Bart Bogaerts

4 Reasoning with first-order logic

4.1 Why is reasoning with first-order logic important within the broader artificial intelligence (AI) domain?

In this chapter, the general goal we focus on is the development of a so-called *knowledge-based* agent, which can reason based on a rich body of knowledge, stored in what we will call a *knowledge base*. While later chapters focus on *learning*, in this chapter, we assume that the knowledge is *provided* to the agent, most likely by a domain expert.

What this knowledge represents can depend on the context. For instance, it can be a symbolic representation of regulations relevant for its users, or it can be a representation of the causal mechanisms that govern the world in which the agent operates, or of strategic plans of a company, etc.

When representing knowledge, there are a couple of concerns to keep in mind. A first is the principle of *elaboration tolerance*; that is, when the world changes (e. g., new laws are introduced, company policies shift, the environment is changed), the agent's knowledge base should be easy to update, bring up to speed. To achieve this, it is important that this knowledge is presented in a *clear*, *understandable*, and preferably *modular* way. A second concern is that the underlying knowledge should also only be *represented once*. If the agent relies on its domain knowledge for multiple purposes, it should be able to use a single representation of that knowledge to guarantee consistency. In other words, the *representation of the knowledge is independent of the task to be solved*.

Once the agent is equipped with said knowledge base, we want it to perform various types of tasks, to solve problems that arise in the problem domain. As such, this chapter is concerned with two main tasks: *(1)* how to represent knowledge in a way that is understandable both for humans and computers and *(2)* how to exploit such represented knowledge for different forms of (deterministic) reasoning.

For the former, we study first-order logic (FOL), a much richer logic than propositional logic from the previous chapter; for the latter, we will show that in fact a theory in first-order logic allows for a wide variety of reasoning mechanisms.

First-order logic is chosen here as the base language because of its historic importance, as well as the fact that its connectives are simple, and have a crisp and clear informal semantics (by this, we mean that the formal meaning of statements in first-order logic is often what one would expect; as an example, first-order logic contains a quantor $\forall$ informally read as "for all"; as one can expect formulas of the form $\forall x : \phi(x)$ will hold in case ϕ is true for all possible objects). However, many other knowledge representation languages exist, taking different criteria into account (efficiency of reasoning methods,

Open Access. © 2024 the author(s), published by De Gruyter. This work is licensed under the Creative Commons Attribution 4.0 International License.
https://doi.org/10.1515/9783111426143-004

expressivity of the language, etc.); we discuss alternatives later in the chapter, as well as in Chapter 5, where we discuss knowledge representation languages known as “description logics,” which are designed specifically with the purpose of making deductive reasoning decidable.

In the previous chapter, we studied propositional logic and saw how modern SAT solvers can solve the satisfiability problem of propositional logic with remarkable ease. However, from the perspective of *representation*, propositional logic falls short: the knowledge it represents is always instance-specific; it lacks the expressive power to represent knowledge of a problem domain in a concise way, independently of the instance.

Throughout this chapter, we will use *hospital scheduling* as a running example. In this context, a very natural piece of knowledge is that everyone can only be assigned a single task during each shift (no one can do two tasks at the same time). While it is very easy to specify this in natural language, it is not easy (in fact, even impossible) to do this in propositional logic independently of a concrete instance.

Of course, it *is* possible to encode this natural language constraint, given a concrete set of employees, in propositional logic, but this encoding will have the following undesirable properties:

- it is instance-specific: if the set of employees changes, so does the encoding; and
- it is hard to read, and hence difficult to understand or debug.

On top of that, it is inherent to propositional logic that when the instances get larger (more personnel or more tasks), so does the representation of the knowledge that everyone can do at most one job at a single time. For instance, when considering the set of employees $A(lice)$, $B(ob)$, and $C(harlie)$ with two tasks T_1 and T_2 and two shifts S_1 and S_2, we could use propositional variables p_{ETS} with E an employee, T a task and S a shift, where p_{ETS} has the intended interpretation (the intended interpretation of a symbol is the concept in the real world it represents) that p_{ETS} holds if and only if E is assigned task T during shift S, one can craft the following encoding:

$$\begin{gathered}\neg(p_{A11} \land p_{A21}) \land \neg(p_{A12} \land p_{A22}) \land \\ \neg(p_{B11} \land p_{B21}) \land \neg(p_{B12} \land p_{B22}) \land \\ \neg(p_{C11} \land p_{C21}) \land \neg(p_{C12} \land p_{C22})\end{gathered}$$

While this formula correctly characterizes the given natural language constraint for the given instance, it raises challenges of understandability (it can hardly be called transparent and convincing someone of its correctness is not an easy job, especially if the instances get larger), as well as maintainability (if the knowledge of what constitutes a valid schedule changes, or even if the set of tasks changes, the entire formula needs to be updated). Furthermore, even if one could convince someone of the correctness of this encoding, nothing could be concluded about other instances. These limitations of

representation in propositional logic are in stark contrast with the simplicity of a natural language expression

"everyone can do at most one job at a single time."

To address the understandability problem, it would make sense to not use the actual propositional theory, but rather use a *schema* of how to generate the propositional theory. Concretely, here this could be something of the form

$$\neg(p_{EXS} \wedge p_{EYS}) \quad \text{for each employee } E \text{, each shift } S \text{, and each two distinct tasks } X \text{ and } Y.$$

While this representation solves the issue of understandability, it has other weaknesses, most notably that this notation is informal and not computer-readable. This is exactly where representation languages with a higher level of expressivity (such as first-order logic) come into play: first-order logic provides us with the means to express the knowledge underlying the scheduling problem compactly, in a formal way that is independent of the instance. Concretely, in first-order logic, we would make use of a relation *Assignment* with intended interpretation that $Assignment(e, t, s)$ holds if employee e is assigned task t at shift s. The constraint that every employee can be assigned at most one task during each shift then becomes

$$\forall e, \forall s, \forall t_1, t_2 : (t_1 \neq t_2) \Rightarrow \neg(Assignment(e, t_1, s) \wedge Assignment(e, t_2, s)),$$

or, equivalently,

$$\forall e, \forall s, \forall t_1, t_2 : Assignment(e, t_1, s) \wedge Assignment(e, t_2, s) \Rightarrow t_1 = t_2,$$

which intuitively states that if a single employee e is given the tasks t_1 and t_2 during a given shift s, then it must be that t_1 and t_2 are actually the same. Moreover, providing a problem-independent, formal description of this knowledge, provides us with the means to use it for various types of reasoning. Indeed, the knowledge about "what constitutes a good scheduling" is independent of any problem to be solved; while it is often associated to it, this knowledge is not inherently linked to the problem of *finding schedules*. It could also for instance be used for *proving properties that all valid schedules satisfy*, that is, to reason *independently of a specific instance*.

4.2 What category of problems does reasoning with first-order logic solve?

The focus of this chapter lies on problems that arise in *knowledge-intensive problem domains*. These are domains about which background knowledge is available, either

explicitly (for instance, in the form of regulations, policies, etc.), or implicitly (e. g., in the mind of a domain expert). In Chapter 7, we will study how to make intelligent agents in case the knowledge is *not* available explicitly, but instead we want to *learn* it from data. In such domains, we are concerned with problems whose solutions can *unambiguously*, and *deterministically* be defined in terms of this background knowledge. In Chapter 6, we will see other forms of reasoning that work *probabilistically*, for example, taking uncertainty about the world into account.

Some examples of such knowledge-intensive problem domains include *decision management* (where the logic underlying everyday business decisions can be made explicit), *scheduling* (where the knowledge about what constitutes a valid scheduling can be made explicit), *product configuration*(where the information about what constitutes a good configuration can be made explicit), legal reasoning (where laws or regulations can be made explicit), etc.

In such a knowledge intensive domain, many different problems can arise; and part of the research in logic-based reasoning is about classifying such problems as more generic problem patterns. These generic patterns are defined independently of the problem domain at hand, directly in terms of the logical representation. Some examples of such generic inference methods include:

– *Model checking*: Given a complete specification of a state-of-affairs, check whether this state of affairs indeed satisfies the knowledge that was made explicit. For instance, in case the knowledge describes "what are valid schedules" and the state-of-affairs is one concrete proposed schedule (that might have been crafted manually), this task boils down to checking whether the manually crafted schedule indeed satisfies the scheduling constraints.
– *Model expansion*: Given a partial description of the state-of-affairs, complete this to a *complete* description *in such a way that* the given knowledge is respected. In the scheduling example, the partial description of the state-of-affairs could be information about the available rooms and personnel and shifts, but not contain information about the actual schedule. The complete information would then also contain the actual schedule. That is, in that case the problem-specific task would be *searching* for a schedule. This inference is sometimes extended to *optimal model expansion*, where one is not just interested in searching *any* extension of the given input but one that is *optimal* with respect to a certain criterion. For instance in the scheduling domain, one might want to optimize conformance to explicated preferences of employees.
– *Querying*: Given a representation of the world, finding all instances that satisfy a certain logical formula. This type of querying corresponds exactly to querying a database (and in fact, many successful database languages are based on first-order logic).
– *(Finite domain) Propagation*: Given partial information about the world, derive more information based on the explicit knowledge. For instance in case a partial schedule is constructed, an automated reasoner could already infer that a given

employee could not do a certain task at a certain moment (because they already have another task at that moment, or because regulation forbids to fill more than two consecutive shifts, or ...)
- *Deduction*: Checking whether a certain logical sentence follows from the formalized knowledge independently of the instance. For instance, in the scheduling use case, one could use this to verify whether the constraints used for scheduling guarantee that the local fire safety regulations are respected (given a representation of the fire safety regulations in first-order logic). By proving this independently of the instance, we obtain the guarantee that all previous and future schedules created based on this knowledge indeed satisfy those requirements.

All the methods presented in this chapter will be based on an explicit representation of domain knowledge. This, however, does not mean that *all* knowledge needs to be represented formally. Indeed, again consider the hospital scheduling application. In such an application, a lot of knowledge will be easy to formalize (e. g., constraints imposed by the hospital, as well as constraints imposed by the government), but other parts of the knowledge might be *tacit*, that is, expert schedulers might know about certain sensitivities: who can work well with whom, and which kind of schedules are deemed fair by employees. This kind of knowledge is often hard to formalize. In such cases, we can still use the formalization of the “hard” knowledge, not to *solve* the complete scheduling problem, but to *assist* the expert scheduler, that is, the knowledge on scheduling policies can be used to develop a *decision support system*. Such a system can for instance use *propagation* to derive consequences of choices made the expert scheduler and even *explain* him/her why certain (undesired) consequences follow from other assignments. In situations like this, multiple inference can be used on the same knowledge; the idea of doing this is known as the *knowledge base system paradigm* (Denecker and Vennekens, 2008).

4.3 How are those problems solved?

In order to solve problems in knowledge-intensive domains, as described above, we need:
- A language in which domain knowledge can be expressed. This language should be understandable both by a computer (for reasoning) and by humans (for transparency, as well as to enable maintenance/updates).
- Reasoning engines that can exploit this knowledge.

As for the language, this chapter focuses on first-order logic, because of its historic importance in different domains, its expressive connectives, and its intuitive informal se-

mantics. However, many other knowledge representation languages exist, each of them designed with specific goals in mind. A nonexhaustive list of some examples follows.

For instance, Chapter 5 is concerned with the study of description logics (Baader et al., 2003), which are essentially *limitations* of first-order logic that enable efficient deductive reasoning. On the other hand, in other languages the expressivity of first order is *extended* instead of reduced, for example, by adding second-order features (such as one for instance in ProB (Leuschel and Butler, 2003) and Alloy (Jackson, 2002)).

Statements in first-order logic are in essence *objective*, they are either true in the world or not. However, in some cases one might want to express knowledge not about the actual world, but about the *state of mind of an agent*. For instance, rather than reasoning about what is true, in some cases it might be useful to reason about *what another agent knows*. For this purpose, a variety of so-called *epistemic logics* has been developed (Hintikka, 1962; Fagin et al., 1995; Halpern and Moses, 1990).

First-order logic is a *monotonic* logic. By this, we mean that if one can make a conclusion from a limited set of statements, adding more statements can never undo the conclusion, can only result in us making *more* conclusions. Certain natural language statements do not have this property. For instance, if I tell you that "Birds can usually fly," and "Tweety is a bird," it is natural to assume that Tweety can fly. However, when given more information, such as, for example, that Tweety is in fact a penguin, the conclusion might become invalid. This kind of logics is studied in the field of *nonmonotonic reasoning* (Reiter, 1980; Moore, 1985; McCarthy, 1986; Gelfond and Lifschitz, 1988).

Other types of languages, often with a basis in first-order logic, include

- languages to express dynamic domains, in which the world is modeled in a temporal setting and actions can change the state of the world (Reiter, 2001; Mueller, 2007; Reiter, 1991);
- languages to express *causal information* (Pearl, 2000; Pearl and Mackenzie);
- languages to express *spatial relations* (Cohn and Renz);
- domain-specific languages, to represent knowledge about a particular application domain. For instance when taking the example of modeling business logic and decision management, several domain-specific languages have been proposed (Group, 2008; Governatori, 2005; Abdelsalam and Shoaeb, 2016).

4.3.1 Representing knowledge in first-order logic

To represent knowledge in first-order logic, we will discuss three important concepts. The first is a *vocabulary*; it specifies the set of symbols we will use. By choosing the vocabulary, and agreeing on the informal semantics (the meaning in the real world) of each of the symbols, we determine which concepts in the world we can express knowledge about. Second, a *structure* consists of a domain (all the objects in the world), as well as an interpretation of the symbols in the vocabulary; it is an abstract representation of the world of interest; it has a *domain* (the set of all objects in the world) and assigns a

value (of the right type, see below) to each of the symbols. Third, a *formula* then represents the actual knowledge to be formalized; it specifies how different symbols from the vocabulary are related. Formulas should follow the *syntax* of first-order logic. The *semantics* of first-order logic is used to determine whether or not a formula is satisfied in a given world (structure).

Vocabularies

While propositional logic starts from a set of *propositions*, statements that can be true or false in the world, first-order logic takes the view that the world consists of (different sets of) objects (often called *domain elements*), with *relations* and *functions* between them. Uncoincidentally, this assumption is very similar to the one made in the context of relational databases (with a strong focus on relations there). Of course, it is debatable whether the real world is indeed made up of such objects and relations, but knowledge representation also starts from the idea that a certain abstraction of the actual world needs to be made. Thus, we will abstract away certain details of the world to be modeled and end up with a set of objects with relations and functions between them. This immediately brings us to the first challenge to be tackled when using knowledge representation techniques: *finding the right level of abstraction for representing a problem domain*. Often, an informal specification of the problem domain can give us an idea of what this level of abstraction should be. Take, for instance, the problem of hospital scheduling. A specification of what constitutes a valid scheduling will most probably mention *nurses* (a set of "objects" we cannot take abstraction of) and shifts (suggesting that using entire days as building blocks is probably too coarse, but scheduling per minute on the other hand is probably a too detailed representation). Such a specification will also mention that there is a relation between nurses, time points (abstracted in shifts) and assignments, etc.

The exercise described above does not just establish a level of abstraction, it also gives us an idea what the objects and relations between them will be. In other words, this exercise determines the *ontology* to be used in the representation of our domain knowledge. In first-order logic, the concept of a *vocabulary* is used to represent this information: a vocabulary is a collection of relation symbols (often called *predicate symbols*[1]), and *function symbols*, each with an associated arity—the number of arguments they take. Nillary function symbols (function symbols that take no arguments) are often called *constants*. For instance, in our running example on hospital scheduling, it is very likely that one needs:

- Constant symbols, such as *CEO*, to refer to specific persons (or other objects of interest in the domain).

1 First-order logic is also often called *predicate logic*.

- A set of nurses; this will become a *unary* (one input argument) symbol *Nurse*, meaning that every object in the abstracted world, either is a nurse or is not. For instance *Nurse*(*CEO*) will (as we see below) express that the chief executive officer (CEO) is a nurse, which can either be true or false.
- A set of employees containing all employees of the hospital; again, in the vocabulary this will correspond to a unary predicate symbol, for example, called *Employee*. It is very likely that all the nurses in the domain of interest are employees; we will see below how to express this.
- A unary predicate *Surgeon* representing a set of surgeons.
- A set of shifts (e. g., "Monday 6am–4pm" could be a shift); again, this will be represented by a unary predicate *Shift*: every object in the domain of discourse either is a shift or it is not.
- A set of possible qualifications, again represented by a unary predicate symbol, for example, *Qualification*.
- A relation between employees and qualifications stating who has which qualification; since this is a relation between two objects (for every employee e and every qualification q, either e has the qualification q or e does not have it), it is represented by a binary predicate symbol (e. g., *HasQualification*) in the vocabulary. The intended interpretation of this symbol is that $\mathit{HasQualification}(e, q)$ holds when e has the qualification q (e. g., a $C1$ qualification for the English language).
- A set of tasks to be performed, represented by a unary predicate *Task*.
- A relation stating who does what at which moment. Since this is a relation between three objects (a person, a task, and a shift), this is modeled by a ternary relation in first-order logic, for example, a relation *Assignment* with intended interpretation that $\mathit{Assignment}(e, t, s)$ holds if employee e is assigned task t at shift s.
- A relation stating who is on call during each shift: $\mathit{OnCall}(e, s)$ means that e is on call during shift s.
- A function *Manager* with intended interpretation that it maps every employee to its manager. This would be a unary function, taking one input argument; every function in first-order logic produces one output.

In the description of this vocabulary, already a couple of important points show up. First, these symbols are independent of which task one wants to solve in the context of hospital schedules: it might be used for checking whether a schedule satisfies the hospital's rules, for generating a work schedule, or for proving that the hospital's rules conform to the state regulation. Second, the choice of symbols, and thereby level of abstraction, determines which kind of information we will be able to express, and how easy it is to express that information. Third, with each symbol here we specified its *intended interpretation*; this is an often overlooked, but extremely important thing to do. Similar to documentation of procedural code, the intended interpretation of symbols is a means to communicate, which objects and relations in the world one will make claims about

and hence is crucial for making sure these claims are indeed well understood. Also similar to programming, it is important that the relations have a meaningful name, again to ensure readability of the formalized knowledge.

We already associated to the symbols an *intended* interpretation. However, symbols can also have an *actual interpretation*: symbols can have a value; for constant symbols, this is an element of the domain; for predicate symbols this value will be a relation, for function symbols, this will be a function. For instance, in the real world, everyone either has a certain qualification or not; this determines the actual value of the symbol *HasQualification*. Also, in a solution to the scheduling problem, each person is either assigned a task during a certain shift or not, thereby determining a value for the symbol *Assignment*. Such an interpretation of the symbols is called a *structure* and is discussed next.

Structures

The basic semantic object of first-order logic is a *structure*, often denoted S. It is a possible state of the (abstracted) world. A structure consists of two parts. On the one hand, it specifies a *domain* D: the set of all objects in the world. On the other hand, it also specifies for each symbol in the vocabulary an *interpretation*. The interpretation of a function symbol of arity n is a function from D^n to D; the interpretation of an n-ary predicate symbol is a set of n-tuples with elements in D, that is, an n-ary relation over D. Thus, in short, a structure determines

- a set of objects (its domain) D,
- for each constant symbol c, a domain element c^S (i. e., each constant symbol denotes an object).
- for each n-ary predicate symbol p a relation $p^S \subseteq D^n$;
- for each n-ary function symbol, a function $f^S : D^n \to D$.

For instance, in our scheduling example,

- The domain (the set of all objects) could be $\{Ann, Bob, Charlie, Q_1, Q_2, \dots\}$
- The interpretation of *Nurse* should be a set of 1-tuples, that is, elements of the domain. It could for instance be $Nurse^S = \{Ann, Bob\}$.
- The interpretation of *HasQualification* should be a set of 2-tuples indicating who has which qualifications. It could be $HasQualification^S = \{(Ann, Q_1), (Ann, Q_2), (Charlie, Q_1)\}$.
- The interpretation of *Manager* should be a function, mapping each employee to their direct manager. It could map *Ann* to *Bob* (i. e., $Manager^S(Ann) = Bob$), *Bob* to *Charlie* and *Charlie* to himself (assuming *Charlie* is the CEO).

It is important to remark that in our informal discussion, we only specified how the function *Manager* should behave *on employees*. In standard first-order logic, however, functions are assumed to be *total*. That is, they are supposed to map every tuple of values

to some value. In practice for us, this will mean that the function that *Manager* denotes will also assign values to qualifications, assignments, etc. Most likely, we will not be interested in such values. To deal with this in a more natural way, first-order logic is often extended either with *partial functions* (functions that do not need to map every object to a value) or with *types*. In a typed logic, we partition the domain of interest in different sets (the types), and in this setting, each symbol is not just given an arity but also a typing, for example, one could there state that *Manager* is a function *from employees to employees*, and thus get rid of the redundant information.

Terms and formulas

First-order logic now allows us, using the symbols from the vocabulary, to write complex expressions, as detailed below. In first-order logic, *terms* are expressions that denote an object. There are three types of terms that can denote an object of the world, constant symbols, function symbol applications, and variable symbols. For instance, if *HeadNurse* is a constant symbol, then it denotes an element of the domain in every structure; the interpretation of this symbol specifies which element it denotes; let us assume it denotes *Bob* in an extension of the structure described above. Another type of terms is the application of a function symbol (of arity n) to n terms. For instance (with $n = 1$), $Manager(HeadNurse)$ is a term as well. In our structure, it denotes *Charlie* since the expression *HeadNurse* denotes the object *Bob* and the interpretation of *Manager* maps *Charlie* to *Bob*. A last type of terms are *variables*, for example, x, y, z. These symbols are local to some quantification and their value is not part of a structure.

As can be seen, the semantics of terms is *compositional* in the sense that it works by giving meaning to an expression in terms of the meaning of its subexpressions, and *unsurprising* in the sense that it seems like the only reasonable semantics to be given to it. This is the case for the entirety of first-order logic. We now discuss all the language constructs that are used to form formulas in first-order logic. A formula in first-order logic denotes a truth value (true (**t**) or false (**f**)) in the context of a structure and assignment of values to its (free) variables. The basic building block are atomary formulas; they either equality atoms (equality between two terms) or a predicate symbol applied to the right number of terms. For instance,

$$Manager(HeadNurse) = HeadNurse$$

is true if and only if the head nurse is his/her own manager and

$$Employee(Manager(HeadNurse))$$

is true if and only if the manager of the head nurse is an employee.

First-order logic then makes use of the same Boolean connectives we know from propositional logic (Chapter 3): if φ_1 and φ_2 are formulas, then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ (and $\varphi_1 \Rightarrow \varphi_2$ and $\varphi_1 \Leftrightarrow \varphi_2$). Their semantics are as expected, for instance,

$$\neg Manager(HeadNurse) = HeadNurse$$

is true if and only if

$$Manager(HeadNurse) = HeadNurse$$

is false, that is, if the head nurse is not his/her own manager. Similarly,

$$Manager(HeadNurse) = HeadNurse \wedge Employee(Manager(HeadNurse))$$

is true if two claims hold: *(1)* the head nurse is their own manager, and *(2)* the head nurse's manager is an employee. The semantics of $\Rightarrow$ is a bit more subtle. This operator is to be understood as follows: $\varphi_1 \Rightarrow \varphi_2$ states that *if* φ_1 is true, then φ_2 must be true as well. Otherwise, no claim about the value of φ_2 is made and the proposition is always satisfied. For instance, consider the formula

$$Nurse(x) \Rightarrow Employee(x).$$

Taking abstraction for a moment of where the value of the x comes from, this formula states that *if* x is a nurse, then x is an employee as well. This proposition should surely hold for all xs that are nurses, but it actually holds for all nonnurses as well. To see this, assume x denotes an office chair. In that case, this statement holds as well: *if* my office chair is a nurse, then it is an employee as well. While this might seem a bit counter-intuitive at first, this kind of construct is actually used constantly in natural language. Consider someone saying to their friend "If we meet each other at the conference, we go for a drink together, I promise." Now assume the two people do not see each other at the conference (and hence of course, do not go for a drink), would that make the first person a liar? Most people would agree not. Hence, the logical implication (conditional) formalizes this sort of "if …then …" from natural language; however, it deserves to be mentioned that in natural language we often use "if …then …" to mean other things as well (causation, definition, etc.).

We now turn our attention to a language that really sets first-order logic apart from propositional logic. That is *quantification*. That is, we can say that a certain property holds for *all* or for *some* objects. Concretely, if, as discussed above, we wish to express that every nurse is an employee, we can express this as

$$\forall x : Nurse(x) \Rightarrow Employee(x).$$

The semantics of such a formula is defined as follows: $\forall x : \varphi$ is true if φ is true for all assignments of domain elements to x. Similarly, $\exists x : \varphi$ is true if φ is true for some (one or more) assignments of domain elements to x, allowing us for instance to express properties such as

$$\exists x : Employee(x) \wedge Manager(x) = x$$

stating that there exists an employee who is their own manager (e. g., the CEO).

While each of these language constructs are quite simple, combining and nesting them allows us to express complex constraints such as

$$\forall x : (\mathit{Nurse}(x) \land x \neq \mathit{HeadNurse}) \Rightarrow \mathit{Manager}(x) \neq x,$$

which states that all nurses different from the head nurse must have a manager that is different from themselves (i. e., no nurse except for the head nurse can be their own manager).

Remark 4.1. There is one subtlety that deserves some attention. When communicating (in natural language), humans rarely quantify over "everything" (i. e., all objects in the world). Instead, we usually quantify over restricted subsets. For instance, "all men are human" or "all lectures should be scheduled between 8am and 6pm." That is, universal quantification (and the same holds for existential quantification) in natural language is usually of the form

"All *P*'s are *Q*'s"

This kind of construct is sometimes called *binary quantification* but is not present in first-order logic. The way this will typically be written in FO is $\forall x : P(x) \Rightarrow Q(x)$. This statement states that *for all objects in the world*, if they are a P then they should also be a Q (and otherwise no restriction is imposed on them). Existential quantification in natural language takes the form

"There is a *P* that satisfies *Q*" or "Some *P* is a *Q*"

for instance "(at all times) there is a surgeon on call," where P is "surgeon" and Q is "is on call." In first-order logic, this would be expressed as $\exists x : P(x) \land Q(x)$, which immediately highlights an asymmetry between universal and existential quantification. Indeed, for conditional/binary quantification, a universal quantifier is paired with an implication while an existential typically occurs with a conjunction. Both connectives are visible when formalizing the previously mentioned statement "at all times, there is a surgeon on call" as

$$\forall sh : \mathit{Shift}(sh) \Rightarrow \exists s : \mathit{Surgeon}(s) \land \mathit{OnCall}(s, sh).$$

This rule-of-thumb can be of great value when debugging knowledge expressed in first-order logic: expression of the form $\exists x : \varphi \Rightarrow \psi$ or $\forall x : \varphi \land \psi$ are rarely correct.

The formal semantics of the different type of formulas is summarized in Table 4.1; the informal reading of different formulas is summarized in Table 4.2. Often, one will not be interested in a single arbitrary formula, but rather in a set of so-called *sentences*: formulas in which all variables are quantified (for instance, $\mathit{Nurse}(x) \Rightarrow \mathit{Employee}(x)$ is not a sentence; it is not clear what this expresses (the existence of such an x or that this

Table 4.1: Semantics of formulas in first-order logic.

formula	interpretation in structure *S*
$t_1 = t_2$	**t** if t_1 and t_2 have the same interpretation in S; **f** otherwise.
$P(t)$	**t** if $t^S \in P^S$; **f** otherwise.
$\phi \wedge \psi$	**t** if both ϕ and ψ are true in S; **f** otherwise.
$\phi \vee \psi$	**t** if at least one of ϕ and ψ is true in S; **f** otherwise.
$\phi \Rightarrow \psi$	**t** if ϕ implies ψ in S (i. e., if ϕ is true, so is ψ); **f** otherwise.
$\forall x : \phi(x)$	**t** if $\phi(d)$ is true for all the elements d in the domain of S; **f** otherwise.
$\exists x : \phi(x)$	**t** if $\phi(d)$ is true for at least one element d in the domain of S; **f** otherwise.

Table 4.2: Informal interpretation of first-order logic.

formula	informal interpretation
$t_1 = t_2$	t_1 and t_2 have the same value
$P(t)$	t is in the interpretation of P
$\phi \wedge \psi$	ϕ and ψ are both true
$\phi \vee \psi$	ϕ is true, or ψ is true, or both
$\phi \Rightarrow \psi$	if ϕ is true, so is ψ
$\forall x : \phi(x)$	ϕ holds for all possible values of x
$\exists x : \phi(x)$	ϕ holds for some possible value of x

holds for all such x?), while $\forall x : Nurse(x) \Rightarrow Employee(x)$ is a sentence). Such a set is often called a *theory*, or *knowledge base*.

This concludes our introduction to the type of knowledge representable in first-order logic. An overview of the most important syntactic objects in first-order logic can be found in Table 4.3. To increase natural expressivity, the operators discussed here are often extended, for example, to include aggregates (to naturally express "at least 5"), types (to make quantification more natural and make functions only range over the relevant domain elements), and many more constructs.

Table 4.3: Overview of the most important elements of the syntax of FO.

type of expression	(type of) value in a structure	examples
constant symbol	domain element	*CEO*, *HeadNurse*
function symbol	function	*Manager*
predicates symbol	set of tuples	*Employee*, *OnCall*
variable symbol	–	x, y
term	domain element	*CEO* $Manager(HeadNurse)$
formula	true or false	$\forall x : Nurse(x) \Rightarrow Employee(x)$. $Manager(HeadNurse) = CEO$

We now turn our attention to how this knowledge can be used for various forms of reasoning and how this relates to many other fields.

4.3.2 Reasoning with first-order logic

Now that we have presented the syntax and semantics of first-order logic, we are ready to present a variety of different forms of reasoning that use the knowledge expressed in first-order logic. In fact, the reasoning methods we present (often called *inference methods*) are not just applicable to first-order logic, but are applicable as well to other logics with a *model semantics*, that is, where the semantics is defined by formally stating which states of affairs satisfy expressions in the logic and which do not, that is, which structures are *models* of a theory. In FO, a structure S is called a *model* of a theory T if it satisfies all sentences in T (if all the knowledge in T is indeed true in S). Formally, S is a model of T if for all sentences ϕ in T, $\phi^S = \mathbf{t}$.

Some of the inference methods discussed below make use of a *partial structure*. A partial structure, like a normal structure has a domain and interprets symbols. The difference with a regular structure is that in a partial structure part of the information may be "unknown." For instance, it can be unknown who the CEO is, or it can be unknown whether or not "Alice is a nurse." Partial structures can be compared in *precision*: S_1 is *more precise* than S_2 if there are fewer things unknown in S_1 than in S_2, but everything that has a value in S_2 has the same value in S_1. In case S_1 is a normal structure more precise than S_2 we call S_1 an *expansion of* S_2. There are different ways to define such partial structures formally, the simplest one to think of, for now, is at the granularity level of symbols: *a partial symbol interprets only some symbols in the vocabulary, and leaves the value of the other symbols open/unknown.*

Inference method: **Model checking**
Given: a (finite) structure (an abstraction of the world) and a theory (the formalized knowledge)
Decide: whether or not all sentences of the theory are satisfied in the given structure

---

The first inference method discussed is quite a simple one: for *model checking*, one is given a structure and a theory (the formalized knowledge) and the problem at hand is deciding whether or not all formulas in the theory are satisfied in the structure, the abstraction of the world. That is, in the context of the running example, this inference method would boil down to checking whether a (for instance, manually created) schedule satisfies all the scheduling constraints. This inference method can be implemented very efficiently (in so-called *polynomial time* in terms of the size of the world). One way to implement this would be to directly apply the definition of the semantics, for instance, for the sentence $\forall x : Nurse(x) \Rightarrow Employee(x)$, given a structure S, we can check for each d in the domain whether $Nurse(d) \implies Employee(d)$ holds. As soon as we find one

instance for which it does not, we know the original sentence is not satisfied. If we finish this loop over the domain without finding one such instance, we know that it holds.

Inference method: **(Optimal) Model expansion**
Given: a (finite) partial structure (a structure that interprets some symbols, but not all of them) and a theory (the formalized knowledge) and an optimization criterion that specifies which worlds are preferred
Find: an expansion of the partial structure that satisfies the theory (and that is optimal with respect to the given criterion)

The second task, model expansion, is in fact a generalization of model checking. Here, the world is not completely given, but parts of it are to be searched. For instance in nurse scheduling, it would be a realistic assumption the parts of the structure given include which rooms/shifts/etc. there are, while the nongiven part includes the actual schedule. As such in this case, the problem of model expansion is the problem of searching a schedule that satisfies all the constraints (and that is optimal with respect to a certain criterion, e. g., how well it respects preferences of the employees or fairness). This problem is typically solved by the techniques from the previous chapter, such as SAT solving. To do this, the combination of the structure and theory is first reduced to a SAT problem by a technique called *grounding*, which eliminates all quantifications, intuitively by replacing a quantifier

$$\forall x : \varphi(x)$$

by

$$\varphi(d_1) \wedge \varphi(d_2) \wedge \cdots \wedge \varphi(d_n)$$

where the d_i are all the elements of the domain. Similarly, an existential quantifier

$$\exists x : \varphi(x)$$

would be replaced by

$$\varphi(d_1) \vee \varphi(d_2) \vee \cdots \vee \varphi(d_n).$$

When this procedure is applied recursively, for example, translating $\exists x : \forall y : \varphi(x, y)$, to

$$(\varphi(d_1, d_1) \wedge \varphi(d_1, d_2) \wedge \dots) \vee (\varphi(d_2, d_1) \wedge \varphi(d_2, d_2) \wedge \dots) \vee \dots,$$

the resulting formula will have no more quantifiers or variables. In case there are no function symbols, the result will be a *theory in propositional logic,* and hence the SAT solving techniques from the previous chapter are directly applicable to find satisfying assignments, which can then subsequently be translated back into a first-order structure. In case the original theory had function symbols, there are two options. The first

option is to replace the function symbols by predicate symbols by a technique that is known as graphing (a function f is replaced by a predicate G_f such that $G_f(x, y)$ holds if and only if $f(x) = y$), to again, arrive in propositional logic. The second option is to keep the function symbols, but in this case, after replacing all variables by all their instantiations, not only propositional symbols will remain, but also *finite domain variables*. In that case, a *constraint solver* can be used to solve the remaining problem, using all the techniques from the previous chapter.

Inference method: **Querying**
Given: a (finite) structure (an abstraction of the world) and a formula
Find: assignments to the free variables of the formula for which it is satisfied

---

The query inference is used mainly in the context of databases, where the structure is represented by a database, and the query typically by an structured query language (SQL) statement. However, it deserves to be mentioned that first-order logic (there often referred to as the *relational calculus*) lies at the basis of SQL. The goal of the query inference is to evaluate a certain formula (with free variables) in a given structure. For instance, in case a complete structure of the scheduling vocabulary (i. e., a complete schedule) is found, one might want to ask questions about it such as “which surgeons are on call during the July 11 Saturday AM shift?”, for instance, to inspect the schedule or to develop applications that visualize it. In first-order logic, this would be expressed as

$$Surgeon(s) \land OnCall(s, \text{“}Jul11 - SatAM\text{”})$$

or as

$$\{s \mid Surgeon(s) \land OnCall(s, \text{“}Jul11 - SatAM\text{”})\}$$

in case the variables whose instantiations is to be found is made explicit. For reference, the same query in SQL would be written as

SELECT person FROM *OnCall* NATURAL JOIN *Surgeon*
WHERE shift = “Jul11-SatAM” ;

Another example would be a query that searches for the set of all people who worked together with a certain individual who tested positive for COVID-19, which would be expressed as

$$\{p \mid \exists t, s : Assignment(p, t, s) \land Assignment(\text{“}PersonX\text{”}, t, s) \land s < Today\}$$

where *Today* is a constant interpreted as the current day (to only select shifts that have already passed, not shifts that are planned in the future). Computing the solutions to

such a query efficiently is extensively studied in the field of databases. One very important technique is *join reordering*, which when translated into first-order logic, essentially boils down to using the fact that for all formulas α, β, and γ, for instance,

$$(\alpha \wedge \beta) \wedge \gamma$$

is equivalent to

$$(\alpha \wedge \gamma) \wedge \beta,$$

and hence that in order to compute all instances that satisfy each of these three formulas, we can first compute all instance that satisfy any of the two and afterwards intersect with those instances satisfying the last formula. Join reordering techniques often make use of the size of the interpretation of certain symbols in order to reorder the query so that the internal processors will compute the result much faster.

Like model expansion, querying is a generalization of model checking; in case all occurrences of variables are quantified, the query becomes a so-called *Boolean query* and the task reduces to checking whether a formula is satisfied in the structure or not.

Inference method: **Finite domain propagation**
Given: a partial structure (an abstraction of parts of the world) and a theory (the formalized knowledge)
Find: a more precise structure (one that interprets more symbols) that is a consequence of the input

---

The task of propagation takes as input some partial information about the world and produces more refined (possibly still partial) information based on the logical theory. That is, it will only derive consequences of the theory given the current information. To define this formally, if the partial structure in the input is S and the theory is T, the output S' should be such that for every model M of T more precise than S, M is also more precise than S' (i. e., "no models are lost"). For instance, in the hospital scheduling application, this type of inference could be used in a support system for expert schedulers who make the schedule by hand, but in doing so interact with the assistant. For instance, if the scheduler has assigned a surgeon to be on call during a certain shift and the theory contains a constraint stating that exactly one surgeon should be on call during every shift, the system can automatically derive that none of the other surgeons should be on call. Or in case a nurse is scheduled to work a certain number of shifts in a period of time, the system could (depending on the actual regulations holding for that particular hospital) decide that by law the nurse cannot work any more shifts in that period of time. Such propagations can help the expert planner keep an overview of the consequences of their previous actions, or detect inconsistencies early on. In this kind of setting, a formalization of the knowledge underlying the scheduling problem is useful even when the scheduling problem is not solved fully automatically.

There are different ways in which propagation can be implemented. One way would make use of *grounding*, as described in the section on model expansion, to reduce the first-order problem to a propositional problem or a finite-domain constraint problem. Afterwards, the propagation can be done by the techniques seen in the previous chapter, for example, *unit propagation* or *constraint propagation*. Another solution would be to use model expansion to compute *all models* of the theory that are more precise than the given partial interpretation. Everything that is true in all those models is then a consequence of the given input. This second method is computationally much more demanding than the first, but also produces more precise information. Which of the two is preferred depends on the size of the problem (for small problems, computational cost is not an issue), and the required precision.

Inference method: **Deduction**
Given: two theories
Decide: if the first theory entails the second

Inference method: **Satisfaction checking**
Given: a theory (the formalized knowledge)
Decide: if the theory has a model

The last two types of inference methods are discussed together since they are two sides of the same coin. What is important to notice about these two inference methods is that they do not take a structure as input. As such, these inference methods operate purely on the formalized knowledge in an instance-independent way. Such types of reasoning are what really sets first-order logic apart from satisfiability solving, where conclusions are always for a specific instance. Let us discuss these methods in a bit more detail. The deduction inference methods takes two theories as input and checks whether the first entails the second, that is, whether all models of the first are also models of the second. In the nurse scheduling application, this could be used as follows: Suppose theory $\mathcal{T}_1$ contains the scheduling constraints given to an automatic scheduler, or to a system that supports an expert in creating a schedule. Now assume that $\mathcal{T}_2$ contains a formalization of a novel labor law hospitals are supposed to adhere to. In this case, we could wonder whether the “old” scheduling constraints already enforced conformance to the laws or not. If that is the case, we are sure that
- every schedule ever generated by the software remains valid, taking the new law into account, and
- no updates to the knowledge base used by the software are needed.

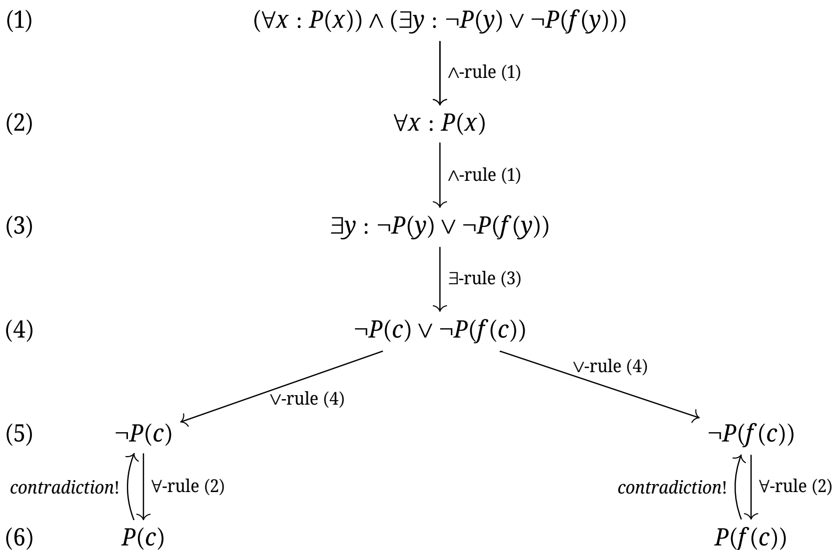
This can be checked using the deduction inference. Alternatively, we could also solve this problem using satisfaction: we can create a novel theory $\mathcal{T}_3$, which states that the all the “old” scheduling constraints are satisfied while the new law is violated, and check

whether this theory has a model. If it does, it means that the $\mathcal{T}_1$ does not entail $\mathcal{T}_2$. On the other hand, if it does not have a model, $\mathcal{T}_1$ does entail $\mathcal{T}_2$. The satisfaction can in a similar way be reduced to the deduction problem.

This stronger form of reasoning is the main focus of the research domain on *automated theorem proving*. As the name suggests, when deciding satisfiability of a logical theory, automated theorem provers will not just provide a yes/no answer, but in case the theory is unsatisfiable, they will also produce a *proof of unsatisfiability*. An often-used technique to create such proofs is the method of analytic tableaux. Here, a specified set of inference rules is used to construct a tree-shaped proof (called the *tableau*). This tree has the property that if each of its branches contains an inconsistency, the original formula is unsatisfiable. An example of a semantic tableau for the formula

$$(\forall x : P(x)) \wedge (\exists y : \neg P(y) \vee \neg P(f(y)))$$

is depicted. Several types of inference rules are applied here: the $\wedge$-rule allows reducing a formula $\alpha \wedge \beta$ to two formulas α and β. The $\exists$-rule transforms an existential statement $\exists x : \phi(x)$ into $\phi(C)$ where C is a new constant name, often called a *Skolem*. Intuitively, what it does is given the knowledge that some existential statement is true, it gives a name to an instance satisfying it. The $\forall$-rule simply instantiates universally quantified formulas: if a certain formula holds for all domain elements, then we can fill in any term and it should also holds for that term. The $\vee$-rule splits a branch in two branches: if it is known that $\alpha \vee \beta$ holds, and we can assume that both options lead to a contradiction, then the whole is unsatisfiable:



Another domain concerned with the satisfication problem is *SAT modulo theories*. Solvers in this field combine the efficient search methods of SAT solvers with rich modules supporting theories in first-order logic; specifically, for various theories, such as one axiomatizing certain forms of arithmetic, special-purpose propagators are developed that can check satisfication efficiently and translate their findings (lazily) into clauses for the SAT solver to avoid duplicate work.

4.4 What are the limitations of reasoning with first-order logic

The approach described in this section starts from the assumption that a large body of **domain knowledge is available and can be represented explicitly**. In case the assumption is satisfied, this is often considered a strength, since it gives you a lot of control, and typically results in high confidence in correctness of the conclusions of the system. However, there are many cases in which this assumption is not satisfied. To explicate knowledge about a problem domain, humans need not just be able to make correct decisions, but also need to be able to provide rational arguments about why certain decisions are made, and need to be able to align exactly when the same decision is to be made. Instead, humans often rely on *tacit knowledge*—knowledge that is difficult to transfer to another person by means of writing it down or verbalizing it—to make decisions, making it hard to replace the human completely by a knowledge-based system. For instance, in the context of scheduling, an expert scheduler might take personal relationships and preferences that are hard to formalize into account, and might, from experience have already learned that certain combinations work better than others. Furthermore, even when the knowledge is not tacit, the process of extracting it from experts possessing the knowledge is often not a one-shot procedure: it is not easy to provide a complete formalization of the knowledge used in decision-making. Instead, when doing so, the experts will often only realize that they use certain laws or rules in case they are presented with a situation that requires applying it. In Chapter 7, we will study different approaches that instead of starting from an explicit representation of knowledge start from data, for instance, historic decisions and *learn* from that data what the desired behavior is.

Another weakness is that the types of knowledge and reasoning studied in this chapter are all **deterministic**. That is, a black-and-white view on the world is taken: a structure is either possible (if it is a model, i. e., if it satisfies all the sentences in the theory) or impossible (if it is not a model) according to a given logical theory. In certain cases, however, this type of black-and-white representations does not suffice, for instance in case probabilistic knowledge (e. g., when rolling a die, there is a one in six chance of rolling a one) is relevant for the application. Such probabilistic approaches will be studied in Chapter 6.

Finally, first-order logic was presented here as a knowledge representation language, because of its natural informal semantics and historic importance. However, it is far from perfect (in fact, we believe that there is no such thing as the perfect knowledge representation language for all applications). There are two important points to be discussed here. The first is its **inability to express certain concepts (naturally)**. Certain concepts cannot be expressed in first-order logic. One example is the claim that one graph is the *transitive closure* of another graph. Formally, this means that in the one graph, there is an edge from node a to node b if and only if in the other graph, there is a *path* from node a to node b. Another example is in general *nonobjective information*, such as, for example, claims about the knowledge of another agent, for example, the other agent knows that I have either the King or the Queen of Spades. Certain concepts can be expressed, but cannot be expressed naturally. In this case, extensions are needed to improve the language. We already encountered this issue when discussing the lack of types: for instance, the fact that variables always range over “everything in the world,” is an unnatural assumptions that does not match with how quantifications occur in the wild (in natural language). This is often solved by using a multisorted extension of first-order logic. Another limitation would be for instance constraints of the form “at most 7 people satisfy a certain restriction” (e. g., at most 7 people can be in a room together at the same time). While this is expressible in FO, it would take several lines to do so and would be very error-prone. For this reason, FO is sometimes extended with *aggregates*. A second important point to mention that has historically been a source of great criticism, **undecidability of deductive inference**. It is well known that deduction for first-order logic is not decidable in general. Hence, **if** one wants to use their knowledge only for deductive reasoning, it makes sense to only consider fragments of the language. This observation gave rise to the field of description logics, as discussed in the next chapter.

4.5 Industry examples

4.5.1 Automated design-driven diagnostics for lithography machines at ASML

Pieter Van Hertum, Thomas Nagele

4.5.1.1 Introduction

ASML is the world’s leading provider of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or chips. A typical semiconductor manufacturing process consists of a number of steps, as can be seen in Figure 4.1. These are the 5 major steps:

1. **Deposition**: Semiconductors are made with silicon wafers (extremely smooth discs of 99.99 % pure silicon). In the deposition step, thin layers of conducting, isolating, or

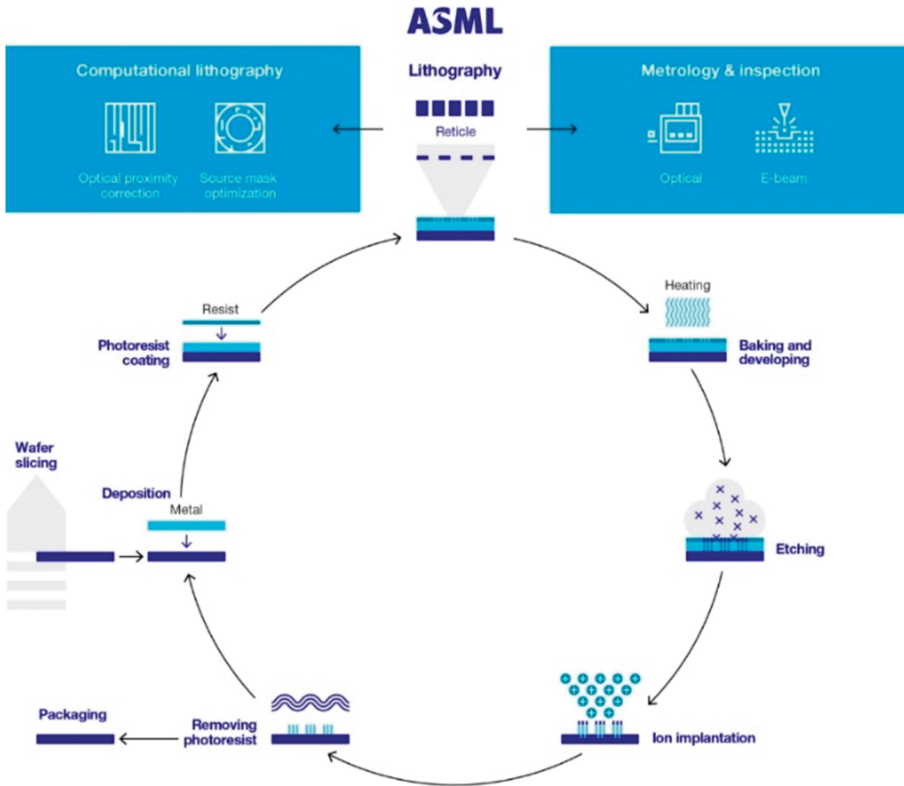


Figure 4.1: The semiconductor manufacturing process.

semiconducting material are deposited on the wafer to enable layers to be printed on.

2. **Photoresist Coating:** On top of this material layer, a layer of light sensitive coating, the photoresist, is deposited. This layer enables the subsequent step of printing patterns onto the wafer. A photoresist layer changes chemical structure when exposed to light, for example, becoming more soluble.
3. **Lithography:** The lithography step is the step in the manufacturing process where the actual patterns are transferred from a blueprint (a *reticle*) toward the wafer, using light that shines through the reticle. With the pattern encoded in the light, the pattern is shrunk by the system's optics and focused on the photosensitive wafer, chip by chip.
4. **Developing and etching:** When the pattern has been transferred, the next step is to remove the degraded photoresist. By etching, the redundant material is removed to reveal the intended 3D pattern. Baking and developing is done to fix the structure permanently.

5. **Ionization:** By bombarding the wafer with ions (positive or negative), the electrical conducting elements of the silicon are modified, in order to create the transistors. After the ionization step, the remaining layers of photoresist are removed to prepare the wafer for the next layer.

This process is repeated hundreds of times to create a wafer full of microchips, every step crucial and sensitive. Think of the resolution, focused and perfect positioning of the lithography step, the perfect depth that has to be etched or the sensitivity to the type of material in the deposition and coating steps. After this process, the wafer is cut into its individual chips (ranging anywhere from 10's to 1000's per wafer) and it is packaged and placed onto its baseboard.

To support this lithography step, ASML also develops tools to optimize and finetune the lithography machine and different types of measurements on the produced layers on the wafers.

4.5.1.2 Diagnostics of a lithography machine

A lithography machine is a complex piece of equipment, consisting of 10,000's different parts interacting to print patterns at nanometer scale. Sometimes, interactions between these parts or the aging of certain components can cause a machine to work suboptimally. ASML's service organization focuses on optimizing performance, and, in case of performance issues, on getting a machine back up to specification as quickly as possible.

The high physical complexity and nanometer resolutions lead to many interactions between different parts and modules, making system diagnostics a big challenge. Next to training customer support engineers to diagnose, maintain, and repair the machines in the field at our customer sites, ASML builds diagnostic software that supports them in this process. The tools automatically analyze data and suggest potential causes or tests the engineer can execute.

As a consequence of this high complexity, these tools need to combine domain expertise with the data to make sense of the machine structure. When diagnosing a root cause of a failure or performance problem of a lithography machine, it is this combination of data and knowledge that allows for causal reasoning in this complex domain.

4.5.1.3 Strategies for automated or supported diagnostics

To diagnose a complex machine, many data sources on that machine provide relevant information, coming in the form of software loggings and traces of physical sensors. Since software logs are introduced into the system by the domain experts in design, they are often the first source to use for diagnostics. For more complex issues, often caused by complex interactions between many modules that were not foreseen during design,

these logs are not sufficient and data traces have to be used. The data traces coming from the system sensors contain all types of measurements and need additional interpretation before becoming useable for diagnostics. Models are used to help the engineers to understand data that comes from the system. Such models can be based on the machine design or machine learning techniques being applied on historical and other data sources.

While data from events and interpreted sensor traces give a good overview on the current state of the machine, it does not—on its own—offer an understanding of what causes a certain issue, and what action can resolve it. To do this, the expert working on diagnosing and repairing the machine needs domain knowledge, which can be their own built-up expertise or through documentation. Here, we are looking to automate or support this process by incorporating this knowledge into the diagnostic tools themselves. The knowledge can originate from data, from engineer experience (feedback) or directly from the machine design.

When working with knowledge from expert experience, support engineer feedback or data, the goal is to leverage known and historical cases and experiences to learn relations between patterns of symptoms and root causes. This works well for issues that were identified in advance or appear multiple times. By specifying or learning the relation of a certain failure of a machine to a number of symptoms, a machine shows (sensors exceeding thresholds, specific loggings appearing, performance metrics dropping, etc.), it becomes possible to automate part of the diagnostic process.

Due to the large number of components, and consequently, the large number of sensors, symptoms, and potential causes, specifying every possible problem with their symptoms becomes impossible to scale. Learning these relations directly from the data and comparing machines to others is therefore applied more and more in industry context, and ASML is also incorporating this in its diagnostic landscape. Data-driven diagnostics is a booming field, and there is much to learn from the other machines in the field. This approach is strongest when you have many other samples to learn from (i. e., when diagnosing lightbulbs, coffee machines, or cars). Typically, a certain type of lithography machine has at most a few 100 machines (which even function in different configurations, making it more challenging). This problem of low sample size, together with the high complexity inspired to investigate a third solution: design-based diagnostics.

Instead of directly linking symptoms to causes, another option is to specify the intended behavior of a system and use this knowledge to reason backwards to the most likely root causes of a certain issue, by deducing and excluding. In this approach, the intended behavior of the system is first captured in logical formulae. When a failure occurs in one of the systems, its observations—such as sensor data or error logging—can be inserted in the logic structure, after which a diagnostic engine will compute possible explanations for what was observed.

4.5.1.4 An example

To control the system, it is filled with electrical and hydraulic circuits. Electricals are used to control the different subsystems, gather the sensor information and hydraulics for the supply of liquids, and for the distribution of cooling water over several other subsystems. To illustrate both our modeling approach and diagnostic method, we will apply the approach on an example of such a hydraulic system, which is a simple example based on a hydraulics subsystem of an ASML lithography machine. The hydraulic system consists mostly of interconnected pipes that transfer the water. Figure 4.2 shows a schematic representation of the hydraulic system. The components from the hydraulic domain are shown in blue and the electronic domain is shown in green. A pump pushes the water via a pipe through a manifold, which splits the water over two branches. Each of the branches can be manually closed by a valve at the end of the branch. A manifold combines the water coming from the branches again toward the final pipe. The water pressure is measured with pressure sensors at the end of each branch. The pump and the pressure sensors are powered by two separate power supplies.

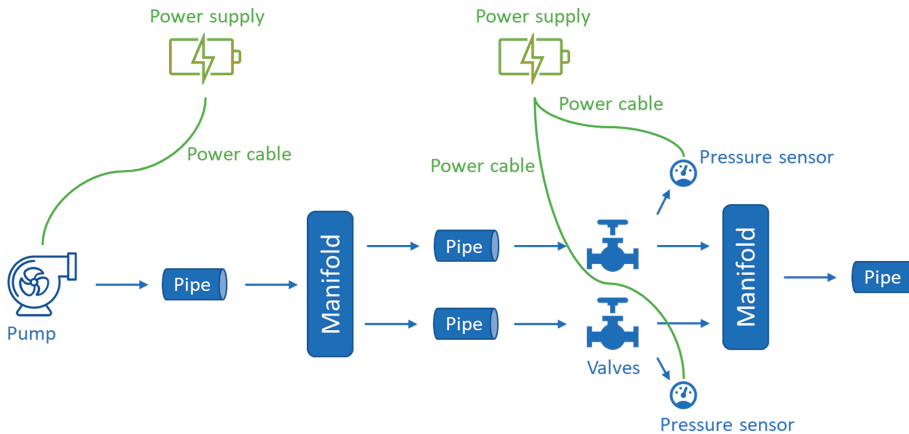


Figure 4.2: A schematic representation of the hydraulic system. The hydraulic domain is shown in blue, and the electronic domain is green.

Modeling behavior

The schematic shows the components of the system and how these are connected to each other, but it does not provide the behavior yet. The intended behavior is specified per component, after which the system behavior is the composition of those individual behaviors by connecting the components to each other. The behavior of each component is specified as the relation between inputs and outputs. These inputs and outputs represent the (physical) variables at either side of the component in a discrete way to abstract away from the highly detailed physics domain. For example, relevant properties of the

water flowing through a system can be pressure and flow. The intended behavior is that the water entering the pipe will also exit. If there is water going in, but nothing exiting the pipe must leak and if pressure is applied to the water, and it is not flowing, it must be blocked. This expression of both normal and failure behavior can be translated to a logic expression.

A logic specification

To specify the behavior of pipes, we would need following symbols:[2]

- A type ***Pipe***, used to specify the collection of pipes we have in a system.
- A type ***HealthStatePipe*** stating the possible health states of a pipe, containing *HealthyPipe, BlockedPipe, LeakingPipe*.
- A predicate ***Connected***(***Pipe***, ***Pipe***) to specify the connection between two pipes.
- A predicate ***FlowIn***(***Pipe***) specifying the flow coming into a certain pipe.
- A predicate ***FlowOut***(***Pipe***) specifying the flow coming out of a certain pipe.
- A predicate ***PressureIn***(***Pipe***) specifying the pressure at the beginning of a certain pipe.
- A predicate ***PressureOut***(***Pipe***) specifying the pressure at the end of a certain pipe.
- A function ***StatePipe***(***Pipe***) : ***Pipe*** $\longrightarrow$ ***HealthStatePipe*** specifying the health state of a certain pipe.

These symbols are stored in a Vocabulary $\boldsymbol{V_{pipe}}$.

A theory $\boldsymbol{T_{pipe}}$ is used to specify the relations between these symbols, detailing out pipe behavior.

- *For a healthy pipe, the input state of the fluid always equals the output state.*

$$\forall p[Pipe] : StatePipe(p) = HealthyPipe \Leftrightarrow PressureIn(p)$$
$$= PressureOut(p) \wedge FlowIn(p) = FlowOut(p)$$

- *If no fluid can pass through (pressure but no flow), the pipe is blocked.*

$$\forall p[Pipe] : StatePipe(p) = BlockedPipe$$
$$\Leftrightarrow PressureIn(p) \wedge \neg FlowIn(p) \wedge \neg PressureOut(p)$$

- *If there is fluid going in, but not coming out, the pipe is leaking.*

$$\forall p[Pipe] : StatePipe(p) = LeakingPipe \Rightarrow FlowIn(p) \wedge \neg FlowOut(p)$$

---

2 Note that we are using a typed extension of first-order logic, where the domain is split into a number of subsets, called types. Predicates, functions are defined over those types instead of over the entire domain. When writing a theory, we allow quantification over these types.

This specification now depicts all possible behaviors of the pipe, together with a health state. This health state can identify the state of the pipe and later be used to diagnose specific issues. Together with the following theory containing a few *commonsense* constraints, inferences can be executed on this specification to build diagnostic solutions.

Theory $\boldsymbol{T}_{common}$:

– *Flow cannot be created:*

$$\forall p[Pipe] : \neg FlowIn(p) \Rightarrow \neg FlowOut(p)$$

– *Pressure cannot be created:*

$$\forall p[Pipe] : \neg PressureIn(p) \Rightarrow \neg PressureOut(p)$$

– *No flow without pressure:*

$$\forall p[Pipe] : FlowIn(p) \Rightarrow PressureIn(c)$$
$$\forall p[Pipe] : FlowOut(p) \Rightarrow PressureOut(c)$$

– *The meaning of "Connected":*

$$\forall p1[Pipe]p2[Pipe] : Connected(p1, p2) \Leftrightarrow FlowIn(p2)$$
$$= FlowOut(p1) \wedge PressureIn(p2) = PressureOut(p1)$$

In similar ways, specifications for the functioning of valves, manifolds, pumps, power supply units (PSUs) can be created by domain experts. By combining these specifications and supplying information of the known state of the world (the specific pipes, valves, pumps, etc., their connections and sensor measurements) diagnostic reasoning can be done.

Inferences

Using the model for diagnosis

Once the behavior of the complete system has been captured in a logical specification, it can be used to help the service engineer in finding the defective part. The diagnosis is based on what is observed from the system, a combination of the data coming from sensors and software logging. An input structure for the model can be compiled from the observed data, after which model expansion inference can be used to find explanations for the observations.

For example, given the pipe specification above, the following structure could be compiled from design information and observations:

$$
\begin{aligned}
S_1 = \{ & \\
& \mathit{Pipe} = \{p1, p2\} \\
& \mathit{Connected} = \{(p1, p2)\} \\
& \mathit{FlowOut} = \{p2\} \\
& \}
\end{aligned}
$$

By performing model expansion on this theory, with this partial structure, the following complete structure is calculated:

$$
\begin{aligned}
S = \{ & \\
& \mathit{Pipe} = \{p1, p2\} \\
& \mathit{Connected} = \{(p1, p2)\} \\
& \mathit{FlowIn} = \{p1, p2\} \\
& \mathit{PressureIn} = \{p1, p2\} \\
& \mathit{PressureOut} = \{p1, p2\} \\
& \mathit{FlowOut} = \{p1, p2\} \\
& \mathit{StatePipe} = \{p1 \rightarrow \mathit{HealthyPipe}, p2 \rightarrow \mathit{HealthyPipe}\} \\
& \}
\end{aligned}
$$

This structure (the only full structure extending the partial structure and satisfying the theory) states that both pipes are performing healthy and as such, the fluid is flowing through the system, and the pressure is propagated as expected.

However, when starting with the following structure:

$$
\begin{aligned}
S_2 = \{ & \\
& \mathit{Pipe} = \{p1, p2\} \\
& \mathit{Connected} = \{(p1, p2)\} \\
& \mathit{FlowIn} = \{p1 \rightarrow \mathit{Flow}\} \\
& \mathit{FlowOut} = \{p2 \rightarrow \mathit{NoFlow}\} \\
& \}
\end{aligned}
$$

the model expansion inference creates many different possible structures that extend this partial structure and satisfy the theory, for example, the first pipe can leak, or the second pipe can leak. This approach shows its value when reasoning over larger specifications, such as shown in Figure 4.2. While only expecting a domain expert to specify simple behavior of components and how they are interconnected (which can be further simplified through a graphical interface), quite complex possible scenarios can be calculated with incomplete input information.

When handling more complex scenarios with limited input information, there are often many possible structures that satisfy a theory extending the input. For example, in the Figure 4.2 specification, where there is flow coming into the first pipe and not coming out of the last pipe, it could be that the first pipe is leaking, or it could also be a failure of both parallel valves. When reasoning, it is often good practice to assume that failures happen rarely, so a model with less failures is more likely than a model with more failures. To this end, an optimization inference can help. By adding a term $t = \#\{p \mid \mathit{StatePipe}(p) \neq \mathit{HealthyPipe}\}$ and minimizing that term, we encode the assumption that in most situations components are behaving as we expect.

Assessing diagnosability during design

The model can also be used during the system design to assess its diagnosability. Instead of using observations coming from the real system as input to the model, one can also run the inference for an assumed failure. For this, the observability configuration should be known, which comprises a list of variables in the system for which you know you can observe its value, either via a sensor or software logging.

Through inference, one can assess the values on the observable variables when all components are healthy to understand what the observables of the running system will tell when everything is working as expected. This set of readings on the observables is referred to as a *signature*. This assessment can also be done for every single failure mode in the system. For this, all other components are assumed to behave normally, while only one component has a problem. Each of these failure simulations provides one or more signatures, which are the computed failure signatures.

Repeating this assessment for all failures or possibly even failure combinations results in a set of failure signatures. Depending on the numbers and locations of observables in the system, multiple failures may have identical signatures. Based on this insight, the designer may add more observability to the design to reduce the number of failures with identical failure signatures, or procedures could be formulated to help the service engineer in the field to find out what failure really caused the issue.

4.5.1.5 Conclusion

When doing diagnostics, machine learning can help us to interpretate data sources and sensor data. However, to separate cause from effect, and optimally make use of the available knowledge, reasoning systems can help. By specifying system design and interpreting software loggings and sensors, inferences can be used to support diagnostic tooling, or to support system design by analyzing observability.

In order to enable this for entire systems, the scalability of building these specifications is crucial. Good (graphical) interfaces and tooling can help immensely for domain experts to build these specifications. On top of this, it is important to build tools and

study techniques that can (semi)automatically build these specifications directly from design documents.

4.5.2 Modeling and verifying simple vehicle controller, such as the Triton unmanned aircraft systems of the US Navy: using Imandra system and first-order logic

Djordje Markovic, Bart Bogaerts, Grant Passmore

4.5.2.1 Introduction

Designing complex systems is quite an extensive and expensive process, and sometimes mistakes are just not affordable. In these cases, before developing the desired product, it is essential to model it and prove its specific properties. Formal verification methods often use mathematical logic and symbolic AI (artificial intelligence) for designing and analyzing engineering artifacts such as software and hardware. The difficulties are that complex systems strive to have infinitely many different possible behaviors. This seemingly miraculous feat— surveying an infinite number of possible system behaviors through a finite computation—is made possible using symbolic mathematics and mechanized techniques for logical inference.

Formal verification has a tremendous use-value for safety-critical systems, that is, computer systems whose correctness have a direct bearing on the safety of others. For example, autopilot systems in aircraft, control systems in nuclear power plants, and collision avoidance controllers in drones are directly related to public safety.

In practice, designing and tuning functions like the controller is a considerable challenge, and formal verification is necessary to ensure they operate correctly and safely. Consider that you are creating an algorithm for controlling some aspect of an aircraft; would you be able to trust it and be a passenger on a test flight? So, before giving it a test ride, how is the safeness of the controller verified? The first step is often simulation. That is, we may first gain some primary assurance of its safety and correctness through simulating its behavior in many different situations. However, no matter how many unique runs are done through a simulator, only a finite number of scenarios can be observed. But this controller can, in general, be in an infinite number of possible situations. How can this gap be bridged? How can we ensure that the algorithm is correct? Formal verification provides an answer. The key is to use logic and reason symbolically about its possible behaviors to prove that the controller follows desired safety and correctness properties.

In this section, we shall model a *simple autonomous vehicle controller* dynamic system and verify its correctness. This controller may be seen as a simplified version of a controller found in modern day drones and autopilot systems, such as the Triton UAS of the US Navy.

The system is modeled in first-order logic and proofs are explained accordingly. Nevertheless, we give a few examples using an Imandra[3] syntax side-by-side with first-order logic statements, illustrating the connection between reasoning about programs and first-order logic. These parts are more suitable for advanced readers, and they are not relevant for the understanding of this section; therefore, less experienced readers can safely skip them.

The upcoming text is split into two parts: first, the discussion of modeling of a simple autonomous vehicle controller and, second, detailed analysis and proof of the two safety properties of the system.

4.5.2.2 The domain knowledge

This section introduces the problem of a *simple autonomous vehicle controller* and essential elements needed for modeling such a system. The analyzed controller is originally described in the article (Boyer et al., 1990) by Boyer, Green, and Moore.

The goal of a vehicle controller is to take care of wind changes and keep the vehicle on the course. The brief specification of the system follows: The system is restricted to only one space dimension (y-component). The time is abstractly represented as a sequence of discreet time points. The vehicle can move with a certain velocity in the positive or negative direction of the y-axis, and wind can blow with a particular speed (in the positive or negative direction of the y-axis). Wind cannot change more than one unit between two time points. The drone controller can increase vehicle velocity at any time point for an arbitrary value. The controller has an insight into all values at a certain time point.

---

3 Imandra is a formal verification environment that facilitates the design and verification of safety-critical algorithms. More on the official website: https://www.imandra.ai/

Representing the world (state of affairs)

Formal modeling of any system usually starts with the choice of an adequate ontology. Ontology serves to represent the *states of affairs* in the world we are formalizing. Naturally, this choice is of immense importance because it has a strong impact on formalization.

First, it is important to notice that first-order logic does not have the concept of time, and yet we would like to model a dynamic system. The common approach is to interpret time as natural numbers,[4] zero being the *start* point in time and *next* being a function mapping time point n to time point $n + 1$.

Once we have the abstract representation of the time, we can start representing the simple vehicle controller system. The system is described by wind speed, vehicle speed, vehicle position, and wind change at any time point. As we abstract away units, all these values are represented by integers. Knowing that values are unique per time point, it is natural to represent them as temporal functions mapping time points to their values.

Finally, vehicle *controller* should update vehicle speed based on two consecutive time points. However, we shall see later the abstract version of the controller that takes as input two integers and returns relative speed change. Table 4.4 supplies first-order logic vocabulary suitable for a simple vehicle controller system.

Table 4.4: Vehicle controller— first-order logic ontology.[5]

First-order logic:	
Temporal functions describing state:	
– $w : Time \to \mathbb{Z}$	– Wind velocity
– $y : Time \to \mathbb{Z}$	– Position of the vehicle
– $v : Time \to \mathbb{Z}$	– Velocity of the vehicle
Temporal function describing wind change:	
– $dw : Time \to \mathbb{Z}$	– Wind change
Controller function:	– Binary controller function
– $controller: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	
Time vocabulary:	
– $start : () \to Time$	– Starting point
– $next : Time \to Time$	– Next function

4 For the simplicity of the example, we assume that every structure in first-order logic contains two types, integers $\mathbb{Z}$ and Time. Time is interpreted by natural numbers, where “start” is a constant designating 0 in every structure and “next” is a function mapping each time point to the following (i. e., mapping n to n+1).

5 We use (w, y, v) to represent a state in the later text, dw is not part of the state, and it can be treated as a parameter of the system.

State transition

The system that we want to model is dynamic; the vehicle can move, and the wind can blow in one direction of the y-axis with a certain velocity. The vehicle controller is supposed to control vehicle speed and balance the wind impact. An essential property of this dynamic system is that it is deterministic, that is, for any state of a system (including the wind change), there is exactly one next state. Here, we are going to model this transition between two states.

For a given current state and wind change, we can compute the next state of the system. Wind velocity changes with respect to the wind change. Vehicle position changes concerning the previous position, vehicle velocity, and new wind velocity. The exciting part is how the controller updates the vehicle velocity, but let's keep it abstract for a moment.

The next state function is defined as: For a given state (w, y, v) and for a wind change dw, the new state (nw, ny, nv) is

– Wind change: $nw = w + dw$
– Vehicle position is changed: $ny = y + v + w + dw$
– New vehicle velocity depends on the controller: $nv = v + controller(w, y, v, dw)$

The controller from (Boyer, 1990) is more abstract and considers a sign of the new and old position of the vehicle. So, we can model it as a $controller((sgn(ny)), (sgn(y)))$.

Table 4.5 shows the first-order logic statements expressing this state transition. It is important to notice that each statement starts with universal quantification "For each time point t…" So, we are saying something about time. Let us translate the first statement to the natural language statement given standard informal semantics for first-order logic and expected interpretation for w – wind and dw – wind change. The first statement expresses: "*For each time point t, wind at the next time point is equal to the wind at time point t augmented with wind change at time point t*." This translation clearly states the informal interpretation of the first statement, and it precisely describes our thoughts of this dynamic system.

Table 4.5: Vehicle controller—first-order logic specification.

First-order logic theory – T_{vc} (Theory – vehicle controller):
$\forall t : w(next(t)) = w(t) + dw(t).$
$\forall t : y(next(t)) = y(t) + v(t) + w(t) + dw(t).$
$\forall t : v(next(t)) = v(t) + controller(sgn(y(t) + v(t) + w(t) + dw(t)), sgn(y(t))).$

Table 4.6 represents the same function in the Imandra syntax. The syntax is clear, and the representation is very readable and compact. This function returns a new state vector that represents the state after applying wind change dw to the state vector s. One

Table 4.6: Vehicle controller—Imandra specification.

Imandra:

next_state dw s = {
  w = s.w + dw;
  y = s.y + s.v + s.w + dw;
  v = s.v + controller (sgn (s.y + s.v + s.w + dw)) (sgn s.y)
}

can see that statement talks about two consecutive time points, but time is not mentioned explicitly.

Controller

Let's assume that the controller function is defined as

$$\forall x, y : controller(x, y) = (-3 * x) + (2 * y).$$

Given such a controller, we may first gain some elementary assurance of its correctness through simulation and testing. Let us look at one particular use case when the wind increases in the positive direction of the y-axis for three consecutive time points and then stays constant for four time points.

The scenario from the Table 4.7 is also graphically represented in Figure 4.3. One can observe that after four time points of constant wind, the vehicle is back at the course. We can try this many times with different setups, and the vehicle always gets back at the center. Furthermore, we are not able to find counterexamples. In an analogous way, one can notice that the drone never strays further than 3 units from the center. All these suggest that these properties always hold. But how can we prove them?

Table 4.7: Vehicle controller simulation example.

Time	**0**	**1**	**2**	**3**	**4**
w	0	0 + 1 = 1	1 + 1 = 2	2 + 1 = 3	3 + 0 = 3
y	0	0 + 0 + 0 + 1 = 1	1 − 3 + 1 + 1 = 0	0 − 1 + 2 + 1 = 2	2 − 4 + 3 + 0 = 1
v	0	0 − 3 = −3	−3 + 2 = −1	−1 − 3 = −4	−4 − 1 = −5
dw	1	1	1	1	0

Time	**5**	**6**	**7**	**8**
w	3 + 0 = 3	3 + 0 = 3	3 + 0 = 3	3 + 0 = 3
y	1 − 5 + 3 + 0 = −1	−1 + 0 + 3 + 0 = 2	2 − 5 + 3 + 0 = 0	0 − 3 + 3 + 0 = 0
v	−5 + 5 = 0	0 − 5 = −5	−5 + 2 = −3	−3 + 0 = −3
dw	0	0	0	0

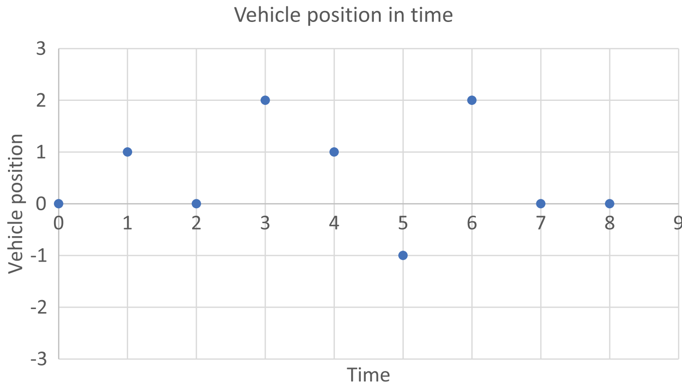


Figure 4.3: Controller behavior when wind remains steady for four time points.

4.5.2.3 Reasoning and theorem proving

This section opens with a concise introduction to theorem proving. Suppose that there is a theory T and a formula φ both expressible in first-order logic. Proving that T entails φ would be as simple as formalizing both and using an adequate automated theorem prover. However, these provers could struggle to prove the complex properties of dynamic systems. The main reason for these problems is the *time*, which is interpreted by natural numbers. One typical way to solve these problems is induction, which naturally can be conducted on time: *First, we prove that the property holds at the first time point, and then we prove that the property is preserved by the transition function*.

Since induction is a crucial concept in the following text, it deserves a brief explanation. Induction proofs in the context of dynamic systems are usually applicable for proving single state properties. A single state property ($\forall t : \varphi[t]$) is a formula quantifying over *time* and the only time point mentioned in the body of the formula is the quantified one (t). Consider a case of proving that theory T entails some property $\forall t : \varphi[t]$, the induction proof would consist of

- *Base case* – Proof that property φ holds initially: $T \models \varphi[Start]$.
- *Induction case* – When φ holds at some time point, it will also hold at the next time point: $T \cup \{\varphi[t]\} \models \varphi[Next(t)]$.

Proving these two entailments ensures that property φ always holds. It is important that φ is a single state formula since otherwise, we cannot split it easily into base and induction cases. A more detailed explanation of this approach can be found in the paper (Bogaerts et al., 2014).

Let's look at the theorems (properties) we would like to prove.

Theorem 4.1. *If the vehicle starts at the initial state $w = 0; y = 0; v = 0$, then the controller guarantees the vehicle never strays farther than three units from the y-axis.*

Theorem 4.2. *If the wind ever becomes constant for at least four sampling intervals, then the vehicle returns to the 0 of y-axis and stays there as long as the wind is still.*

Proving these theorems formally ensures that the vehicle appropriately stays on the course under each of the infinite number of possible wind change sequences. Let us prove our theorems.

4.5.2.4 Discussion and proof of Theorem 4.1

No matter how the wind behaves, we want to prove that if at the beginning the system is in the state $(y = 0, w = 0, v = 0)$, then the controller guarantees the vehicle never strays farther than three units from the y-axis, or more formally,

$$w(Start) = y(Start) = v(Start) = 0 \Rightarrow \forall t : -3 \leq y(t) \leq 3.$$

This theorem states that vehicle position is always in some range if some preconditions are met. The part that states conditions on each state looks like a suitable candidate for induction, but the problem is that the entire formula is not a single state.

For clarity, let's name *if part* of the theorem φ_{cond} (*condition*) and *then* part φ_{Inv_1} (*invariant*[6]). Now the theorem can be abbreviated as $\varphi_{cond} \Rightarrow \varphi_{Inv_1}$. Our goal is to show that $T_{vc} \models \{\varphi_{cond} \Rightarrow \varphi_{Inv_1}\}$. Using sequence calculus, we can transform this question to equivalent one $(T_{vc} \cup \{\varphi_{cond}\}) \models \{\varphi_{Inv_1}\}$. This transformation allows us to *merge* vehicle controller theory and *condition* of the theorem leaving single state formula on the right-hand side. Now the problem is entailment of a single state proposition, which allow us to use induction to prove it.

Induction proof starts with the *base case*, the goal is to show that initially vehicle position is in an adequate range, or formally $-3 \leq y(Start) \leq 3$. This is trivial since the start condition (φ_{cond}) is added to the main theory, and hence the vehicle position at the *start* time point is always 0.

The more difficult part is the induction case. Here, the goal is to show that at any time point t if the vehicle position is between 3 and –3, it will still be at time point $t + 1$ (this is known as induction hypothesis). As the whole idea is to eliminate time from the theory so we can use theorem provers, we must transform our theory to consider only two consecutive time points. This can be done by introducing constants instead of functions for vehicle position, wind speed, and vehicle speed (technical details are available in the provided formalizations). As these two time points stand for an arbitrary segment of time, the start constraints (φ_{cond}) are not applicable for the first time point (t). Since the first time point can be any possible state, the induction hypotheses start to sound a bit too optimistic. Consider a state $(y = 0, w = 100, v = 0)$, no matter how the

6 In the context of dynamic systems, "Invariant" denotes a single state property that always holds.

wind and vehicle speed change in the next time point, vehicle position will certainly be somewhere around 100. The problem is that connection with the starting point is lost, it is probably not possible to reach such a state from the state (0, 0, 0).

To handle this issue, we can strengthen the induction hypotheses by adding the information to it. In the paper (Boyer, 1990), the notion of a good state is introduced for this purpose. A good state is a class of states represented as a pair $(y, w + v)$ reachable from the state (0, 0, 0). The good states are represented in the Table 4.8.

Table 4.8: Good state— class of states reachable from the state (0, 0, 0).

y	–3	–2	–2	–1	–1	0	0	0	1	1	2	2	3
$w + v$	1	2	1	3	2	–1	0	1	–2	–3	–1	–2	–1

The new invariant would look like $\varphi_{Inv_2} = \forall t : GS(y(t), w(t) + v(t))$. Note that this invariant entails the old one, since in any good state vehicle position is always between 3 and –3. The new invariant is stronger because the first time point in the induction case must be a good state, and hence reachable from the initial state. The final shape of the entailment to be proven is represented in Table 4.9.

Table 4.9: Theorem 4.1— induction schema in first-order logic.[7]

First-order logic:
$T_{vc}[t] \cup T_{as}[t] \cup \{\varphi_{Inv2}[t]\} \models \{\varphi_{Inv2}[Next(t)]\}$ Where: – T_{vc} – Is a simple vehicle controller theory. – $T_{as} = \{\forall\, t : -1 \leq dw(t) \leq 1.\}$ – Assumption that wind change is between –1 and 1. – $T[t]$ – is a temporal theory applied to the time point t

We use the IDP system (De Cat et al., 2018) to provide specification of simple vehicle controller example using first-order logic. IDP is a knowledge base system for the FO(·) language[8] and it supports a multiple forms of inference methods, among others also proving invariants in dynamic system specifications.

7 Note that we didn't express the base case of induction since it is trivially true. Also, note that wind change constraint appears as an assumption in the theorems.

8 FO(·) is an extension of first-order logic with types, aggregates, inductive definitions, bounded arithmetic, etc.

Detailed proof procedure for Imandra is available here,[9] while the solution using the IDP system based on first-order logic can be found at this link.[10]

Good state discussion
There is something puzzling about the notion of a good state, namely how to compute it, and if we can compute it, does it not prove Theorem 4.1? The difference is in one subtle puzzle piece. To compute the set of good states, we can bound the system and use finite domains. Because of this restriction, the computed set is not enough to be considered as proof of Theorem 4.1. Therefore, induction comes in to prove that this set of good states is the maximal one.

One can try as an exercise to remove extremes from the good state relation and retry the induction proof.

4.5.2.5 Discussion and proof of Theorem 4.2

The second property to prove an autonomous vehicle controller is “Whenever the wind is constant for four consecutive time points, vehicle will be back to the course and will remain there as long the wind is constant.” This statement talks about at least five different states, which makes it harder to prove. Here, we explain some ideas on how to simplify this statement and how to prove it. We are not going into details about this theorem, rather we sketch how the proof can be constructed.

It is important to notice that the theorem is composed of *two* parts, first expressing that the vehicle will come back to the course after four consequent time points of steady wind, and second expressing that the vehicle remains there if wind remains steady. This suggests that the theorem can be split. The next two formulas stand for these new theorems expressed in first-order logic. Note that we abuse the notation and write $GS(t)$ where it should be $GS(y(t), w(t) + v(t))$. Also, $GSw(s)$ stands for the state after four consecutive time points of steady wind:

$$\forall s : GS(s) \land \big(\forall t : 0 \le t \le 4 \Rightarrow dw(s+t) = 0\big) \Rightarrow y(s+4) = 0$$
$$\forall s : GSw(s) \land dw(s) = 0 \Rightarrow y(s) = 0$$

Intuitively, after the vehicle comes back to the center, to stay there (if the wind does not change) the wind and vehicle speed should cancel each other. Speaking in terms of *good states*, this is the state (0,0), and we will refer to this state with $GS_0(t)$ for a time

9 https://docs.imandra.ai/imandra-docs/notebooks/simple-vehicle-controller/

10 The full solution is explained at https://djordje.rs/posts/svc.html. The raw IDP files can be retrieved from https://gist.github.com/dmkoder/6c39aa5768a9fb3305745f7f999285f4 and https://gist.github.com/dmkoder/2a1c564c7a3eba07b5b54f2ed3799e9a.

point t. So, the $GSw(s)$ in the second part of the theorem is actually $GS_0(t)$. To restore the connection between the two theorems, the consequent of the first part should be strengthened to $GS_0(t)$. The new theorems would look like the following:

$$\forall s : GS(s) \wedge \forall t : \big(0 \le t \le 4 \Rightarrow dw(s+t) = 0\big) \Rightarrow GS_0(s+4)$$
$$\forall s : GS_0(s) \wedge dw(s) = 0 \Rightarrow y(s) = 0$$

The second part satisfies all preconditions for induction to be applied, and hence we will not discuss it further here as it is the same as in Theorem 4.1. However, the first part still talks about four different time points relative to s. Keeping in mind that a *good state* stands for a class of states, quantification over states is a bit redundant here, and hence can be eliminated:

$$GS(Start) \wedge \forall t : \big(Start \le t \le 4 \Rightarrow dw(t) = 0\big) \Rightarrow GS_0(4)$$

Finally, we have a statement that talks about exactly four consecutive time points. Now we can drop the time (natural numbers) from the theory by simply introducing fresh new constants for each time point and defining the transition between each of them. This method is sometimes called forward-chaining.

This idea can be automated, and that is what the Imandra system is doing. We don't show details of this idea here, since they are too technical, but they are available in supplied full specifications of both Imandra and IDP solutions.

4.5.2.6 Conclusion

The focus of this section was on understanding the pragmatic importance of formal systems as first-order logic in industrial use cases. We have shown these on the example of the simple vehicle controller, but the same ideas could be applied to other dynamic systems, perhaps much more complex.

The first-order logic allowed us to go deep into the essence of problems of proving invariants of dynamic systems and to analyze them. It is important that in higher-level tools that we could provide more automated procedures for the problems that we have discussed; the underlying principles are the same as in this section. Hence, to be a profound user of such systems, one should keep these ideas in mind.

Bibliography

Abdelsalam Hisham M., Shoaeb Amal R. S., and Elassal Magy M. Enhancing decision model notation (DMN) for better use in business analytics (BA). In *Proceedings of the 10th International Conference on Informatics and Systems*, INFOS '16, page 321–322, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340625. https://doi.org/10.1145/2908446.2908514.

Baader Franz, Calvanese Diego, McGuinness Deborah L., Nardi Daniele, and Patel-Schneider Peter F., editors. The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, 2003. ISBN 0-521-78176-0. URL http://www.cambridge.org/asia/catalogue/catalogue.asp?isbn=9780521876254. Second edition, 2007.

Bogaerts Bart, Jansen Joachim, Bruynooghe Maurice, De Cat Broes, Vennekens Joost and Denecker Marc. Simulating dynamic systems using linear time calculus theories. Theory and Practice of Logic Programming, 14(4-5):477–492, 2014.

Boyer Robert S., Green Milton W. and Moore J Strother. The use of a formal simulator to verify a simple real time control program. In Beauty Is Our Business. Springer, 1990.

Cohn Anthony G. and Renz Jochen. Qualitative spatial representation and reasoning. In Handbook of Knowledge Representation van Harmelen et al. (2007), pages 551–596, 2007. ISBN 0444522115, 9780444522115. https://doi.org/10.1016/S1574-6526(07)03013-1.

De Cat Broes, Bogaerts Bart, Bruynooghe Maurice, Janssens Gerda and Denecker Marc. Predicate logic as a modeling language: the IDP system. In Declarative Logic Programming: Theory, Systems, and Applications (pp. 279–323), 2018.

Denecker Marc and Vennekens Joost. Building a knowledge base system for an integration of logic programming and classical logic. In María García de la Banda and Enrico Pontelli, editors, ICLP, volume 5366 of LNCS, pages 71–76. Springer, 2008. ISBN 978-3-540-89981-5. URL http://dx.doi.org/10.1007/978-3-540-89982-2_12.

Fagin Ronald, Halpern Joseph Y., Moses Yoram, and Vardi Moshe. Reasoning About Knowledge. MIT Press, 1995. URL http://library.books24x7.com.libproxy.mit.edu/toc.asp?site=bbbga&bookid=7008.

Gelfond Michael and Lifschitz Vladimir. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, ICLP/SLP, pages 1070–1080. MIT Press, 1988. ISBN 0-262-61056-6. URL http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.6050.

Governatori Guido. Representing business contracts in *RuleML*. International Journal of Cooperative Information Systems., 14(2-3):181–216, 2005. https://doi.org/10.1142/S0218843005001092.

Halpern Joseph Y. and Moses Yoram. Knowledge and common knowledge in a distributed environment. Journal of the ACM, 37(3):549–587, July 1990. ISSN 0004-5411. https://doi.org/10.1145/79147.79161.

Hintikka Jaakko. Knowledge and Belief. Ithaca: Cornell University Press, 1962.

Jackson Daniel. Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM'02), 11(2):256–290, 2002.

Leuschel Michael and Butler Michael. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, FME 2003: Formal Methods, LNCS 2805, pages 855–874. Springer-Verlag, 2003. ISBN 3-540-40828-2.

McCarthy John. Applications of circumscription to formalizing common-sense knowledge. Artificial Intelligence, 28(1):89–116, 1986.

Moore Robert C. Semantical considerations on nonmonotonic logic. Artificial Intelligence, 25(1):75–94, 1985. https://doi.org/10.1016/0004-3702(85)90042-6.

Mueller Erik T. Event calculus. In Handbook of Knowledge Representation van Harmelen et al. (2007), pages 671–708, 2007. ISBN 0444522115, 9780444522115. https://doi.org/10.1016/S1574-6526(07)03017-9.

Object Management Group. Semantics of business vocabulary and business rules (sbvr). OMG document number formal/08-01-02, January 2008. Version 1.0.

Pearl Judea. Causality: Models, Reasoning, and Inference. Cambridge University Press, 2000.

Pearl Judea and Mackenzie Dana. The Book of Why: The New Science of Cause and Effect. Basic Books, Inc., USA, 1st edition, 2018. ISBN 046509760X.

Reiter Raymond. Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, 2001. ISBN 9780262264310. URL http://books.google.be/books?id=exa4f6BOZdYC.

Reiter Raymond. A logic for default reasoning. Artificial Intelligence, 13(1-2):81–132, 1980. https://doi.org/10.1016/0004-3702(80)90014-4.

Reiter Raymond. The frame problem in situation the calculus: A simple solution (sometimes) and a completeness result for goal regression. In Artificial Intelligence and Mathematical Theory of Computation, pages 359–380. Academic Press Professional, Inc., San Diego, CA, USA, 1991. ISBN 0-12-450010-2.

van Harmelen Frank, Lifschitz Vladimir, and Porter Bruce. Handbook of Knowledge Representation. Elsevier Science, San Diego, USA, 2007. ISBN 0444522115, 9780444522115. URL https://www.elsevier.com/books/handbook-of-knowledge-representation/van-harmelen/978-0-444-52211-5.