohesive module, performing so many tasks, will be difficult to understand.

It is difficult to reuse a task-overloaded module, while a highly cohesive module is simpler to reuse and to extend.

Coupling is the interdependency between a method and the environment (other methods, objects and classes). Low coupling means to keep dependencies (communication, information sharing) between components at a minimum.

A design that has low coupling is more amenable to change, since it reduces the probability of changes cascading and affecting a larger part of the system.

Unanimously in object-oriented development, the best way to design a software is to have low coupling and high cohesion. We sum this up in two informal design principles:

P1. **Maximal cohesion**: *Every module should be conceived in such a way that it is maximally cohesive.*

P2. **Minimal coupling**: *All modules should be conceived in such a way that they minimize coupling.*

3 Natural modules in domain descriptions

Like in object-oriented programming, in describing a domain different entities should be separated in different modules. Moreover, each module should be conceived in such a way that it has no direct access to the contents of the others. In reasoning about actions, accessing a module means using it to perform reasoning tasks like prediction, postdiction, planning and others. This amounts to using its logical formulas in inferences. In this section we establish the ontology of domain descriptions and present the way we arrange in different modules the axioms commonly used to describe them.

Every domain description contains a representation of action effects. We call *effect laws* formulas relating an action to its effects. Statements of conditions under which an action cannot be executed are called *inexecutability laws*. *Executability laws* in turn stipulate the context where an action is guaranteed to be executable. Finally, *state constraints* are formulas that do not mention actions and express constraints that must hold in every possible state. These are our four ingredients that we introduce more formally in the sequel.

If we think of a domain description as a software application, we can imagine its organization in an object-oriented view and attempt to have a kind of class diagram for it. This is illustrated by Figure 1, where we can see the relationship among the different types of entities.

A domain description consists of a description of effects of actions, their non-effects, executabilities, inexecutabilities and also state constraints that do not depend on any particular action.

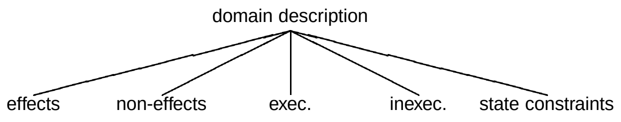

Figure 1: “Class diagram” of modules in designing domain descriptions. Edges represent *has-a* relations.

Among the effects of actions, we can distinguish *direct effects* and *indirect effects* (ramifications).

Non-effects of actions are related with the frame problem [McCarthy and Hayes, 1969], and indirect effects with the ramification problem [Finger, 1987]. In this work we abstract from these problems and assume we have a consequence relation powerful enough to derive the intended conclusions. We suppose given a ‘doped’ consequence relation $\mid\!\approx$, which encapsulates some traditional approach in the literature (e.g., [Schubert, 1990; Lin, 1995; McCain and Turner, 1995]), with which all intended frame axioms and indirect effects can be derived, and we use it henceforth. As examples we have

$$\{loaded(s)\} \mid\!\approx loaded(do(wait, s))$$

(i.e., waiting does not change the status of *loaded*) and

$$\left\{ \begin{array}{c} walking(S_0), \\ walking(s) \rightarrow alive(s), \\ \neg alive(do(shoot, s)) \end{array} \right\} \mid\!\approx \neg walking(do(shoot, S_0))$$

Hence shooting has the indirect effect that the victim will no longer be walking.

We use small letters to denote variables, and capital letters to denote constant symbols. Free variables are supposed to be universally quantified.

To sum it up, our main concern here will be with direct effects (henceforth effects), inexecutabilities, executabilities and state constraints. We introduce this in what follows.

Effect laws Logical frameworks for reasoning about actions contain expressions linking actions and their effects. We suppose that such effects might be conditional, and thus get a third component of such axioms. An *effect law for* action a is of the form

$$Poss(a, s) \rightarrow (\Phi(s) \rightarrow \Psi(do(a, s)))$$

where $\Phi(s)$ is a *simple state formula about situation* s, and $\Psi(do(a, s))$ is a *simple state formula about situation* $do(a, s)$.

A simple state formula about a situation term t contains no *Poss*-predicate and no situation terms other than t [Lin, 1995].

An example of effect law is

$Poss(shoot, s) \rightarrow (loaded(s) \rightarrow \neg alive(do(shoot, s)))$,

saying that whenever *shoot* is executable and the gun is loaded then after shooting the turkey is dead. Another one is $Poss(tease, s) \rightarrow walking(do(tease, s))$: the result of teasing is that the turkey starts walking.

Inexecutability laws The design of domain descriptions must also provide a way to express qualifications of actions, i.e., conditions under which an action cannot be executed at all. An *inexecutability law for* action a is of the form

$$\Phi(s) \rightarrow \neg \textit{Poss}(a, s)$$

where $\Phi(s)$ is a simple state formula about s.

For example, $\neg \textit{HasGun}(s) \rightarrow \neg \textit{Poss}(\textit{shoot}, s)$ states that *shoot* cannot be executed if the agent has no gun.

State constraints (alias domain constraints) Frameworks allowing for indirect effects make use of formulas that link invariant propositions about the world. Such formulas characterize the set of possible states. A *state constraint* is a simple state formula about the situation term s that is consistent. An example is $\textit{walking}(s) \rightarrow \textit{alive}(s)$, saying that if a turkey is walking, then it must be alive [Thielscher, 1995].

Executability laws With only state constraints and effect laws one cannot guarantee that action *shoot* is executable if the agent has a gun. An *executability law for* action a is of the form $\Phi(s) \rightarrow \textit{Poss}(a, s)$, where $\Phi(s)$ is a simple state formula about s. For instance $\textit{HasGun}(s) \rightarrow \textit{Poss}(\textit{shoot}, s)$ says that shooting can be executed whenever the agent has a gun, and $\textit{Poss}(\textit{tease}, s)$ that the turkey can always be teased.

Whereas all the extant approaches in the literature that allow for indirect effects of actions contain state constraints and effect laws, the status of executability laws is less consensual: some authors [Schubert, 1990; Doherty *et al.*, 1996; McCain and Turner, 1995; Thielscher, 1995] more or less tacitly consider that executability laws should not be made explicit but rather inferred by the reasoning mechanism. Others [Lin, 1995; Zhang *et al.*, 2002] have executability laws as first class objects one can reason about.

We nevertheless would like to point out that maximizing executability, as usually done in the literature, is not always intuitive: suppose we know that if we have the ignition key, the tank is full, ..., and the battery tension is beyond 10V, then the car (necessarily) will start. Suppose we also know that if the tension is below 8V, then the car will not start. What should we conclude in situations where we know that the tension is 9V? Maximizing executabilities makes us infer that it will start, but such reasoning is not what we want if we would like to be sure that all possible executions lead to the goal.

It seems a matter of debate whether one can always do without executabilities. We think that in several domains one wants to explicitly state under which conditions a given action is guaranteed to be executable, such as that a robot should never get stuck and should always be able to execute a move action. In any case, allowing for executability laws gives us more flexibility and expressive power.

Domain descriptions Given the four types of entities defined above, we arrange them in the following way: for a given action a, $\mathbf{Eff}_a$ is the set of its effect laws, $\mathbf{Inex}_a$ is the set of its inexecutability laws, and $\mathbf{Exe}_a$ is the set of its executability laws. $\mathbf{Stat}$ denotes the set of all state constraints of a given domain. Thus, $\mathbf{Eff}_a$, $\mathbf{Exe}_a$ and $\mathbf{Inex}_a$, for each action a, and $\mathbf{Stat}$ are the natural modules we consider here in designing a domain description.

For parsimony's sake, we define $\mathbf{Eff} = \bigcup \mathbf{Eff}_a$, $\mathbf{Inex} = \bigcup \mathbf{Inex}_a$, and $\mathbf{Exe} = \bigcup \mathbf{Exe}_a$. We suppose all these sets are consistent.

A *domain description* $\mathbf{D}$ is a tuple of the form $\langle \mathbf{Eff}, \mathbf{Inex}, \mathbf{Exe}, \mathbf{Stat} \rangle$.

Once the information contained in a module is not mixed with others', it can be expected that undesirable side effects due to further modifications are less likely to propagate to other parts of the domain description. The same thing can be obtained for the consistency check if beyond of being separated the modules are designed in such a way that their interaction is minimized. This is what we address in this section.

As we have seen, in software engineering functional independence is evaluated by means of two criteria: cohesion, a criterion for evaluating the relative functional strength of a module, and coupling, an assessment of relative interdependence among different modules. Both these notions are quite informal, even in software engineering, and cannot be measured in an objective way.

Here we explore these concepts when applied to domain descriptions and show how the informal requirements of software engineering can be turned into tests of consistency of several different arrangements of modules.

4 Cohesion

Normally cohesion comes with modularization, and its evaluation depends mainly on the entities that one takes into account when describing a domain.

In talking about sets of logical formulas we take cohesion as how simple or well-defined a logical module is, considering the different types of formulas that can be derived from it. We thus refine our first design principle:

P1'. *The less types of laws a given module entails alone, the more cohesive it is.*

As an example consider the following module:

$$\left\{ \begin{array}{c} \neg \textit{HasGun}(s) \rightarrow \neg \textit{Poss}(\textit{shoot}, s), \\ \textit{HasGun}(s) \rightarrow \textit{Poss}(\textit{shoot}, s) \end{array} \right\}$$

From such a set alone one can derive both $\neg \textit{HasGun}(s) \rightarrow \neg \textit{Poss}(\textit{shoot}, s)$ and $\textit{HasGun}(s) \rightarrow \textit{Poss}(\textit{shoot}, s)$, which are formulas of two different kinds. In this case we say that such a set is a lowly cohesive module, for alone it functions to derive executabilities and inexecutabilities. A better approach would be to decompose such a module into the following ones:

$$\mathbf{Inex}_{\textit{shoot}} = \{\neg \textit{HasGun}(s) \rightarrow \neg \textit{Poss}(\textit{shoot}, s)\}$$

$$\mathbf{Exe}_{\textit{shoot}} = \{\textit{HasGun}(s) \rightarrow \textit{Poss}(\textit{shoot}, s)\}$$

Total cohesion is not always easy to achieve. Suppose, for instance, a hypothetical situation in which we reason about the effects of drinking a cup of coffee:

$$\mathbf{Eff}_{\textit{drink}} = \left\{ \begin{array}{l} \textit{Poss}(\textit{drink}, s) \rightarrow \\ (\textit{sugar}(s) \rightarrow \textit{happy}(\textit{do}(\textit{drink}, s)), \\ \textit{Poss}(\textit{drink}, s) \rightarrow \\ (\textit{salt}(s) \rightarrow \neg \textit{happy}(\textit{do}(\textit{drink}, s)) \end{array} \right\}$$

Then, $\mathbf{Eff}_{\textit{drink}}$ entails $(\textit{sugar}(s) \wedge \textit{salt}(s)) \rightarrow \neg \textit{Poss}(\textit{drink}, s)$. This means that from $\mathbf{Eff}_{\textit{drink}}$ alone we do not get only effect laws but also inexecutability laws. Therefore $\mathbf{Eff}_{drink}$ is not as cohesive as one might have expected.

One step towards augmenting cohesion of a module of effect laws can be by completely specifying the preconditions of effects of actions. For example, the weaker effect laws

$$\mathbf{Eff}'_{drink} = \left\{ \begin{array}{l} Poss(drink, s) \to \\ (sugar(s) \land \neg salt(s)) \to happy(do(drink, s)), \\ Poss(drink, s) \to \\ (salt(s) \land \neg sugar(s)) \to \neg happy(do(drink, s)) \end{array} \right\}$$

guarantee a higher cohesion of module $\mathbf{Eff}'_{drink}$ in comparison to that of $\mathbf{Eff}_{drink}$.

By the definition of $\mathbf{Stat}$, it is easy to see that from the state constraints we can derive formulas of any type, so $\mathbf{Stat}$ is by nature a lowly cohesive module.

We are thus interested in refining even more our principle of high cohesion P1' by the following ones:

P1'-1. If $\mathbf{Inex} \mid\!\approx \Phi(s)$, then $\emptyset \mid\!\approx \Phi(s)$.

P1'-2. If $\mathbf{Inex} \mid\!\approx \Phi(s) \to Poss(a, s)$, then $\emptyset \mid\!\approx \Phi(s) \to Poss(a, s)$.

P1'-3. If $\mathbf{Exe} \mid\!\approx \Phi(s)$, then $\emptyset \mid\!\approx \Phi(s)$.

P1'-4. If $\mathbf{Exe} \mid\!\approx \Phi(s) \to \neg Poss(a, s)$, then $\emptyset \mid\!\approx \Phi(s) \to \neg Poss(a, s)$.

P1'-5. If $\mathbf{Exe} \mid\!\approx Poss(a, s) \to (\Phi(s) \to \Psi(do(a, s)))$, then $\emptyset \mid\!\approx Poss(a, s) \to (\Phi(s) \to \Psi(do(a, s)))$.

P1'-6. If $\mathbf{Eff} \mid\!\approx \Phi(s)$, then $\emptyset \mid\!\approx \Phi(s)$.

P1'-7. If $\mathbf{Eff} \mid\!\approx \Phi(s) \to Poss(a, s)$, then $\emptyset \mid\!\approx \Phi(s) \to Poss(a, s)$.

P1'-8. If $\mathbf{Eff} \mid\!\approx \Phi(s) \to \neg Poss(a, s)$, then $\emptyset \mid\!\approx \Phi(s) \to \neg Poss(a, s)$.

All these principles say is that a formula of a given type entailed by a module of a different type must be a theorem of the logic.

5 Coupling

As we have seen, coupling evaluates how much a module is tied to or dependent upon other modules. We take as coupling of two or more sets of different types of action laws how much interaction among them is needed to derive a formula of a given type. Interaction here means sharing logical formulas. Now we refine our second design principle:

P2'. *The less new consequences two or several modules have, the less coupled they are.*

(The new consequences of modules $\mathbf{M}_1$ and $\mathbf{M}_2$ are those consequences of $\mathbf{M}_1 \cup \mathbf{M}_2$ that are not consequences neither of $\mathbf{M}_1$ nor of $\mathbf{M}_2$ alone.)

For instance, consider the domain description $\mathbf{D}_1$:

$$\mathbf{Eff}_1 = \left\{ \begin{array}{l} Poss(tease, s) \to walking(do(tease, s)), \\ Poss(shoot, s) \to \\ (loaded(s) \to \neg alive(do(shoot, s))) \end{array} \right\}$$

$$\mathbf{Inex}_1 = \{\neg alive(s) \to \neg Poss(tease, s)\}, \mathbf{Exe}_1 = \emptyset$$

$$\mathbf{Stat}_1 = \left\{ \begin{array}{c} walking(s) \to alive(s), \\ dead(s) \leftrightarrow \neg alive(s) \end{array} \right\}$$

Observe that to derive the domain constraint $walking(s) \to \neg dead(s)$ one only needs $\mathbf{Stat}_1$, i.e., no other module is required for that. On the other hand, to conclude $dead(s) \to \neg Poss(tease, s)$ one needs both $\mathbf{Stat}_1$ and $\mathbf{Inex}_1$.

Totally decoupled descriptions are not common in applications of real interest. For the example above, it seems to be impossible to diminish the interaction between $\mathbf{Stat}_1$ and $\mathbf{Inex}_1$ without abandoning the concept of state constraints.

On the other hand, if $\mathbf{Exe}_1$ in our example contained $Poss(tease, s)$, things would be different: in this case, with $\mathbf{Inex}_1$ one would be able to infer the state constraint $alive(s)$, but such a law cannot be derived from $\mathbf{Stat}_1$ alone. A higher degree of interaction between this set and the others is necessary in order to do that. In such a case one would say that there is a high coupling among $\mathbf{D}_1$'s modules.

The principle of minimal coupling P2' can be refined in two more specific design principles:

P2'-1. **No implicit inexecutability laws**:
if $\mathbf{D} \mid\!\approx \Phi(s) \to \neg Poss(a, s)$, then
$\mathbf{Inex}, \mathbf{Stat} \mid\!\approx \Phi(s) \to \neg Poss(a, s)$

P2'-2. **No implicit state constraints**:
if $\mathbf{D} \mid\!\approx \Phi(s)$, then $\mathbf{Stat} \models \Phi(s)$.

P2'-2 is a useful feature of descriptions: beyond being a reasonable principle of design that helps avoiding mistakes, it clearly restricts the search space, and thus makes reasoning easier. To witness, if $\mathbf{D}$ satisfies P2'-2, then its consistency amounts to that of $\mathbf{Stat}$:

Theorem 5.1 If $\mathbf{D}$ has no implicit state constraints, then

$$\mathbf{D} \mid\!\approx \bot \text{ iff } \mathbf{Stat} \models \bot.$$

5.1 No implicit inexecutability laws

Consider the following domain description $\mathbf{D}_2$:

$$\mathbf{Eff}_2 = \left\{ \begin{array}{l} Poss(tease, s) \to walking(do(tease, s)), \\ Poss(shoot, s) \to \\ (loaded(s) \to \neg alive(do(shoot, s))) \end{array} \right\}$$

$$\mathbf{Inex}_2 = \mathbf{Exe}_2 = \emptyset,\ \mathbf{Stat}_2 = \{walking(s) \to alive(s)\}$$

From $Poss(tease, s) \to walking(do(tease, s))$ it follows with $\mathbf{Stat}_2$ that $Poss(tease, s) \to alive(do(tease, s))$, i.e., in every situation, after teasing the turkey is alive:

$$\mathbf{Eff}_2, \mathbf{Stat}_2 \mid\!\approx Poss(tease, s) \to alive(do(tease, s))$$

Now as $\mathbf{D}_2 \mid\!\approx \neg alive(s) \to \neg alive(do(tease, s))$, the status of fluent *alive* is not modified by the *tease* action, and we have $\mathbf{Eff}_2, \mathbf{Stat}_2 \mid\!\approx (Poss(tease, s) \land \neg alive(s)) \to (alive(do(tease, s)) \land \neg alive(do(tease, s)))$. From this it follows $\mathbf{D}_2 \mid\!\approx \neg alive(s) \to \neg Poss(tease, s)$, i.e., the turkey cannot be teased if it is dead. But $\mathbf{Inex}_2, \mathbf{Stat}_2 \not\mid\!\approx \neg alive(s) \to \neg Poss(tease, s)$, hence Principle P2'-1 is violated. The formula $\neg alive(s) \to \neg Poss(tease, s)$ is an example of what we call an *implicit inexecutability law*.

In the literature, such laws are also known as *implicit qualifications* [Ginsberg and Smith, 1988], and it has been argued that it is a positive feature of reasoning about actions frameworks to leave them implicit and provide mechanisms for inferring them [Lin, 1995; Thielscher, 1995]. The other way round, one might argue as well that implicit qualifications indicate that the domain has not been described in an adequate manner: inexecutability laws have a form simpler than that of effect laws, and it might be reasonably expected that it is easier to exhaustively describe them. (Note that nevertheless this is not related to the qualification problem, which basically says that it is difficult to state all the executability laws of a domain.) Thus, all the inexecutabilities should be explicitly stated, and this is what Principle P2'-1 says.

5.2 No implicit state constraints

Executability laws increase expressive power, but might conflict with inexecutability laws. For instance, let $\mathbf{D}_3$ be such that $\mathbf{Eff}_3 = \mathbf{Eff}_2$, $\mathbf{Inex}_3 = \{\neg alive(s) \rightarrow \neg Poss(tease, s)\}$, $\mathbf{Exe}_3 = \{Poss(tease, s)\}$, and $\mathbf{Stat}_3 = \mathbf{Stat}_2$. (Note that Principle P2'-1 is satisfied.) We have the unintuitive $\mathbf{Inex}_3, \mathbf{Exe}_3 \approx alive(s)$: the turkey is immortal! This is an *implicit state constraint* because $alive(s)$ does not follow from $\mathbf{Stat}_3$ alone: P2'-2 is violated.

The existence of implicit state constraints may thus indicate too strong executability laws: in our example, one wrongly assumed that *tease* is always executable. It may also indicate that the inexecutability laws are too strong, or that the state constraints are too weak.

6 Results for a dependence based solution to the frame problem

Given an axiomatic theory of actions with a solution to the frame and the ramification problems, we are interested in knowing whether domain descriptions encoded in it satisfy or not our set of design principles. Here we chose to use the modal framework of $\mathcal{LAP}_{\leadsto}$ [Castilho *et al.*, 1999], which has been shown to support Reiter's solution to the frame problem [Demolombe *et al.*, 2003] and also proposes an assessment of the ramification problem.

Let $tr_{SitCalc}$ be a translation of a domain description in $\mathcal{LAP}_{\leadsto}$ into the Situation Calculus. Dependences $a \leadsto l$ are translated into predicates $dep(a, l)$, meaning that action a *may cause* literal l to be true. The extension of *dep* is then circumscribed (cf. Schubert's explanation closure assumption). As examples, $dep(shoot, \neg walking)$ means that *shoot* may cause *walking* to be false, and the absence of $dep(tease, alive)$ induces the frame axiom $\neg alive(s) \rightarrow \neg alive(do(tease, s))$.

Theorem 6.1 If $\mathbf{D}_{\mathcal{LAP}_{\leadsto}}$ is a domain description in $\mathcal{LAP}_{\leadsto}$, then $tr_{SitCalc}(\mathbf{D}_{\mathcal{LAP}_{\leadsto}})$ satisfies Principles P1'-1—P1'-7.

Even in $\mathcal{LAP}_{\leadsto}$, however, it is possible to derive inexecutabilities from $\mathbf{Eff}$ (see the example in Section 4), which violates Principle P1'-8. Establishing maximal cohesion of $\mathbf{Eff}$ in this case involves weakening of preconditions of action effects. Anyway, conceiving an algorithm to accomplish this task is not difficult (due to space limitations we omit its presentation here).

Checking whether a domain description satisfies Principle P2'-2 can be made with little adaptation of the material on the subject present in the literature [Zhang *et al.*, 2002; Lang *et al.*, 2003; Herzig and Varzinczak, 2004]. We do not deepen into further details here, and just present the main results that we obtain when considering descriptions that satisfy the design principles that have been proposed (due to space limits no proof is given).

Theorem 6.2 Let $\mathbf{D}$ be the translation into Situation Calculus of a domain description in $\mathcal{LAP}_{\leadsto}$. If $\mathbf{D}$ has no implicit state constraints, then $\mathbf{D} \approx Poss(a, s) \rightarrow (\Phi(s) \rightarrow \Psi(do(a, s)))$ iff $\mathbf{Eff}_a, \mathbf{Inex}_a, \mathbf{Stat} \approx Poss(a, s) \rightarrow (\Phi(s) \rightarrow \Psi(do(a, s)))$.

This means that under P2'-2 one has modularity inside $\mathbf{Eff}$, too: when deducing the effects of action a we need not consider the action laws for the other actions. Versions for executability and inexecutability can be stated as well.

Theorem 6.3 There exist descriptions $\mathbf{D}$ not satisfying P2'-2 such that $\mathbf{D} \approx Poss(a, s) \rightarrow (\Phi(s) \rightarrow \Psi(do(a, s)))$ and $\mathbf{Eff}_a, \mathbf{Inex}_a, \mathbf{Stat} \not\approx Poss(a, s) \rightarrow (\Phi(s) \rightarrow \Psi(do(a, s)))$.

For example, just take $\mathbf{D}_3$ as before:
$\mathbf{D}_3 \approx Poss(shoot, s) \rightarrow (\neg alive(s) \rightarrow alive(do(shoot, s)))$, however $\mathbf{Eff}_{3_{shoot}}, \mathbf{Inex}_{3_{shoot}}, \mathbf{Stat}_3 \not\approx Poss(shoot, s) \rightarrow (\neg alive(s) \rightarrow alive(do(shoot, s)))$.

7 Related work

Pirri and Reiter [1999] have investigated the metatheory of the Situation Calculus. In a spirit similar to ours, they use executability laws and effect laws. Contrarily to us, their executability laws are equivalences and are thus at the same time inexecutability laws. There are no state constraints, i.e., $\mathbf{Stat} = \emptyset$. For this setting they give a syntactical condition on effect laws forcing them not to interact with executability laws, which precludes implicit state constraints. Basically, the condition says that when there are effect laws $Poss(a, s) \rightarrow (\Phi(s) \rightarrow \Phi(do(a, s)))$ and $Poss(a, s) \rightarrow (\Phi'(s) \rightarrow \Phi'(do(a, s)))$, then $\Phi(s)$ and $\Phi'(s)$ are inconsistent (which essentially amounts to having in their domain descriptions a kind of "implicit state constraint schema" of the form $\neg(\Phi(s) \wedge \Phi'(s))$).

This then allows them to show that such descriptions are always consistent. Moreover they thus simplify the entailment problem for this calculus, and show for several problems such as consistency or regression that only some of the modules of a domain description are necessary.

Amir [2000] focuses on design and maintenance of action descriptions applying concepts of the object-oriented paradigm in the Situation Calculus. In that work, guidelines for a partitioned representation of a given description are presented, with which the inference task can also be optimized, as it is restricted to the part of the domain description that is really relevant to a given query. This is observed specially when different agents are involved: the design of an agent's description can be done with no regard to others', and after the integration of multiple agents, queries about an agent's beliefs do not take into account the belief state of other agents.

In that work, executabilities are as in [Pirri and Reiter, 1999] and the same condition on effect laws is assumed, which syntactically avoids implicit state constraints.

Despite of using many of the object-oriented paradigm tools and techniques, no mention is made to the concepts of cohesion and coupling. In the approach presented in [Amir, 2000], even if modules are highly cohesive, they are not lowly coupled, due to the dependence between objects in the reasoning process defined there. We do not investigate this further here, but conjecture that this could be done there by, during the reasoning process, avoiding passing to a module a formula of a type different from those it contains.

The present work generalizes and extends Pirri and Reiter's result to the case where $\mathbf{Stat} \neq \emptyset$ and both [Pirri and Reiter, 1999; Amir, 2000] where the syntactical restriction on effect laws is not made. This gives us more expressive power, as we can reason about inexecutabilities, and a better modularity in the sense that we do not combine formulas that are conceptually different (viz. executabilities and inexecutabilities).

8 Conclusion

We have established a link between knowledge engineering and software engineering showing that many of the concepts and techniques developed for the latter are useful in the design and maintenance of domain descriptions. In particular, with the concepts of cohesion and coupling we get better criteria for domain description evaluation.

Our central hypothesis is that the different types of axioms should be neatly separated and only interfere in one sense: state constraints together with action laws may have consequences that do not follow from the action laws alone. The other way round, action laws should not allow to infer new state constraints, effect laws should not allow to infer inexecutability laws, etc.

At first glance, because of $\mathbf{Stat}$'s interaction with other modules, it could be said that domain descriptions described in our way do not completely minimize coupling. However, given the intrinsic nature of $\mathbf{Stat}$, observe that we cannot do otherwise: in the same way it is not possible to write completely decoupled methods (or the program will not work!), we cannot have totally decoupled domain descriptions (unless, of course, we constrain ourselves to domains without ramifications like in [Pirri and Reiter, 1999]).

It could be argued that unintuitive consequences in domain descriptions are mainly due to badly written axioms and not to the lack of modularity. True enough, but what we have presented here is the case that making a domain description modular gives us a tool to detect at least some of such problems and correct it. (But note that we do not claim to correct badly written axioms automatically and once for all). Besides this, having separate entities in the ontology and controlling their interaction help us to localize where the problems are, which can be crucial for real world applications.

References

[Amir, 2000] E. Amir. (De)composition of situation calculus theories. In *Proc. AAAI'2000*, pages 456–463, 2000.

[Castilho *et al.*, 1999] M. A. Castilho, O. Gasquet, and A. Herzig. Formalizing action and change in modal logic I: the frame problem. *J. of Logic and Computation*, 9(5):701–735, 1999.

[Demolombe *et al.*, 2003] R. Demolombe, A. Herzig, and I. Varzinczak. Regression in modal logic. *J. of Applied Non-classical Logics (JANCL)*, 13(2):165–185, 2003.

[Doherty *et al.*, 1996] P. Doherty, W. Łukaszewicz, and A. Szałas. Explaining explanation closure. In *Proc. Intl. Symp. on Methodologies for Int. Systems*, 1996.

[Finger, 1987] J. J. Finger. *Exploiting constraints in design synthesis*. PhD thesis, Stanford University, 1987.

[Ginsberg and Smith, 1988] M. L. Ginsberg and D. E. Smith. Reasoning about actions II: The qualification problem. *Artificial Intelligence*, 35(3):311–342, 1988.

[Herzig and Varzinczak, 2004] A. Herzig and I. Varzinczak. Domain descriptions should be modular. In *Proc. ECAI'04*, pages 348–352, 2004.

[Lang *et al.*, 2003] J. Lang, F. Lin, and P Marquis. Causal theories of action – a computational core. In *Proc. IJCAI'03*, pages 1073–1078, 2003.

[Lin, 1995] F. Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. IJCAI'95*, pages 1985–1991, 1995.

[McCain and Turner, 1995] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In *Proc. IJCAI'95*, pages 1978–1984, 1995.

[McCarthy and Hayes, 1969] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, volume 4, pages 463–502. 1969.

[McCarthy, 1988] J. McCarthy. *Mathematical logic in artificial intelligence*. Daedalus, 1988.

[Pirri and Reiter, 1999] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–361, 1999.

[Pressman, 1992] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1992.

[Schubert, 1990] L. K. Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, 1990.

[Shanahan, 1997] M. Shanahan. *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT Press, Cambridge, MA, 1997.

[Thielscher, 1995] M. Thielscher. Computing ramifications by postprocessing. In *Proc. IJCAI'95*, pages 1994–2000, 1995.

[Zhang *et al.*, 2002] D. Zhang, S. Chopra, and N. Y. Foo. Consistency of action descriptions. In *PRICAI'02, Topics in Artificial Intelligence*. Springer-Verlag, 2002.