MODULAR ACTION LANGUAGE $\mathcal{ALM}$ FOR DYNAMIC DOMAIN REPRESENTATION

by

Daniela Inclezan, M.S.

A Dissertation

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

Approved

Michael Gelfond
Chair of Committee

Vladimir Lifschitz

Richard Watson

Yuanlin Zhang

Peggy Gordon Miller
Dean of the Graduate School

August, 2012

To my family

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Gelfond, for everything he taught me during my years as a doctoral student. It was a pleasure to work with him on the topic of this dissertation. I appreciate all the hours he spent teaching me what it means to be a good researcher, how to think slowly and critically, how to present my ideas in a clear way, and how to apply all these things to my daily life. I believe that due to him I have learned to appreciate elegance and simplicity, and I have a better understanding of the ups and downs of life as a researcher. I liked our conversations on different topics and I especially enjoyed the stories he shared with me.

I am also grateful to the other members of my committee. Dr. Vladimir Lifschitz gave me numerous useful comments that contributed to the improvement of this work. His experience in designing a modular action language and his mathematical precision were of great help. I also received from him good advice on how to further test the capabilities of the language we designed. I also thank Dr. Richard Watson and Dr. Yualin Zhang for their feedback and questions.

I wish to express my appreciation to the past and present members of the Knowledge Representation Laboratory (KR Lab) at TTU for their support and friendship. I enjoyed the KR Lab seminars and I learned many things from them. In particular, I am grateful to Yana Todorova for being such a good friend and for making me feel like a member of her family. We shared many joyful moments together and hopefully we will create many more in the years to come. Many thanks to Yana's husband, Luis Cabrales, as well. I would also like to mention Justin and Jarred Blount for the interesting conversations we had; and Patrick Kahl for his questions regarding my work, which may have improved it.

This dissertation would not be possible without the support of my family: my parents, Estera Floarea and Alexandru Inclezan, my sister, Adela, and my brother, Alin. Although we lived apart for the last few years, knowing that they would always be there for me was priceless. I thank them deeply for all their love and encouragement.

Finally, I would like to thank my husband, Iñaki Prádanos García, for his love, motivation, and support during the process of writing this dissertation. His dedication, perseverance, and discipline while working on his own doctoral studies and afterwards were a great inspiration to me. I am happy to have him by my side and I hope this dissertation makes him proud.

Daniela Inclezan

Texas Tech University

August, 11, 2012

CONTENTS

ABSTRACT

The goal of this dissertation is to define a modular action language, $\mathcal{ALM}$, for the elaboration tolerant representation of knowledge about medium-size dynamic domains. Discrete dynamic domains can be theoretically captured by transition diagrams. The problem of concisely and precisely specifying such diagrams was the subject of study for several decades. Action languages were introduced as a solution to this problem. However, traditional action languages are not suitable for the specification of medium and large size domains because they lack the means for structuring knowledge and do not thoroughly address the problem of representing objects of the domain. Language $\mathcal{ALM}$ addresses these issues.

The task of designing a modular action language is a relevant one in the field of AI: it is the first step towards the creation of intelligent agents capable of reasoning about, or acting in a *wide variety* of dynamic environments. More importantly, the design of $\mathcal{ALM}$ allows us to better understand how to represent knowledge. Other modular action languages exist (e.g., MAD, TAL-C), but they correspond to different intuitions and knowledge representation styles. $\mathcal{ALM}$ is an alternative to these languages.

The design of $\mathcal{ALM}$ was not a trivial task. $\mathcal{ALM}$ is intended to be a simple but powerful language that would allow for elegant representations of a variety of dynamic domains. Hence, our main design and evaluation criteria were its expressive power and its simplicity and elegance. Generally, there is a tension between the two, which makes it challenging to design a modular action language.

Our language is intended to facilitate the creation of knowledge representation libraries, as well as the stepwise development, testing, and readability of a knowledge base. This is achieved by separating the description of a domain into two parts: a general theory, which consists of several modules that can be reused in modeling other domains, and an interpretation of this theory, which is particular to the specific domain. As well, $\mathcal{ALM}$ introduces classes of objects that can be described in terms of previously defined ones. Classes have attributes that are optional, a feature that

contributes to the elaboration tolerance of the language. We tested that these and other features of our language accomplish our goal by modeling several domains, both commonsensical (e.g., motion) and specialized (e.g., cell division), in $\mathcal{ALM}$. We were satisfied with the formalizations that resulted and with the capabilities of our language.

In this dissertation, we formally present language $\mathcal{ALM}$, illustrate the methodology of its use for knowledge representation and problem solving, report on an application of $\mathcal{ALM}$ to question answering, and compare our language with the closest existing modular action language, MAD.

LIST OF FIGURES

CHAPTER I
INTRODUCTION

One of the goals of Artificial Intelligence is to understand how to create software components for intelligent agents capable of reasoning about and acting in dynamic domains. It is generally assumed that, in order to exhibit intelligent behavior, an agent must be equipped with a mathematical model of the dynamic domain, together with some reasoning algorithms. Researchers studying the problem of modeling dynamic domains normally concentrated on domains that change because of actions and evolve in discrete steps. Such domains are called *discrete*.

Theoretically, discrete dynamic domains are described by transition diagrams whose nodes represent physical states of the domain and whose arcs are labeled by actions. A transition from a state σ_0 to a state σ_1 through an arc labeled by action a in a transition diagram says that "the execution of a in state σ_0 may take the system to state σ_1." A dynamic domain may often have a large, complex, and difficult to specify diagram, which makes it unsuitable for use in software systems. The problem of finding its concise and mathematically accurate description is not trivial and has been the subject of research for over 30 years. Action languages were introduced as one of the solutions to this problem. These are formal languages that describe the effects of actions and action preconditions in a syntax close to that of natural language.

Several action languages exist nowadays. They usually contain, in various degrees, solutions to long debated problems from the field of representing and reasoning about actions: the *frame problem* [McCarthy & Hayes, 1969] (how to describe what does *not* change as result of an action); the *ramification problem* [Finger, 1986] (how to describe the indirect effects of actions); and *the qualification problem* [McCarthy, 1977] (how to describe the preconditions of actions). Action languages sometimes differ in their underlying assumptions. For instance, the semantics of action language $\mathcal{AL}$

[Turner, 1997, Baral & Gelfond, 2000] relies on the Inertia Axiom [McCarthy & Hayes, 1969], which expresses the intuition that *"Normally things stay the same"*. Language $\mathcal{C}$ [Giunchiglia & Lifschitz, 1998], on the other hand, is based on the Causality Principle [McCain & Turner, 1997, Giunchiglia et al., 2004a], which says that *"Everything true in the world must be caused."*

Although traditional action languages represent a substantial advancement, they are unsuitable for representing knowledge about *large* dynamic domains, and that is due to two reasons. First of all, they *lack the means for structuring knowledge and for representing hierarchies of abstraction.* Such means are relevant for the design of knowledge bases and the creation of libraries. Secondly, traditional action languages *do not thoroughly address the issue of how to represent objects of the domain.* They do not possess the means for describing objects as special cases of other objects, which is a common practice in natural language (e.g., action "*carry*" is defined in the English dictionary as "*move while holding*", hence it is a special case of action "*move*"). As well, objects (including actions) are usually represented in the traditional approach as constants or terms, which is not always elaboration tolerant and does not allow for the specification of *optional* attributes of objects. Let us illustrate this point via the following example. Consider the action described by the sentence "John goes from London to Paris". It would normally be denoted in traditional action languages by a term, for instance $move(john, london, paris)$; axioms would be written about the effects and preconditions of this action. However, the origin of a "going to" action is not always relevant (it is what we call an optional attribute), as shown by the sentence "John goes to Paris". The optionality of the origin is not captured in the traditional approach. To denote this new action, one would not be able to use the previous term and instead would have to create a new one, $move(john, paris)$. New axioms would have to be written about this action, which is not elaboration tolerant.

The goal of this dissertation is to define a modular action language, $\mathcal{ALM}$, that remedies the above mentioned problems and allows for the elaboration tolerant representation of knowledge about large dynamic domains.

$\mathcal{ALM}$ is based on the basic underlying ideas of $\mathcal{AL}$. There are however several differences between the two languages. $\mathcal{ALM}$ introduces classes of objects with optional attributes. A class can be described in terms of other, previously defined, classes. Objects of the domain are instances of classes. The design features mentioned so far address the limitation of traditional action languages with respect to the representation of objects. To remedy the lack of structuring of knowledge in the traditional approach, in $\mathcal{ALM}$ we separate the declaration of classes from the definition of instances, organize classes in a specialization hierarchy, and structure knowledge into modules. A particular dynamic domain is represented in our language using what we call a *system description*. A system description consists of two parts: a *theory*, which is a collection of modules with a common theme organized in a hierarchy, and a *structure*, which is an interpretation, in the logical sense, of certain symbols in the theory. Syntactically, a *module* of our language is a collection of declarations of classes, declarations of functions, and axioms.

Other modular action languages preceded $\mathcal{ALM}$ and attempted to address some of the problems encountered in traditional action languages (e.g., MAD, TAL-C, Modular BAT). However, they are based on different intuitions and knowledge representation styles than ours. MAD [Lifschitz & Ren, 2006, Erdoğan & Lifschitz, 2006] is the closest language to ours as its goal is to facilitate the elaboration tolerant representation of knowledge about large dynamic domains. MAD and $\mathcal{ALM}$ differ in their underlying assumptions: MAD is an expansion of action language $\mathcal{C}$ and hence incorporates the Causality Principle, whereas $\mathcal{ALM}$ is inspired by $\mathcal{AL}$ and relies on the Inertia Axiom. Additionally, some of the features of MAD do not match our thought style. For instance, it is natural for us to separate classes from their instances conceptually; however, this distinction is not expressible in MAD. We prefer all concepts of a language to have some meaning, but in MAD not all modules have one: the only modules with a meaning are those which contain definitions of object constants for every sort. Finally, we believe that the user should have some freedom in the way he structures knowledge into modules. In MAD though, when a user wants to declare an

action class as a special case of another one, he is *required* to place the new action class in a separate module. We attempted to address such features of MAD in the design of our language and created $\mathcal{ALM}$ as an alternative that matches our intuitions. (In that respect, $\mathcal{ALM}$ is a successor of modular language $\mathcal{M}$ [Gelfond, 2006].) Although our decisions reflect our personal preferences in knowledge representation, we have reasons to believe that they are shared by a sufficient number of people to make them broadly appealing.

In the process of designing $\mathcal{ALM}$ we were driven by two criteria: *simplicity/elegance* and *expressive power*. Normally, there is a tension between the two, as increasing the expressive power of a language usually causes it to lose simplicity and elegance, and vice-versa. As a consequence, designing $\mathcal{ALM}$ was not a trivial task. This is demonstrated by the fact that the design of $\mathcal{ALM}$ was a two-step process in which we had an initial version that we later improved in the second version, as we will describe below. (In this dissertation we will present both versions, but we will give more attention to the current one.)

The more specific goals of $\mathcal{ALM}$ were to facilitate the stepwise development, testing, and readability of large knowledge bases, as well as the development of knowledge representation libraries. To verify that our language satisfies these goals, we used it to formalize libraries about various domains. We encoded in the first version of $\mathcal{ALM}$ a basic formalization of commonsense motion and used it to model some well-known problems from the field of representing and reasoning about actions and change (Monkey and Banana, Blocks World). We also modeled a specialized domain in $\mathcal{ALM}$: the biological phenomenon of cell division. We used this model in answering questions about cell division. In both cases, we were pleased with the capabilities of our language and with the representations it allowed us to create. In particular, we confirmed that separating the general theory from its interpretation was a good design decision, as it allowed us to represent the same domain (cell division) at different levels of granularity in an elaboration tolerant way. For that, we used the same theory and only modified the initial interpretation whenever we needed to add more

details. Success of these experiments increased our confidence in the suitability of the underlying ideas of the language.

Although we were generally satisfied with our design of $\mathcal{ALM}$, our practice in formalizing knowledge helped us notice some areas of possible improvement. For example, we realized that our initial syntax of axioms could be simplified to allow for a more elegant representation of certain information (e.g., conditions for the concurrent execution of two or more actions). We noticed that non-Boolean functions would be useful in representing properties of dynamic domains and expanded the original Boolean functions to arbitrary ones. Finally, we noticed that separating primitive sorts from action classes, as it was done in the original design and also in MAD, is not justified. Sorts other than action classes may have optional attributes too. For instance, when describing sequences, it seems natural to refer to the length and components of a sequence as its attributes. Hence, we decided to combine the initial basic concepts of sort and action class into a single concept, that of a class. We created a second version of $\mathcal{ALM}$ that incorporates these changes. The main features of our language remained unchanged: we still separate a general theory from its interpretation, knowledge is organized into modules, and classes can be declared in terms of previously defined ones. We tested this second version of $\mathcal{ALM}$ by expanding the initial formalization of commonsense motion and using its more general forms to solve different computational tasks: temporal projection, planning, question answering. While doing so, we focused on formulating a methodology of $\mathcal{ALM}$ both for the creation of knowledge bases and for its use in solving computational problems.

The remainder of this dissertation is structured as follows: in Chapter II we give some background on Answer Set Prolog and CR-Prolog, discrete dynamic domains, important problems in the field of reasoning about actions, and action languages. We then continue with the presentation of the current version of $\mathcal{ALM}$. (The initial version can be seen in Appendix D.) We informally describe the main features and concepts of our language in Chapter III. We present the signature of $\mathcal{ALM}$ in Chapter IV, give its formal syntax in Chapter V and its formal semantics in Chapter

VI, illustrated by examples. The methodology of representing knowledge in $\mathcal{ALM}$ is presented in Chapter VII. Here, we show how we formalized knowledge about the commonsense domain of motion and used it to represent some benchmark examples from the field of reasoning about actions. In Chapter VIII, we discuss the methodology of use of our language in solving various computational tasks, such as temporal projection, planning, diagnosis, and question answering. A case study of applying this methodology to question answering is given in Chapter IX. We compare $\mathcal{ALM}$ with related work, and with modular action language MAD in particular, in Chapter X. We end with conclusions and future work in Chapter XI.

CHAPTER II
BACKGROUND

In this chapter we present some background information that may be relevant to the reader. As the semantics of our language is defined in terms of Answer Set Prolog logic programs, we begin this chapter with a presentation of Answer Set Prolog in Section 2.1 and of its extension CR-Prolog in Section 2.2. We continue by describing in more detail, in Section 2.3, the type of dynamic domains we are addressing. In Section 2.4 we review some important problems in the field of representing and reasoning about actions and present some solutions to these problems. In Section 2.5 we talk about an established action language, $\mathcal{AL}$, which was the inspiration for our modular language, $\mathcal{ALM}$. We give a short description of another modular action language, MAD, in Section 2.6. This will be relevant when we compare $\mathcal{ALM}$ with MAD in Chapter X.

2.1 Answer Set Prolog

Answer Set Prolog (ASP) [Gelfond & Lifschitz, 1988, Gelfond & Lifschitz, 1991] is a declarative language for knowledge representation that emerged as a result of research in logic programming and nonmonotonic reasoning. It has an established methodology of use [Baral & Gelfond, 1994, Baral, 2003]. It is particularly suitable for the representation of dynamic domains due to its declarative and nonmonotonic features. For that reason, we chose to give the semantics of our modular language in terms of ASP logic programs. Moreover, there are several ASP solvers nowadays (e.g., CLASP [Gebser et al., 2007], DLV [Leone et al., 2006], SMODELS [Niemelä & Simons, 1997]). By translating $\mathcal{ALM}$ into ASP, we facilitate the use of $\mathcal{ALM}$ system descriptions in solving computational tasks.

In the remainder of this section, we give the syntax and semantics of Answer Set Prolog.

2.1.1 Syntax of Answer Set Prolog

The description of ASP appearing in this section is a short version of Chapter 2 of [Gelfond & Kahl, 2012].

A *signature* is a four-tuple $\Sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{P}, \mathcal{V} \rangle$ of disjoint sets, containing the names of the *objects*, *functions*, *predicates*, and *variables* used in the program. Function and predicate names are associated with an *arity* (i.e., a non-negative integer indicating the number of parameters), which is normally determined from the context. Following the Prolog convention, object, function, and predicate constants of Σ are denoted by identifiers starting with lower-case letters; variables are identifiers starting with capital letters.

Signatures may sometimes be extended with a fifth set containing the names of *sorts* used in a program. Sorts are useful in restricting the parameters of predicates or the parameters and values of functions. Such five-tuple signatures are called *sorted signatures*.

Terms over signature Σ are defined as follows:

1. Variables and object constants are terms.
2. If $t_1, \ldots, t_n$ are terms and f is a function symbol of arity n then $f(t_1, \ldots, t_n)$ is a term.

For simplicity arithmetic terms will be written in the standard mathematical notation; i.e. we will write $2 + 3$ instead of $+(2, 3)$.

An *atom* is an expression of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ and $t_1, \ldots, t_n$ are terms.

A *literal* its an atom, $p(t_1, \ldots, t_n)$, or its negation, $\neg p(t_1, \ldots, t_n)$.

A term is called *ground* if it contains no variables and no symbols for arithmetic functions. Similarly for atoms and literals. For instance, $1 + 3$ is not a ground term, and neither is $f(X, Y)$.

A program Π of ASP consists of a signature Σ and a collection of *rules* of the form:

$$l_0 \ or \ \ldots \ or \ l_i \leftarrow l_{i+1}, \ldots, l_m, not \ l_{m+1}, \ldots, not \ l_n$$

where l's are literals of Σ.

The symbol *not* is a logical connective called *default negation*, (or *negation as failure*); *not* l is often read as *"It is not believed that* l *is true."* The disjunction *or* is also a connective, sometimes called *epistemic disjunction*. The statement l_1 *or* l_2 is often read as *"*l_1 *is believed to be true or* l_2 *is believed to be true."*

The left-hand side of an ASP rule is called the *head* and the right-hand side is called the *body*. A rule with the empty head is often referred to as a *constraint* and written as:

$$\leftarrow l_{i+1}, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n.$$

A rule with the empty body is often referred to as a fact and written as

$$l_0\ or\ \ldots\ or\ l_i.$$

A rule r with variables is viewed as the set of its possible ground instantiations – rules obtained from r by replacing r's variables by ground terms of Σ and by evaluating arithmetic terms (e.g. replacing $1 + 3$ by 4). Let us define what it means for a set of ground literals to satisfy a rule. We introduce the following notation. Given a rule r,

$$head(r) = \{l_0, \ldots, l_i\}$$
$$pos(r) = \{l_{i+1}, \ldots, l_m\}$$
$$neg(r) = \{l_{m+1}, \ldots, l_n\}$$

A ground set S of literals *satisfies* a rule r (or *is closed under* r) if any of the following conditions holds:

$$pos(r) \nsubseteq S$$
$$neg(r) \cap S \neq \emptyset$$
$$head(r) \cap S \neq \emptyset$$

2.1.2 Semantics of Answer Set Prolog

First, we will give the informal semantics of ASP and then its formal semantics.

Informal Semantics

A program Π can be viewed as a specification for *answer sets* – sets of beliefs that could be held by a rational reasoner associated with Π. In forming such sets the reasoner must be guided by the following informal principles:

1. Satisfy the rules of Π. In other words, if one believes in the body of a rule, one must also believe in its head.
2. Do not believe in contradictions.
3. Adhere to the *rationality principle* which says: *"Believe nothing you are not forced to believe."*

Formal Semantics

We start by defining consistent sets of literals. A set S of ground literals is called *consistent* if it does not contain both an atom a and its negation $\neg a$. We continue with the definition of an answer set, which is given in two parts: the first part is for programs without default negation and the second part explains how to remove default negation so that the first part can be applied.

Definition 1. [Answer Sets, Part I] Let program Π consist of rules of the form:

$$l_0 \ or \ \ldots \ or \ l_i \leftarrow l_{i+1}, \ldots, l_m.$$

An *answer set* of Π is a consistent set S of ground literals such that:

- S satisfies the rules of Π.
- S is minimal, i.e., there is no proper subset of S which satisfies the rules of Π.

Definition 2. [Answer Sets, Part II] Let Π be an arbitrary program and S be a set of ground literals. By Π^S we denote the program obtained from Π by:

1. removing all rules containing *not* l such that $l \in S$;
2. removing all other premises containing *not* .

S is an *answer set* of Π if S is an answer set of Π^S.

2.2 CR-Prolog

CR-Prolog [Balduccini & Gelfond, 2003b] is an extension of ASP by means for expressing rare events (or unexpected exceptions to defaults). A signature, a term, an atom, and a literal have the same definitions as in Section 2.1.1.

A program Π of CR-Prolog consists of a signature Σ and a collection of rules of the form:

$$l_0 \ or \ \ldots \ or \ l_i \leftarrow l_{i+1}, \ldots, l_m, not \ l_{m+1}, \ldots, not \ l_n \tag{2.1}$$

$$l_0 \ or \ \ldots \ or \ l_i \stackrel{+}{\leftarrow} l_{i+1}, \ldots, l_m, not \ l_{m+1}, \ldots, not \ l_n \tag{2.2}$$

where l's are literals of Σ. Rules of the type 2.1 are called *regular*; rules of the type 2.2 are called *consistency restoring* or *cr-rules*. Informally, a cr-rule says "if you believe $l_{i+1}, \ldots, l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then believe one of the literals $l_0, \ldots, l_i$, but only if strictly necessary, meaning if there is no other way to obtain a consistent set of beliefs."

The set of regular rules of Π is denoted by Π^r, the set of cr-rules by Π^{cr}. The regular rule obtained from a cr-rule r by replacing the symbol $\stackrel{+}{\leftarrow}$ by the symbol $\leftarrow$ is denoted by $\alpha(r)$; the notation α is extended to sets of cr-rules in the standard way.

The semantics of a CR-Prolog program is defined in two steps as follows:

Definition 3. [Abductive Support] A minimal (with respect to set-theoretic inclusion) collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent is called an *abductive support of* Π.

Definition 4. [Answer Set] A set of literals A is an *answer set* of Π if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.

2.3 Discrete Dynamic Domains

We are interested in representing knowledge about dynamic domains in which change is caused by the execution of actions. In this dissertation, we limit ourselves to dynamic domains satisfying the following conditions:

- the evolution of the domain is followed over a sequence of discrete time steps
- states of the domain are characterized by an assignment of values to functions denoting the relevant properties of the domain
- actions occur instantaneously and their effects appear at the next time step.

Generally, such domains are called *discrete*. The difference between the domains considered here and the discrete dynamic domains addressed in [Balduccini, 2005], for example, is that in [Balduccini, 2005] all relevant properties of a domain are Boolean, whereas we consider relevant properties with *arbitrary ranges*. Also, note that slight modifications to our language would allow for the representation of other types of domains, for example *hybrid* domains, which allow both discrete and continuous change [Chintabathina et al., 2005, Chintabathina, 2010].

Dynamic domains of the type considered here can be modeled by transition diagrams like the one in Figure 2.1. This particular transition diagram describes a domain in which a person, John, moves between different cities: Paris, London, and Rome. The relevant property of the domain is the location of John at different time steps, denoted by the function named loc_in. States of this domain are assignments of values to the function loc_in for its input, $john$. We have three possible states: in one of them John is located in Paris, in another one he is located in London, and in the third in Rome. The labels of arcs of this transition diagram denote actions that can be performed in this domain, e.g., $move(john, paris)$ stands for the action of John moving to Paris, etc. Arcs from one state to the other indicate that, for instance, the consequence of executing action $move(john, london)$ in a state in which $loc\_in(john) = paris$ is that $loc\_in(john) = london$.

Formally, a transition diagram is a directed graph. Its nodes represent states of the domain and its arcs are labeled by actions. The existence of a transition $\langle \sigma_0, a, \sigma_1 \rangle$ from state σ_0 to state σ_1 via an arc labeled by a says that "the occurrence of action a in σ_0 may take the system to state σ_1." The use of the word "may" is justified by the possible existence of non-deterministic actions: actions whose execution may take

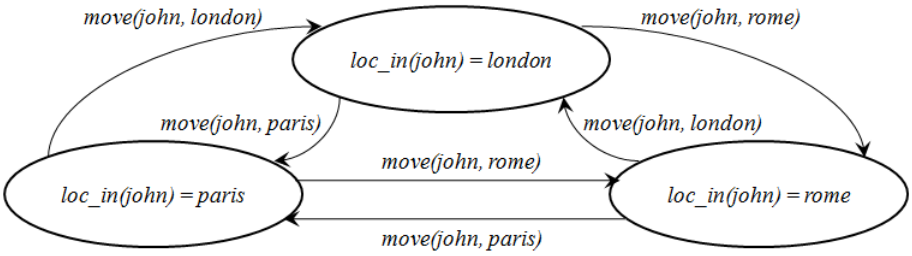

Figure 2.1: Transition Diagram of a Dynamic Domain

the system into one of many states. An example of a non-deterministic domain can be seen later in Example 2.

2.4 Important Problems in Representing and Reasoning about Actions

A substantial part of the artificial intelligence community is concerned with the representation of and commonsense reasoning about actions using logical formalisms. In the initial years of this field, researchers attempted to formalize toy domains, such as the Yale Shooting domain [Hanks & McDermott, 1987], Blocks World, or Lin's Suitcase domain [Lin, 1995]. These simple scenarios played a significant role in identifying important difficulties in reasoning about actions.

The *frame problem*, first mentioned by McCarthy and Hayes in [McCarthy & Hayes, 1969], points to the challenge of succinctly describing what does not change as a result of an action being executed. For instance, in the Yale Shooting domain loading the gun does not affect the state of the turkey, which continues to be alive if it was alive before loading the gun or dead if it was already dead. For domains with a large number of relevant properties, listing all the properties that are not affected by the execution of an action is impractical. In the area of nonmonotonic logic, solutions to the frame problem were developed based on the commonsense law of inertia: *"Normally things stay the same"*.

The *ramification problem* [Finger, 1986] refers to the difficulty of describing the indirect effects of actions. For example, in Lin's Suitcase domain, the indirect effect of unfastening both latches is that the suitcase becomes open. The number of indirect

effects of an action can increase rapidly, as an indirectly affected property of the domain can affect other properties and so on. A solution to the ramification problem for action languages was proposed by McCain and Turner in [McCain & Turner, 1995]. They introduced statements called "static causal laws" to express relations between properties of the domain. This idea was incorporated in action languages such as $\mathcal{AL}$ or $\mathcal{C}$.

The *qualification problem* [McCarthy, 1977] is the counterpart of the ramification problem. It focuses on the difficulty of enumerating all the preconditions for the execution of an action. For instance, in the Blocks World domain, a block b_1 cannot be moved atop of a block b_2 if there is some block on top of b_1 or on top of b_2. Also, two blocks cannot be moved at the same time. Solutions to the qualification problem were derived from the solutions to the ramification problem.

Solving these important challenges represented an important contribution to the field of reasoning about actions, which opened the way for the development of action languages.

2.5 Traditional Action Languages

Action languages, as described in [Gelfond & Lifschitz, 1998], are formal languages that specify the effects of actions and action preconditions in a syntax close to that of natural language. In such languages, properties of the domain that may be changed by actions are called *fluents*. We will refer to an action language as *traditional* if it does not address the higher-level structuring of knowledge about a domain, i.e., if it is not modular.

A predecessor of action languages is the STRIPS action description language [Fikes & Nilsson, 1971]. In STRIPS, properties of the domain are represented by first-order formulas. Collections of such formulas are called "world models". Intuitively, world models are similar to states but may be incomplete. Actions are described via three lists: a list of properties that must hold in the world model in which the action is executed, a list of properties to add to the successor model with

respect to the current one, and a list of properties to delete from the current world model. The language ADL [Pednault, 1987] expands STRIPS by allowing conditional "add" and "delete" lists. The main drawback of STRIPS and ADL is that they cannot express indirect effects of actions. The more frequently used traditional action languages generally address this problem.

In what follows, we briefly present action language $\mathcal{AL}$ extended with defined fluents, which served as an inspiration for $\mathcal{ALM}$.

2.5.1 Action Language $\mathcal{AL}$ with Defined Fluents

$\mathcal{AL}$ [Turner, 1997, Baral & Gelfond, 2000] is an action language based on previous languages $\mathcal{A}$ and $\mathcal{B}$ [Gelfond & Lifschitz, 1998]. $\mathcal{A}$ corresponds to the propositional fragment of ADL; $\mathcal{B}$ expands $\mathcal{A}$ by the addition of "static causal laws", which allow for the description of indirect effects of actions; $\mathcal{AL}$ expands $\mathcal{B}$ by the addition of "executability conditions". In [Gelfond & Inclezan, 2009], $\mathcal{AL}$ is extended by "defined fluents" – properties of the domain that are subject to the Closed World Assumption [Reiter, 1978]. The semantics of $\mathcal{AL}$ incorporates the law of inertia for inertial fluents. The description of $\mathcal{AL}$ that follows is based on Chapter 7 of [Gelfond & Kahl, 2012].

A *system description* of $\mathcal{AL}$ consists of a *signature* and a collection of *axioms*. The signature consists of three non-empty and disjoint sets of names for *actions*, *statics*, and *fluents*. Together, statics and fluents are called *domain properties*. Fluents are partitioned into *inertial fluents* and *defined fluents*. The truth values of statics cannot be changed by actions. Inertial fluents can be changed by actions and are subject to the law of inertia. Defined fluents are non-static fluents which are defined in terms of other fluents. They can be changed by actions but only indirectly.

A *domain literal* is a domain property f or its negation $\neg f$. Depending on the type of domain property forming a domain literal we will use the terms *static*, *inertial*, and *defined literal*.

Direct causal effects of actions are described in $\mathcal{AL}$ by *dynamic causal laws* –

statements of the form:

$$a \textbf{ causes } l \textbf{ if } p \tag{2.3}$$

where a is an action, l is an inertial literal, and p is a collection of arbitrary domain literals. (2.3) says that if action a were executed in a state satisfying p then l would be true in a state resulting from this execution.

Dependencies between fluents are described by *state constraints* — statements of the form:

$$l \textbf{ if } p \tag{2.4}$$

where l is a domain literal and p is a set of domain literals, with the restriction that l cannot be a negated *defined* literal. (2.4) says that every state satisfying p must satisfy l.

Executability conditions of $\mathcal{AL}$ are statements of the form:

$$\textbf{impossible } a_0, \ldots, a_k \textbf{ if } p \tag{2.5}$$

where $a_0, \ldots, a_k$ are actions with $k \geq 0$ and p is a collection of domain literals. The statement says that actions $a_0, \ldots, a_k$ cannot be executed together in any state which satisfies p. We refer to l as the head of the corresponding rule and to p as its body. The collection of state constraints whose head is a defined fluent f is often referred to as the *definition of* f. Similarly to logic programming definitions, f is true in a state σ if the body of at least one of its defining constraints is true in σ. Otherwise, f is false.

The semantics of $\mathcal{AL}$ are given by defining the states and transitions of the transition diagram described by a system description. Some preliminary definitions: a set σ of literals is called *complete* if for any domain property f either f or $\neg f$ is in σ; σ is called *consistent* if there is no f such that $f \in \sigma$ and $\neg f \in \sigma$. The definition of the transition relation $\langle \sigma_0, a, \sigma_1 \rangle$ of $\mathcal{T}(\mathcal{D})$ *is based on the notion of an answer set of a logic program.* For that purpose, a program $\Pi(\mathcal{D})$ is constructed from the logic programming encodings of statements from $\mathcal{D}$. The answer sets of the union of $\Pi(\mathcal{D})$

with the encodings of a state σ_0 and an action a determine the states into which the system can move after the execution of a in σ_0.

The signature of $\Pi(\mathcal{D})$ will consist of: (a) names from the signature of $\mathcal{D}$; (b) two new sorts: *steps* with two constants, 0 and 1, and *fluent_type* with constants *inertial* and *defined*; and (c) the relations: $holds(fluent, step)$ ($holds(f, i)$ says that fluent f is true at step i), $occurs(action, step)$ ($occurs(a, i)$ says that action a occurred at step i), and $fluent(fluent\_type, fluent)$ ($fluent(t, f)$ says that f is a fluent of type t). If l is a domain literal formed by a fluent, $h(l, i)$ denotes $holds(f, i)$ if $l = f$ or $\neg holds(f, i)$ if $l = \neg f$. Otherwise $h(l, i)$ is simply l. If e is a set of actions, $occurs(e, i) = \{occurs(a, i) : a \in e\}$. If p is a set of literals, $h(p, i) = \{h(l, i) : l \in p\}$.

Definition of $\Pi(\mathcal{D})$

1. For every static causal law "l **if** p" from $\mathcal{D}$, $\Pi(\mathcal{D})$ contains:

$$h(l, I) \leftarrow h(p, I). \tag{2.6}$$

2. For every dynamic causal law "a **causes** l **if** p" from $\mathcal{D}$, $\Pi(\mathcal{D})$ contains:

$$\begin{aligned} h(l, I+1) \leftarrow\ & h(p, I), \\ & occurs(a, I). \end{aligned} \tag{2.7}$$

3. For every executability condition "**impossible** $a_0, \ldots, a_k$ **if** p" from $\mathcal{D}$, $\Pi(\mathcal{D})$ contains:

$$\neg occurs(a_0, I) \vee \cdots \vee \neg occurs(a_k, I) \leftarrow h(p, I). \tag{2.8}$$

4. $\Pi(\mathcal{D})$ contains the Inertia Axioms:

$$\begin{aligned} holds(F, I+1) \leftarrow\ & fluent(inertial, F), \\ & holds(F, I), \\ & not\ \neg holds(F, I+1). \\ \neg holds(F, I+1) \leftarrow\ & fluent(inertial, F), \\ & \neg holds(F, I), \\ & not\ holds(F, I+1). \end{aligned} \tag{2.9}$$

5. $\Pi(\mathcal{D})$ contains the Closed World Assumption for defined fluents:

$$\begin{aligned} \neg holds(F, I) \quad \leftarrow \quad & fluent(defined, F), \\ & not\ holds(F, I). \end{aligned} \tag{2.10}$$

6. $\Pi(\mathcal{D})$ contains the constraint:

$$\begin{aligned} \leftarrow \quad & fluent(inertial, F), \\ & not\ holds(F, I), \\ & not\ \neg holds(F, I). \end{aligned} \tag{2.11}$$

7. For every static fluent f, $\Pi(\mathcal{D})$ contains the constraint:

$$\begin{aligned} \leftarrow \quad & not\ f, \\ & not\ \neg f. \end{aligned} \tag{2.12}$$

Let $\Pi_c(\mathcal{D})$ be a program constructed by rules (2.6), (2.10), (2.11), and (2.12) above. For any set σ of domain literals, σ_{nd} denotes the collection of all inertial and static literals in σ. $\Pi_c(\mathcal{D}, \sigma)$ is obtained from $\Pi_c(\mathcal{D}) \cup h(\sigma_{nd}, 0)$ by replacing I by 0.

Definition 5. [State] A set σ of literals is a *state* of $\mathcal{T}(\mathcal{D})$ if $\Pi_c(\mathcal{D}, \sigma)$ has a unique answer set, A, and $\sigma = \{l : h(l, 0) \in A\}$.

Now let σ_0 be a state and e a collection of actions.

$$\Pi(\mathcal{D}, \sigma_0, e) =_{def} \Pi(\mathcal{D}) \cup h(\sigma_0, 0) \cup occurs(e, 0)\ .$$

Definition 6. [Transition] A *transition* $\langle \sigma_0, e, \sigma_1 \rangle$ is in $\mathcal{T}(\mathcal{D})$ iff $\Pi(\mathcal{D}, \sigma_0, e)$ has an answer set A such that $\sigma_1 = \{l : h(l, 1) \in A\}$.

The semantics of system descriptions that do not contain defined fluents is equivalent to the semantics described in [Baral & Lobo, 1997, McCain & Turner, 1995]. As far as we know, [Baral & Lobo, 1997] is the first work which uses ASP to describe the semantics of action languages. Some mathematical properties of $\mathcal{AL}$ extended with defined fluents are presented in Appendix A.

We illustrate the syntax and semantics of $\mathcal{AL}$ by representing Lin's Suitcase domain.

Example 1. [Lin's Suitcase [Lin, 1995]] The signature of this domain consists of sort $latch = \{l_1, l_2\}$, action $toggle(latch)$, and inertial fluents $up(latch)$ and $open$. The system description $\mathcal{D}_{Lin}$ defining this domain consists of axioms:

$toggle(L)$ **causes** $up(L)$ **if** $\neg up(L)$

$toggle(L)$ **causes** $\neg up(L)$ **if** $up(L)$

$open$ **if** $up(l_1), up(l_2)$.

where L ranges over sort $latch$. Strictly speaking, $\mathcal{D}_{Lin}$ is not a system description of $\mathcal{AL}$, as the first two axioms contain variables. We call such axioms *schemas*. An $\mathcal{AL}$ system description is obtained from $\mathcal{D}_{Lin}$ by grounding its variables, i.e., by replacing the variable L by its possible values, l_1 and l_2 as follows:

$toggle(l_1)$ **causes** $up(l_1)$ **if** $\neg up(l_1)$

$toggle(l_2)$ **causes** $up(l_2)$ **if** $\neg up(l_2)$

$toggle(l_1)$ **causes** $\neg up(l_1)$ **if** $up(l_1)$

$toggle(l_2)$ **causes** $\neg up(l_2)$ **if** $up(l_2)$

$open$ **if** $up(l_1), up(l_2)$.

The system contains the transitions:

$\langle \{\neg up(l_1), up(l_2), \neg open\}, toggle(l_1), \{up(l_1), up(l_2), open\}\rangle$,

$\langle \{up(l_1), up(l_2), open\}, toggle(l_1), \{\neg up(l_1), up(l_2), open\}\rangle$, *etc.*

Note that a set $\{up(l_1), up(l_2), \neg open\}$ is not a state of our system.

$\mathcal{AL}$ is capable of representing not only deterministic domains like the one above, but also non-deterministic domains. A known example is given below.

Example 2. [Non-determinism] Let us consider a domain described by the system description:

$$a \quad \textbf{causes} \quad f$$

$$\neg g_1 \quad \textbf{if} \quad f, g_2$$

$$\neg g_2 \quad \textbf{if} \quad f, g_1$$

where f, g_1, and g_2 are inertial fluents and a is an action. The execution of action a in a state $\sigma_0 = \{\neg f, g_1, g_2\}$ may take the system into one of the states: $\sigma_1 = \{f, g_1, \neg g_2\}$ or $\sigma_1' = \{f, \neg g_1, g_2\}$.

2.6 Modular Action Description (MAD) Language

MAD [Lifschitz & Ren, 2006, Erdoğan & Lifschitz, 2006] is a modular action language based on language $\mathcal{C}+$ [Giunchiglia et al., 2004a], which is an extension of language $\mathcal{C}$ [Giunchiglia & Lifschitz, 1998]. As such, MAD incorporates the Causality Principle stating that *"Everything true in the world must be caused"* [McCain & Turner, 1997, Giunchiglia et al., 2004a]. The goal of MAD is to allow for the elaboration tolerant creation of general-purpose libraries of knowledge.

A dynamic domain is described in MAD by an *action description*, which consists of a set of sort declarations (with their subclass relations specified) and a set of *modules*. A module of MAD is a collection of declarations of fluents, declarations of actions, and axioms. Optionally, a module may also contain definitions of objects. A module M of an action description AD may import the knowledge contained by other modules of AD, while possibly restricting the imported knowledge to the context and signature of M. The modular structure of MAD and its import statements address the issue of elaboration tolerance in the design of libraries of knowledge.

We will illustrate the syntax and semantics of MAD by some examples. Note that in MAD, constant symbols that start with a letter have this letter capitalized and variable symbols start with a lower case letter.

Example 3. [Action Description] The following is a formalization in MAD of the Monkey and Bananas domain described in [Giunchiglia et al., 2004a]. This action description is taken from [Lifschitz & Ren, 2006] and adapted to the syntax of MAD presented in [Erdoğan, 2008]. Note that the expression $Location(Thing) : simple(Place)$ below says that *Location* is a *simple* fluent with an argument from sort *Thing* and ranging over sort *Place*.

sorts

$Thing;\ Place;$

module $MOVE;$

actions

$Move(Thing, Place);$

fluents

$Location(Thing) : simple(Place);$

variables

$x : Thing;\ p : Place;$

axioms

inertial $Location(x);$

exogenous $Move(x, p);$

$Move(x, p)$ **causes** $Location(x) = p;$

nonexecutable $Move(x, p)$ **if** $Location(x) = p;$

sorts

$Thing;\ Place;\ Level;$

module $MONKEY;$

objects

$Monkey, Box : Thing;$

$P_1, P_2 : Place;$

$BoxTop, Floor : Level;$

actions

$Walk(Place);$

$ClimbOn;$

$ClimbOff;$

fluents

$OnBox : simple;$

variables

$x : Thing;\ p : Place;\ l : Level;$

import $MOVE;$

$Move(Monkey, p)$ **is** $Walk(p);$

import $MOVE;$

$Place$ **is** $Level;$

$Location(Monkey) = l$ **is**

case $l = BoxTop : OnBox;$

case $l = Floor : \neg OnBox;$

$Move(Monkey, l)$ **is**

case $l = BoxTop : ClimbOn;$

case $l = Floor : ClimbOff;$

axioms

$Location(Monkey) = p$ **if** $OnBox \wedge Location(Box) = p;$

nonexecutable $ClimbOn$ **if** $Location(Monkey) \neq Location(Box);$

nonexecutable $ClimbOff \wedge Walk(p);$

The first import statement in module $MONKEY$ says that the declarations and axioms of module $MOVE$ are inherited with $Move(Monkey, p)$ renamed as $Walk(p)$. The second import statement indicates that module $MOVE$ is inherited again, this time with its sort $Place$ renamed as $Level$, its fluent $Location(Monkey)$ renamed as either $OnBox$ or $\neg Onbox$, and its action $Move(Monkey, l)$ renamed as either $ClimbOn$ or $ClimbOff$, depending on the value of l.

The semantics of an action description of MAD are defined by first translating it into a single-module description, which is then translated into an action description of $\mathcal{C}+$. Note that not all action descriptions of MAD have a meaning. The only action descriptions with a meaning are those that contain definitions of objects for every declared sort.

Example 4. [Generating a Single-Module Action Description] According to the

semantics of MAD described in [Lifschitz & Ren, 2006, Erdoğan, 2008], the action description in Example 3 is transformed into a single-module that looks as follows:

sorts

$Thing$; $Place$; $Level$;

module $MONKEY$;

objects

$Monkey, Box : Thing$;

$P_1, P_2 : Place$;

$BoxTop, Floor : Level$;

actions

$Walk(Place)$;

$ClimbOn$;

$ClimbOff$;

$I1.Move(Thing, Place)$;

$I2.Move(Thing, Level)$;

fluents

$OnBox : simple$;

$Location(Thing) : simple(Place)$;

$I2.Location(Thing) : simple(Level)$;

variables

$x : Thing$; $p : Place$; $l : Level$;

$I1.x : Thing$; $I1.p : Place$;

$I2.x : Thing$; $I2.p : Level$;

axioms

$Location(Monkey) = p$ **if** $OnBox \wedge Location(Box) = p$;

nonexecutable $ClimbOn$ **if** $Location(Monkey) \neq Location(Box)$;

nonexecutable $ClimbOff \wedge Walk(p)$;

$I1.Move(x, p) \equiv Walk(p) \wedge x = Monkey$;

inertial $Location(I1.x)$;

exogenous $I1.Move(I1.x, I1.p)$;

$I1.Move(I1.x, I1.p)$ **causes** $Location(I1.x) = I1.p$;

nonexecutable $I1.Move(I1.x, I1.p)$ **if** $Location(I1.x) = I1.p$;

$$I2.Location(x) = l \equiv ((x = Monkey \wedge OnBox) \wedge l = BoxTop) \vee (\neg(x = Monkey \wedge OnBox) \wedge l = Floor);$$

$$I2.Move(x, l) \equiv (x = Monkey \wedge l = BoxTop \wedge ClimbOn) \vee (x = Monkey \wedge l = Floor \wedge ClimbOff);$$

inertial $I2.Location(I2.x)$;

exogenous $I2.Move(I2.x, I2.p)$;

$I2.Move(I2.x, I2.p)$ **causes** $I2.Location(I2.x) = I2.p$;

nonexecutable$I2.Move(I2.x, I2.p)$ **if** $I2.Location(I2.x) = I2.p$;

The $\mathcal{C}+$ action description corresponding to an action description of MAD is obtained by grounding the axioms of the single-module description produced in the preliminary step.

CHAPTER III
INFORMAL DESCRIPTION OF $\mathcal{ALM}$

In this chapter, we give an intuitive description of the basic concepts of $\mathcal{ALM}$, which will be formally defined in the next chapter. We illustrate these concepts by some short examples. The basic concepts of $\mathcal{ALM}$ are the following ones: *class*, *instance of a class*, *function* defined on instances of classes, *module*, *theory*, *structure*, and *system description*.

3.1 Class, Instance of a Class, Function

We start by presenting the basic concepts of $\mathcal{ALM}$ that correspond to a lower level of abstraction.

For the reader familiar with traditional action languages such as $\mathcal{AL}$, we need to make some preliminary clarifications. Such action languages operate with three basic concepts: *object*, *action*, and *property* of an object or objects. In $\mathcal{ALM}$, we do not distinguish between objects and actions; we use the term *objects* for both objects and actions, in the traditional sense. Moreover, we are interested in the inherent properties of objects, which we call *attributes*. We define an *instance* as an object together with the values associated with its attributes.

Example 5. [Instance] Let a be an object (in the $\mathcal{ALM}$ sense) denoting an action: the movement of $John$ from a point in space, $Paris$, to another point, $London$. The attributes of a are the actor of action a, its origin, and its destination, which have the following values: $John$, $Paris$, and $London$, respectively. We say that a together with the values of its attributes ($John$, $Paris$, and $London$) is an instance. Note that $John$, $Paris$, and $London$ are also instances.

A *class* is a collection of instances that share common attributes and obey common rules. Instances belonging to a class c will be called *instances of class c*. We use the term *constraints* to denote the common rules. Classes describing a dynamic

domain are organized into a specialization hierarchy. The class that is probably most important for action languages is that of *actions*.

Example 6. [Class] We can group all objects that refer to motion actions and have the attributes *actor*, *origin*, and *destination* into a class called *move*. Such a class would be a special case of the class *actions*. Object *a* from Example 5 would be an instance of class *move*, and also of class *actions*. Similarly, we can group points in space into a class called *points* and instances like *John* in a class *movers*, which will be a special case of a more general class *things*.

Functions defined on instances of classes are of two types: (1) functions defining inherent properties of instances of classes and (2) functional relations between instances of classes. We already mentioned that we refer to the functions in (1) as *attributes*. For simplicity, we assume that the values of attributes cannot be changed. The functions in (2) are divided into: *statics* and *fluents*. The values of statics cannot be changed by actions. Fluents may be changed by actions, but otherwise maintain their values unchanged. This statement characterizing fluents is a default. Depending on the means used to represent this default, we divide fluents into *inertial fluents*, which are subject to the Law of Inertia [McCarthy & Hayes, 1969], and *defined fluents*, which, in any state, can only take values that meet their definition and are otherwise subject to the Closed World Assumption [Reiter, 1978].

Example 7. [Functions] The attribute *actor* of class *move* from Example 6 is a function: it maps instances of class *move* into instances of class *movers*. For example, it maps instance *a* of *move* into instance *John* of *movers*. Similarly, *origin* and *destination* map instances of *move* into instances of *points*.

We can consider a non-attribute function defined on classes from Example 6: *loc_in* mapping *things* into *points* and intuitively specifying the (unique) locations of things. As normally the locations of things may change over time, *loc_in* would be a fluent, most probably an inertial fluent.

3.2 Module, Theory, Structure, System Description

At a higher level of abstraction than that of classes, instances of classes, and functions, we have the basic concept of a *module*. A module describes general knowledge about a fragment of our model of the world. It may reuse information from already existing modules. If we were to compare $\mathcal{ALM}$ with procedural languages, a module would correspond to a procedure. Modules that are very general can be stored in libraries. Normally, modules are small and do not fully describe a domain, but can be combined in different ways and can be reused in the representation of several domains.

Example 8. [Module] Building upon Examples 5, 6, and 7, we can envision a module describing how movement affects the locations of things. Let us call this module *move_between_points*. Another module, reusing information from *move_between_points*, may describe how carrying things from one point in space to another affects the locations of things. We call this second module *carrying_things*.

The main concept of $\mathcal{ALM}$ is that of a *theory*. A theory bundles together multiple modules with a common theme. The resulting model of the world can be used in different applications. The symbols in a theory do not have a fixed interpretation in the standard logical sense (with the exception of some pre-interpreted symbols). This is what makes theories reusable.

Example 9. [Theory] A theory consisting of modules *move_between_points* and *carrying_things* from Example 8 would represent a general (and simple) model of commonsense motion.

The interpretation of classes of a theory for a particular domain is given in what is called a *structure*. The structure defines instances relevant to that domain, together with the values of their attributes. Not all attributes of the defined instances need to be specified, as attributes are optional.

Example 10. [Structure] The theory in Example 18 can be used to formalize a domain in which John goes from *Paris* to *London*. The interpretation of class *movers*

for this domain would consist of instance $John$; that of *points* would consist of $Paris$ and $London$; and the interpretation of class *move* would consist of instance a from Example 5.

By putting together a theory and a structure, we obtain a *system description*, which is a complete description of a particular dynamic domain. It specifies the transition diagram of that domain. In a comparison with procedural languages, a system description would correspond to a program.

Example 11. [System Description] The complete description of the particular domain described in Example 10 would consist of the theory from Example 18 and the structure from Example 10.

In the following chapters we formally define the basic concepts of $\mathcal{ALM}$. We begin with the definition of the signature of our language.

CHAPTER IV
SIGNATURE OF $\mathcal{ALM}$

Our language is parametrized by what we call a *signature of action theory*, a notion that we will define in the current chapter. As was discussed in the introduction, elaboration tolerance of $\mathcal{ALM}$ is achieved, in part, by organizing classes of objects under consideration into a hierarchy and by separating $\mathcal{ALM}$ theories from their interpretations. A signature of an $\mathcal{ALM}$ theory, referred to as *signature of action theory*, together with the corresponding notions of terms and atoms, are defined with that goal in mind. Note that in what follows we may use the word *sort* as a synonym of *class*.

By *signature of action theory* we mean a tuple of sets

$$\Sigma = \langle \mathcal{P}, \mathcal{C}, \leq_{\mathcal{C}}, \mathcal{F} \rangle$$

where

1. $\mathcal{P}$ is a collection of predefined symbols of the language. This includes standard sorts *booleans* and *integers*; sorts of the type $m..n$ where m and n are integers such that $m \leq n$; boolean constants *true* and *false*; integer constants, defined as finite strings of digits (possibly preceded by $-$); integer functions $+$, $-$, $*$, $/$, *mod*; boolean functions defined on ordered pairs of integers such as $<$, $\leq$; etc. (The definition may be extended by adding new standard symbols when needed.)

2. Elements of $\mathcal{C}$ are referred to as *user defined sorts*. In particular, $\mathcal{C}$ contains two sorts:

$$universe$$
$$actions.$$

3. Elements of $\mathcal{F}$ are referred to as *user defined function symbols*. Each function symbol $f \in \mathcal{F}$ is assigned a positive integer n called f's arity, sorts $c_0, \ldots, c_n$

for its parameters, and sort c for its values. To describe this assignment we use the meta-level notation from mathematics

$$f : c_0 \times \cdots \times c_n \rightarrow c \tag{4.1}$$

where $n \geq 0$. We assume that $\mathcal{F}$ contains a function symbol

$$occurs : actions \rightarrow booleans.$$

The other user defined function symbols are divided into four disjoint categories: *attributes*, *statics*, *inertial fluents*, and *defined fluents*, where defined fluents must range over *booleans*.

4. Relation $\leq_{\mathcal{C}}$ is a partial order on $\mathcal{C}$, used to describe the hierarchy of classes of Σ. We assume that, for every class c in $\mathcal{C}$,

$$c \leq_{\mathcal{C}} universe.$$

We say that a class pc is a *parent* of a class c in a signature $\Sigma = \langle \mathcal{P}, \mathcal{C}, \leq_{\mathcal{C}}, \mathcal{F} \rangle$ if there is no class c' different from both c and pc such that $c \leq_{\mathcal{C}} c'$ and $c' \leq_{\mathcal{C}} pc$. If pc is a parent of c in Σ, then we will conversely say that c is a *child* of pc in Σ.

Note that we distinguish between object constants, which in our signature are predefined and belong to the set $\mathcal{P}$, and functions with arity 0, which belong to the set $\mathcal{F}$.[1]

Now we are ready to define terms of action signature Σ.

Terms of a signature Σ of action theory are defined as follows:

1. A variable (i.e., an identifier which starts with a capital letter) is a term.

2. An arithmetic term (i.e., a term constructed in the usual way from integers, constants, and arithmetic operations of $\mathcal{P}$) is a term of type *integers*.

[1] User defined object constants denoting elements of user defined classes will be described in the definition of a structure. This will allow the use of an action theory in conjunction with many possible interpretations of particular classes.

3. If $f : c_0 \times \cdots \times c_n \rightarrow c$ is a function symbol and $t_0, \ldots, t_n$ are terms then $f(t_0, \ldots, t_n)$ is a term of sort c if for every $0 \leq i \leq n$

   - c_i is the sort *integers* and t_i is an arithmetic term or
   - c_i is the sort *booleans* and t_i is a boolean constant or
   - t_i is a variable.

4. If t is of sort c_1 and $c_1 \leq_{\mathcal{C}} c_2$ then t is also of sort c_2.

If t is of sort c we will write $t \in c$.

Note that according to this definition the only (non-arithmetic) terms of a signature with one sort c and two function symbols f_1 and f_2 from c to c are X, $f_1(X)$, and $f_2(X)$. An expression $f_1(f_2(X))$ is not a term. This "flattening" of the notion of term proved to be convenient for establishing relationship between theories of $\mathcal{ALM}$ and logic programs. As well, note that variables do not have a fixed sort. More details on the range of variables will be given in Chapter V.

Terms t_1 and t_2 are called *compatible* if at least one of them is a variable or if there is a sort c such that $t_1 \in c$ and $t_2 \in c$.

An *atom* of Σ is

- an expression of the form

$$t_1 = t_2 \tag{4.2}$$

  where t_1 is a term and t_2 is a constant compatible with t_1 or a variable, or

- an expression of the form

$$instance(t, c) \tag{4.3}$$

  where c is a sort and t is a term of sort c or a variable.

We refer to atoms of type (4.2) as *equality atoms* and to atoms of type (4.3) as *instance atoms*. An *action atom* is an equality atom of the form $occurs(\gamma) = true$ or $occurs(\gamma) = false$, where γ is some variable. All other equality atoms are called *function atoms*.

A *literal* is an atom a or its negation, $\neg a$.

A literal $\neg(t_1 = t_2)$ will often be written as $t_1 \neq t_2$. We will also use other usual shorthands and write $t = true$ as t and $t = false$ as $\neg t$. Infix notation will be used for arithmetic functions. Hence, $\leq (3, 5) = true$ will be written as $3 \leq 5$.

Additionally, we consider *aggregate atoms*, which are expressions of the form

$$\#aggregate\_name\{\gamma_0, \ldots, \gamma_n \ : \ l_1, \ldots, l_m\}$$

where γ_0, ..., γ_n are variables; l_1, ..., l_m are literals; and $aggregate_name$ may be, for instance, *count*, *sum*, *max*, or *min*. The following is an example of an aggregate atom:

$\#count\{X : f(X, Y)\}$.

Aggregate atoms have the same semantics as in DLV [Dell'Armi et al., 2003]. (Variables γ_0, ..., γ_n are called *bound*. In grounding, only the variables that appear in the literals l_1, ..., l_m and are not bound are grounded.)

This concludes the definition of the signature of our language. Next, we present the syntax of $\mathcal{ALM}$.

CHAPTER V
SYNTAX OF $\mathcal{ALM}$

The goal of this section is to familiarize the reader with the syntax and rationales behind the basic concepts of $\mathcal{ALM}$. A number of examples will be used to illustrate the basic ideas. We will start by defining *axioms* of our language with respect to a given signature of action theory. Later, we demonstrate how a signature of action theory can be syntactically defined by the following constructs of $\mathcal{ALM}$: *class declaration*, *function declaration*, *module*, *hierarchy of modules*, *theory*, *structure*, and *system description*.

In our syntax description, we will use boldface symbols to denote keywords of our language, identifiers starting with a lower-case letter to denote constant symbols, and identifiers starting with a capital letter to denote variables. We use square brackets followed by the $^+$ symbol to denote one-to-many repetitions and square brackets alone to denote zero or one repetitions.

5.1 Axioms

An *axiom* of $\mathcal{ALM}$ over action signature Σ is an expression of the form

$$l \textbf{ if } p \tag{5.1}$$

where l is a literal and p is a collection of literals or aggregate atoms. We call l the head of the axiom and p its body. The keyword **if** will be omitted if p is empty.

For future use in the definitions of the remaining syntactic constructs of our language, we divide axioms into several types. We will need the following terminology: A function atom of the type $f(t_0, \ldots, t_n) = t_{n+1}$ is called an *attribute atom* is f is an attribute function symbol. Similarly for *static, inertial,* and *defined atoms*. By *fluent atoms* we mean atoms that are either inertial or defined. The classification extends to literal in the usual way. We will use the symbols γ, δ, and χ (possibly indexed) as meta-variables ranging over all possible variables.

Definition 7. [Types of Axioms] There are five disjoint types of axioms in $\mathcal{ALM}$, characterized by the following syntactic restrictions on the expression in (5.1):

1. An *attribute constraint* is an axiom whose head is a negative instance or attribute literal and whose body only contains attribute or instance literals.

2. A *dynamic causal law* is an axiom whose head is an inertial fluent literal and whose body contains exactly one pair of the form $\{instance(\gamma, c), occurs(\gamma)\}$ and no other action literals.

3. A *state constraint* is an axiom whose head is a function literal and whose body does not contain action literals.

4. An *executability condition* is an axiom whose head is a negative action literal of the form $\neg occurs(\gamma)$ and whose body contains an instance atom of the form $instance(\gamma, c)$. The body may optionally contain pairs of literals of the form $\{instance(\delta, c'), occurs(\delta)\}$.

5. A *trigger* is an axiom whose head is an action atom of the form $occurs(\gamma)$ and whose body contains an instance atom of the form $instance(\gamma, c)$ and no action literals.

The shorthand

$$l_1 \equiv l_2 \textbf{ if } p$$

can be used in $\mathcal{ALM}$ for the set of axioms:

$$\begin{array}{lll} l_1 & \textbf{if} & l_2, p \\ l_2 & \textbf{if} & l_1, p \\ \neg l_1 & \textbf{if} & \neg l_2, p \\ \neg l_2 & \textbf{if} & \neg l_1, p \end{array}$$

In some practical examples, we will use in the body of axioms shorthands of the type:

$$f(\gamma_1, \ldots, \gamma_m) = g(\delta_1, \ldots, \delta_n)$$

for the set of atoms

$$\{\ f(\gamma_1, \ldots, \gamma_m) = \chi,\ g(\delta_1, \ldots, \delta_n) = \chi\ \}$$

and of the type:

$$f(g(\delta_1, \ldots, \delta_n), \ldots, \gamma_m) = \chi$$

for the set of atoms

$$\{\ g(\delta_1, \ldots, \delta_n) = \gamma_1,\ \ldots,\ f(\gamma_1, \ldots, \gamma_m) = \chi\ \}.$$

In Chapter IV we mentioned that variables appearing in a term do not have a fixed range. The range of a variable is defined in the scope of an axiom as the intersection of its required sorts, according to the terms in which it appears in that axiom.

5.2 Class Declaration

In this section, we describe how to declare classes and attributes of a signature Σ in $\mathcal{ALM}$.

When modeling a domain with a signature $\Sigma = \langle \mathcal{P}, \mathcal{C}, \leq_{\mathcal{C}}, \mathcal{F} \rangle$, we declare the user defined sorts in $\mathcal{C}$, their attributes from $\mathcal{F}$, and any attribute constraints that are relevant to this domain using the syntactic construct called *class declaration*. The syntax of a class declaration is as follows:

c : $pc_1, \ldots, pc_k$
attributes
[$attr\_name : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$]$^+$
constraints
[$attribute\_constraint.$]$^+$

where c is a sort name of $\mathcal{C}$; c_0, ..., c_{n+1}, pc_1, ..., pc_k are sort names of $\mathcal{C}$ or $\mathcal{P}$; pc_1, ..., pc_k are the parents of c in Σ; and $attr\_name$ is a function symbol from the subset of attributes of $\mathcal{F}$ with the sort assignment $attr\_name : c \times c_0 \times \cdots \times c_n \rightarrow c_{n+1}$. Note that for readability purposes we omit the parameter c in the declaration of

the attribute. Although omitted in practice, the declared class, c, is always the first (implicit) parameter and the *key* of each of its attributes.[1] The other classes, c_0, ..., c_n, when present, allow us to describe multiple properties for the same key in a compact way. If $n = 0$, the attribute declaration will be written as "$attr\_name : c_1$".

In the first line of the class declaration above, the use of the *specialization* construct, which is denoted syntactically by the symbol ":", says that the class c is a special case of the parent classes pc_1, ..., pc_k. If $k = 1$, then we have an example of single specialization. If $k > 1$ then we have an example of multiple specialization, which means that c has multiple parents. If c is a special class of *universe* only, then the expression ": *universe*" can be omitted from the first line of the class declaration of c. The attribute section can be omitted if there are no attributes to declare. Similarly for the constraint section.

Example 12. [Class Declaration] We consider a dynamic domain in which we have *things* and discrete *points* in space. Certain things, called *movers*, are able to move from one point to another. We call this domain "move between points".

Informally, the signature Σ of this domain contains the class names *points*, *things*, *movers*, and *move*, in addition to the collection of pre-declared class names, where $points \leq_{\mathcal{C}} universe$, $things \leq_{\mathcal{C}} universe$, $movers \leq_{\mathcal{C}} things$, and $move \leq_{\mathcal{C}} actions$. The attributes in Σ are $actor : move \rightarrow movers$, $origin : move \rightarrow points$, and $dest : move \rightarrow points$.

Formally, the classes in Σ and their attributes will be declared in the syntax of $\mathcal{ALM}$ as follows:

$points$

$things$

$movers : things$

[1] The function symbol for an attribute is considered to be formed by the attribute name together with its key. This means that different classes can have the same attribute name, although with different meanings, as you will see in Example 18.

move : **actions**

attributes

actor : *movers*

origin : *points*

dest : *points*

The pre-declared classes in the signature are not declared here, as their declarations are assumed to be implicit. As well, notice that the first parameter of each attribute, which is the class *move*, is omitted from the attribute declaration, and that ": *universe*" is omitted from the declarations of classes *points* and *things* for readability.

Let us now see an example of a class declaration containing attribute constraints.

Example 13. [Attribute Constraint] We consider a refinement of the "move between points" domain from Example 12 in which we only consider non-circular movements, meaning movements that take the actor to some other place than the origin. The signature Σ will remain the same, however, we would additionally have an attribute constraint saying that the origin and destination of an instance of class *move* must be different from each other. Here is how we would declare the class *move* with this additional constraint:

move : **actions**

attributes

actor : *movers*

origin : *points*

dest : *points*

constraints

$\neg instance(X, move)$ **if** $origin(X) = P$,
$dest(X) = P$.

The attribute constraint says that an instance of class *move* cannot have the same value assigned to its *origin* and *dest* attributes.

Attributes of a class are functions defined on all its instances, which includes instances of special case classes of the original class. Hence, attributes of a class are inherited by the classes that specialize it. Only the new attributes, particular to a class and not inherited from its parents, need to be mentioned in the declaration of a class.

Example 14. [Specialization] Let us consider a specialization of the "move between points" domain from Example 12 in which certain things can be carried from one point to another. We call such things *carriables*. The user defined sorts associated with this domain would include two new class names, *carriables* and *carry*, such that $carriables \leq_{\mathcal{C}} things$ and $carry \leq_{\mathcal{C}} move$. The user defined functions would include a new attribute, $carried\_thing : carry \rightarrow carriables$. To describe this new domain, called "carrying things", we would need to expand the previous collection of class declarations by:

carriables : *things*

carry : *move*

attributes

$carried\_thing : carriables$

This declaration says that *carry* inherits all attributes from *move* but has one extra attribute: *carried_thing*.

5.3 Function Declaration

In this section we will show how functions of an action signature are syntactically described in $\mathcal{ALM}$.

Functions from $\mathcal{F}$ that are not attributes (i.e., statics, inertial fluents, and defined fluents) are declared using the syntactic construct of *function declaration*. The syntax of a function declaration is as follows:

$$type\ f : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$$

where *type* is one of the keywords **static**, **inertial**, or **defined**; $c_0, \ldots, c_n, c_{n+1}$ are

sort names from $\mathcal{C}$ or $\mathcal{P}$; and f is a non-attribute function symbol from $\mathcal{F}$, with the sort assignment $f : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$. The presence of the keyword *static*, *inertial*, or $defined$ in the declaration of a function f says that f denotes a static, an inertial fluent, or a defined fluent, respectively.

Example 15. [Function Declaration] In our "move between points" domain from Example 12, we are interested in the locations of *things*. Therefore, the set of function names $\mathcal{F}$ of our signature Σ will contain, in addition to the already mentioned attributes, an inertial fluent *loc_in* with the sort assignment $loc\_in : things \rightarrow points$. The description of this domain will contain the following function declaration:

inertial $loc\_in : things \rightarrow points$

5.4 Module

At this point, we have all the information needed to define a *module* of $\mathcal{ALM}$. Intuitively, a module represents a piece of knowledge which can be developed and tested independently from other pieces of knowledge, and can be combined with other modules in various ways. Syntactically, a module is a collection of declarations of classes with their attributes, declarations of non-attribute functions, and axioms describing inter-relations between functions.

The syntax of a *module declaration* is as follows:

module *module_name*
class declarations
$[class\ declaration]^+$
function declarations
$[function\ declaration]^+$
axioms
dynamic causal laws
$[dynamic\_causal\_law.]^+$

state constraints

$[state\_constraint.]^+$

executability conditions

$[executability\_condition.]^+$

triggers

$[trigger.]^+$

All user defined class and function names from the signature of a module must be declared in that module. Note that all the object constants from the signature of a module are predefined, i.e., belong to $\mathcal{P}$. Hence, the axioms of a module may only contain *predefined* object constants. We call such axioms *open*. This restriction ensures that the information in a module is general enough so that it can be used to model different domains.

Any of the sections of a module declaration may be omitted if the corresponding set of declarations or axioms is empty.

Example 16. [Module] Let us reconsider the "move between points" domain from Example 15 and represent it in a single module called *move_between_points*. This module will contain axioms about the direct effects of action *move* and its executability conditions.

module *move_between_points*

class declarations

points

things

movers : *things*

move : **actions**

attributes

actor : *movers*

origin : *points*

dest : *points*

function declarations

inertial $loc\_in : things \rightarrow points$

axioms

dynamic causal laws

$$loc\_in(A) = D \quad \textbf{if} \quad instance(X, move),\ occurs(X),\ actor(X) = A,\ dest(X) = D.$$

executability conditions

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = A,\ origin(X) = O,\ loc\_in(A) \neq O.$$

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = A,\ dest(X) = D,\ loc\_in(A) = D.$$

The dynamic causal law in this module says that the direct effect of the occurrence of an instance of class *move* is that the actor of this instance will be located at the destination. Similarly, the two executability conditions say that an instance of class *move* cannot be executed when the actor is not located at the origin or when the actor is already located in the destination.

5.5 Hierarchy of Modules

Usually a user's knowledge about a particular class of domains is captured by a collection of modules of $\mathcal{ALM}$. Ideally, such modules should be largely independent from each other in order to facilitate the stepwise development, testing, and readability of the knowledge base. We say that two modules are independent if their signatures, excluding any predefined symbols, are disjoint. However, in practice, mod-

ules of a knowledge base are often not independent from each other, especially if they are concerned with a common theme. In particular, there are cases in which a module is developed based on previously created modules, by extending their signatures or the sets of axioms they contain. In such cases, we say that the new module depends on the previous modules. We call this relation the *module dependence relation*. We define it on a collection of modules $\mathcal{M}$, and denote it by $\leq_{\mathcal{M}}$ as in $m_2 \leq_{\mathcal{M}} m_1$, which means that m_2 depends on m_1.

A collection of modules $\mathcal{M}$ together with the dependence relation $\leq_{\mathcal{M}}$ defined on $\mathcal{M}$ is a *hierarchy of modules* if $\leq_{\mathcal{M}}$ is a partial order on $\mathcal{M}$.

If m is a module depending on several other modules in a hierarchy, then the module description of m in the syntax of $\mathcal{ALM}$ may be of an unmanageable size. In order to remedy this problem, we syntactically encode the dependence relation between two modules using the *module dependence* construct, which is expressed by a statement of the form

depends on *module_name*

that is added after the first line of a module declaration, following the name of the dependent larger module, as in:

module m_2
depends on m_1
...

The addition of such a statement to the declaration of module m_2 says that m_2 depends on m_1 (i.e., $m_2 \leq_{\mathcal{M}} m_1$), and that m_2 inherits the signature and axioms of m_1. All sort and function names from the signature of m_1 are implicitly part of the signature of m_2 as well, but their explicit declarations *must not* appear in m_2; similarly for axioms. This ensures that the information in m_1, although reused, is not explicitly repeated in m_2 and thus the size of m_2 remains manageable. On the other hand, the developer of the knowledge base should make sure that the dependence relations between modules are not overly complex in order to maintain the readability

of modules and facilitate their development.

Example 17. [Module Dependence] Imagine that our knowledge base already contained the module *move_between_points* from Example 16 and we now wanted to write a module describing the "carrying things" domain from Example 14. The new module, which we will call *carring_things*, should be separate from *move_between_points* so that the smaller initial module can be used alone in the representation of domains that do not involve carrying actions. However, *carrying_things* will depend on *move_between_points* because action *carry* is described as a special case of action *move* (as seen in Example 14) and fluent *loc_in* is relevant in describing the effects of *carry* actions too. We use the module dependence construct to express this relation between modules, as you will see below.

Classes and functions from *move_between_points* (e.g., *things*, *move*, *actor*, *loc_in*, etc.) are considered implicitly declared in *carrying_things* and can be used in its declarations and axioms. Additionally, the new module will contain two new sorts, *carriables* and *carry* with attribute *carried_thing*, the inertial fluent *holding*, the defined fluent *is_held* and three new axioms.

The two modules form a hierarchy $\mathcal{M}$ in which *carrying_things* $\leq_{\mathcal{M}}$ *move_between_points*.

module *carrying_things*
depends on *move_between_points*
class declarations
carriables : *things*
carry : *move*
attributes
carried_thing : *carriables*
function declarations
inertial *holding* : *things* × *things* → *booleans*
defined *is_held* : *things* → *booleans*

axioms

state constraints

$loc\_in(C) = P \equiv loc\_in(T) = P$ **if** $holding(T, C)$.

$is\_held(X)$ **if** $holding(T, X)$.

executability conditions

$\neg occurs(X)$ **if** $instance(X, move)$,
$actor(X) = A$,
$is\_held(A)$.

$\neg occurs(X)$ **if** $instance(X, carry)$,
$actor(X) = A$,
$carried\_thing(X) = C$,
$\neg holding(A, C)$.

The first state constraint in this module says that, whenever one thing holds another thing in place, their locations must be identical.

The module dependence construct can also be used to express that one module depends on multiple other modules, as in:

module m_{k+1}
depends on $m_1, \ldots, m_k$
...

A module depending on other modules will be called *open*; modules that do not depend on other modules will be called *closed*. The module *move_between_points* from Example 16 is closed, while *carrying_things* from Example 17 is open. Modules containing sufficiently general knowledge can be stored in a library; we call such modules *library modules*.

5.6 Theory

We can now define one of the main concepts of $\mathcal{ALM}$, that of a *theory*. Roughly speaking, a theory is a hierarchy of modules that satisfies certain semantic conditions.

In order to describe these conditions formally, we need the following preliminary definition.

If $\mathcal{T}$ is a hierarchy of modules $\{m_1, \ldots, m_n\}$ and $\Sigma_i = \langle \mathcal{P}, \mathcal{C}_i, \leq_{\mathcal{C}_i}, \mathcal{F}_i \rangle$ is the signature of module m_i from $\mathcal{T}$, with $1 \leq i \leq n$, let

$$\mathcal{C} = \bigcup_{1 \leq i \leq n} \mathcal{C}_i$$

$$\mathcal{F} = \bigcup_{1 \leq i \leq n} \mathcal{F}_i$$

and let $\leq_{\mathcal{C}}$ be the binary relation defined on $\mathcal{C} \times \mathcal{C}$ as follows: if $x, y \in \mathcal{C}$ and there is a module $m_i \in \mathcal{T}$ such that $x \leq_{\mathcal{C}_i} y$, then $x \leq_{\mathcal{C}} y$.

Definition 8. [Theory] A hierarchy of modules $\mathcal{T} = \{m_1, \ldots, m_n\}$ is a *theory* with signature $\Sigma = \langle \mathcal{P}, \mathcal{C}, \leq_{\mathcal{C}}, \mathcal{F} \rangle$ if:

- The relation $\leq_{\mathcal{C}}$ is a partial order on $\mathcal{C}$.
- Every sort name from $\mathcal{C}$ and every function name from $\mathcal{F}$ is declared *explicitly* exactly once in the collection of module declarations for $m_1, \ldots, m_n$.

Syntactically, a theory is described by an expression of the form:

theory *theory_name*
[*module declaration*]$^+$

A theory can import modules from a library using the *import module* construct, which has the syntax:

import *module_name* **from** *library_name*

Such statements are placed after the first line of a theory's declaration. By importing a module in a theory, that module and all its ancestors get copied within the theory. Notice that only ancestors, and not descendants, of an imported module are copied into the theory. It is necessary to import all ancestors of an imported module in order to ensure that the last condition of the definition of a theory is met, as ancestors contain the explicit declarations of some of the symbols used in the imported module.

Example 18. [Theory] The two modules declared so far, *move_between_points* from Example 16 and *carrying_things* from Example 17, express very general knowledge. Let us assume that they are stored in a library called *basic_motion*. We can now create a theory called *travel* that contains these two library modules. Our import statements only need to mention the module *carrying_things* and its ancestor, *move_between_points*, will be automatically imported as well.

theory *travel*
  import *carrying_things* **from** *basic_motion*

As a result, the theory *travel* is equivalent to a theory *travel_1* below, which contains the two modules explicitly declared:

theory *travel_1*
  module *move_between_points*
    ...
  module *carrying_things*
    depends on *move_between_points*
    ...

The theory *travel* can optionally be expanded by other modules. For instance, we consider a theory *travel_2* that is identical to *travel*, with the exception of a new explicitly declared module called *grasp_and_release* about grasping and releasing things.

theory *travel_2*
  import *carrying_things* **from** *basic_motion*
  module *grasp_and_release*
    depends on *carrying_things*
    class declarations
      grasp : **actions**
        attributes
          actor : *movers*
          patient : *carriables*

$release$: **actions**

attributes

$actor : movers$

$patient : carriables$

axioms

dynamic causal laws

$holding(A, P)$ **if** $instance(X, grasp)$,
$occurs(X)$,
$actor(X) = A$,
$patient(X) = P$.

$\neg holding(A, P)$ **if** $occurs(X)$,
$instance(X, release)$,
$actor(X) = A$,
$patient(X) = P$.

executability conditions

$\neg occurs(X)$ **if** $instance(X, grasp)$,
$actor(X) = A$,
$patient(X) = P$,
$holding(A, P)$.

$\neg occurs(X)$ **if** $instance(X, release)$,
$actor(X) = A$,
$patient(X) = P$,
$\neg holding(A, P)$.

Syntactically, a theory is equivalent to a single closed module whose class declarations consist of the union of class declarations from all of its modules, and similarly for function declarations, dynamic causal laws, state constraints, executability conditions, and triggers. Theories containing sufficiently general knowledge can be stored in knowledge libraries.

5.7 Structure

Symbols in a module and, therefore, in a theory do not have a fixed interpretation, with the exception of the *pre-interpreted* symbols in the set $\mathcal{P}$ of a signature of action theory. The interpretation of other classes and of their attributes, for a particular domain, is given externally to the theory in what we call a *structure*. This is the second main concept of $\mathcal{ALM}$.

Note that a structure does not interpret statics, inertial or defined fluents, nor the function *occurs* in the signature of a theory. The interpretation of fluents and of the function *occurs* depends on the step in a trajectory and therefore is given separately from the system description. We view statics as a special case of fluents. We will discuss the interpretation of these remaining function symbols and of the axioms in which they appear when we give the semantics of our language in Chapter VI.

Let $\mathcal{T}$ be a theory with signature $\Sigma = \langle \mathcal{P}, \mathcal{C}, \leq_{\mathcal{C}}, \mathcal{F} \rangle$.

Definition 9. [Structure] By a *structure* of a theory $\mathcal{T}$ we mean a pair $\langle \mathcal{U}, \mathcal{I} \rangle$ where $\mathcal{U}$ is a set called the *universe* of theory $\mathcal{T}$ [2] and $\mathcal{I}$ is a mapping assigning:

- to each class name c of $\mathcal{T}$ a subset $\mathcal{I}(c)$ of $\mathcal{U}$ called the *base* of c. We call elements of $\mathcal{I}(c)$ *instances* of c.
- to each attribute name $a : c \times c_0 \times \cdots \times c_n \rightarrow c_{n+1}$ a function $\mathcal{I}(a) : \mathcal{I}(c) \times \mathcal{I}(c_0) \times \cdots \times \mathcal{I}(c_n) \rightarrow \mathcal{I}(c_{n+1})$. The function $\mathcal{I}(a)$ is not required to be total, *it may be partial.*

The mapping $\mathcal{I}$ is pre-defined on pre-interpreted symbols from $\mathcal{P}$ accordingly to their respective standard interpretations.

Additionally, the mapping $\mathcal{I}$ should satisfy the following conditions:

1. For every class $c \in \mathcal{C}$

$$\mathcal{I}(c) \supseteq \bigcup_{c_i \ \leq_{\mathcal{C}} \ c} \mathcal{I}(c_i)$$

[2] The term *universe of a theory* should not be confused with the pre-declared class name *universe*.

2. $\mathcal{I}$ satisfies the attribute constraints of $\mathcal{T}$.

   We say that $\mathcal{I}$ *satisfies* an attribute constraint ac if it satisfies all the logic programming rules obtained from ac by replacing its variables by elements from the bases of the appropriate sorts in $\mathcal{I}$ and substituting the keyword **if** by $\leftarrow$. We say that $\mathcal{I}$ satisfies an instance literal $instance(t, c)$ if $t \in \mathcal{I}(c)$ and an instance literal $\neg instance(t, c)$ if $t \notin \mathcal{I}(c)$.

Note that a theory may be interpreted in many different ways to reflect different domains.

Syntactically, a structure of a theory is defined using a *structure description*, which is a collection of definitions of instances together with their attribute assignments. A *structure description* has the form:

structure

$$[instance\ definition]^+$$

An *instance definition* is a statement

$$inst\_name\ \textbf{in}\ c$$
$$[\ attr\_name[(inst_1, \ldots, inst_n)]\ =\ inst_{n+1}\ ]^+$$

where $inst\_name$ is the name of an instance of class c; $attr\_name$ is the name of an attribute of c; and $inst_1$, ..., $inst_n$, $inst_{n+1}$ are instances of the appropriate classes, according to the sort assignment of attribute $attr\_name$. The attribute assignment says that the function $attr\_name$ has value $inst_{n+1}$ for input $inst\_name, inst_1, \ldots, inst_n$. Note that $inst\_name$ is always the first parameter of its attributes, although omitted in practice for readability.

In order to maintain structure descriptions concise, we only require an explicit instance definition for an element $inst$ in $\mathcal{I}(c)$ if c is a minimal element of $\mathcal{C}$ or, otherwise, if $inst$ does not belong to the interpretation of any of the children of c (i.e., $inst \in \mathcal{I}(c) \setminus \bigcup_{c_i \leq_{\mathcal{C}} c} \mathcal{I}(c_i)$). The complete mapping can be recovered from these instance definitions, based on the restriction placed on the mapping $\mathcal{I}$ that $\mathcal{I}(c) \supseteq \bigcup_{c_i \leq_{\mathcal{C}} c} \mathcal{I}(c_i)$.

Multiple instances of the same class can be defined in the same statement separated by comas, as in:

$inst_1, inst_2$ **in** c

which is a shorthand for the set of instance definitions:

$inst_1$ **in** c

$inst_2$ **in** c

Example 19. [Structure] Let us consider the theory *travel* from Example 18. Imagine a particular domain that can be described by this theory, in which there are two *movers*, John and Bob, two *points* in space, London and Paris, and one *carriable*, a suitcase. For simplicity, we only consider two of all the possible actions: $move(john, paris, london)$ (John goes from Paris to London) and $carry(john, suitcase, london)$ (John carries his suitcase to London). This particular interpretation of theory *travel* is described as follows:

structure

$john, bob$ **in** $movers$

$london, paris$ **in** $points$

$suitcase$ **in** $carriables$

$move(john, paris, london)$ **in** $move$

$actor = john$

$origin = paris$

$dest = london$

$carry(john, suitcase, london)$ **in** $carry$

$actor = john$

$carried\_thing = suitcase$

$dest = london$

The above structure description defines a structure $\langle \mathcal{U}, \mathcal{I} \rangle$ such that:

$$\mathcal{U} = \{true, false, john, bob, london, paris, suitcase, move(john, paris, london), carry(john, suitcase, london)\}$$

and

$\mathcal{I}(booleans) = \{true, false\}$

$\mathcal{I}(movers) = \{john, bob\}$

$\mathcal{I}(carriables) = \{suitcase\}$

$\mathcal{I}(things) = \{john, bob, suitcase\}$

$\mathcal{I}(carry) = \{carry(john, suitcase, london)\}$

$\mathcal{I}(move) = \{move(john, paris, london), carry(john, suitcase, london)\}$

Another way of defining multiple instances at a time is by using instance schemas. An *instance schema* is a special type of an instance definition, containing variables. Syntactically, it has the form

$$inst\_name(V_1, \ldots, V_k) \textbf{ in } c \text{ [} \textbf{where } cond \text{]}$$
$$[\ attr\_name[inst_1, \ldots, inst_n] \ = \ V_i\ ]^+$$

where $1 \leq i \leq k$; $V_1, \ldots, V_k$ are variables; and *cond* is a collection of *instance* literals, attribute literals, or literals obtained from pre-defined functions. An instance schema is viewed as a shorthand for the set of instances obtained from the schema by replacing variables by their possibles values (according to the sort assignments of attributes) and satisfying all attribute constraints and *cond*. The name of an instance schema must contain all the variables appearing in its assignments of values to attributes, and no other variables. We make this requirement because: (1) if a variable appearing in an attribute assignment is missing, then multiple different instances will share the same name, which generates ambiguity; (2) if the name of an instance schema contains an extra variable that does not appear in assignments of values to attributes, then this will result in a large number of instances that are identical to each other in terms of the axioms they satisfy and this will increase the cardinality of the universe unnecessarily.

Example 20. [Instance Schema] We may add to the structure in Example 19 the following instance schema:

$move(A, P)$ **in** $move$

$actor = A$

$dest = P$

which stands for the collection of instances of class $move$: $\{move(john, paris)$, $move(john, london)$, $move(bob, paris)$, $move(bob, london)\}$.

5.8 System Description

A particular dynamic system is described in $\mathcal{ALM}$ using the construct called *system description*. Conceptually, a system description consists of a *theory* (the system's declarations) and a *structure* of that theory (the system's structure). We denote a system description consisting of a theory $\mathcal{T}$ and a structure $\langle \mathcal{U}, \mathcal{I} \rangle$ of $\mathcal{T}$ by the pair $\langle \mathcal{T}, \langle \mathcal{U}, \mathcal{I} \rangle \rangle$. A system description defines the transition diagram of the particular dynamic system it describes.

Syntactically, a system description has the form:

system description *name*

theory description

structure description

Example 21. [System Description] A system description called *basic_travel* constructed from theory *travel* in Example 18 and the structure from Example 19 defines the transition diagram describing John's movements from Paris to London while carrying a suitcase or not.

CHAPTER VI
SEMANTICS OF $\mathcal{ALM}$

In the preceding chapters, we described the signature and syntax of our language. In the current chapter, we present the semantics of $\mathcal{ALM}$.

We will give the semantics of our language by describing the transition diagram $\tau(\mathcal{D})$ defined by a system description $\mathcal{D}$. This will require defining the states and transitions of $\tau(\mathcal{D})$. In order to do that, we must first complete the interpretation of the symbols in the signature of $\mathcal{D}$. So far, we have shown that class and attribute names, together with pre-interpreted symbols of our language, are interpreted by the structure of $\mathcal{D}$, $\langle \mathcal{U}, \mathcal{I} \rangle$ (see Chapter V). We now expand the mapping $\mathcal{I}$ on the remaining symbols of the signature, which are the names of user defined statics, inertial fluents, and defined fluents, and the special function *occurs*. $\mathcal{I}$ maps every such function $f : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$ into a total function $\mathcal{I}(f) : \mathcal{I}(c_0) \times \cdots \times \mathcal{I}(c_n) \rightarrow \mathcal{I}(c_{n+1})$. Note that, as was typical years ago, we consider that statics and fluents are total functions. Nowadays there is substantial research in allowing partial functions in action languages and ASP; we plan to address this issue in the context of $\mathcal{ALM}$ in the future.

We call $\langle \mathcal{U}, \mathcal{I} \rangle$, where $\mathcal{I}$ is the extended mapping mentioned above, a *complete interpretation* of the symbols in the theory of $\mathcal{D}$. For later use, we denote by $\mathcal{F}^*$ the collection of user defined statics, defined fluents and inertial fluents from the signature of $\mathcal{D}$. We will now define the transition diagram $\tau(\mathcal{D})$ with respect to a fixed interpretation of classes and attributes, the one given by the structure of $\mathcal{D}$.

6.1 States of $\tau(\mathcal{D})$

Intuitively, a *state* of the transition diagram $\tau(\mathcal{D})$ is the restriction $\mathcal{I}|_{\mathcal{F}^*}$ of $\mathcal{I}$ to $\mathcal{F}^*$, where $\langle \mathcal{U}, \mathcal{I} \rangle$ is a complete interpretation that satisfies the state constraints of the theory $\mathcal{T}$ of $\mathcal{D}$.

The precise expression of this intuition is not entirely trivial because of defined

fluents whose specification is given in terms of the closed world assumption. To deal with this problem we define a state by constructing a logic program $\Pi(\mathcal{D})$ that captures our intuitive requirements and by identifying states of $\tau(\mathcal{D})$ with parts of the answer sets of $\Pi(\mathcal{D})$.

The signature of $\Pi(\mathcal{D})$ consists of:

- object names from $\mathcal{U}$, class names from $\mathcal{C}$, function names from $\mathcal{F}$, predefined function names from $\mathcal{P}$;
- the relation $instance(x, c)$ where x ranges over object names from $\mathcal{U}$ and c ranges over class names from $\mathcal{C}$;
- relations $function(f(c_0, \ldots, c_n), t)$, $range(f(c_0, \ldots, c_n), c_{n+1})$, $val(f(c_0, \ldots, c_n), c_{n+1})$, and $total(f(c_0, \ldots, c_n))$, for every function name f of type t in $\mathcal{F}^*$ such that $f : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$ and $t \in \{attribute, static, inertial, defined\}$;
- a new sort $function$.

Next, we define a transformation lp that maps axioms of $\mathcal{T}$ into logic programming rules. For now, we will use this transformation on state constraints only. For every axiom l **if** p from $\mathcal{T}$,

$$lp(l \textbf{ if } p)$$

is obtained as follows:

(a) replace the keyword " **if** " in l **if** p by the connective "$\leftarrow$";

(b) for every term $f(t_0, \ldots, t_n)$ appearing in the rule resulting from step (a), where $f : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$, and for every t_i, $0 \leq i \leq n$, in this term, add to the body of the rule the atom $instance(t_i, c_i)$.

(c) for every function atom $f(t_0, \ldots, t_n) = t_{n+1}$ occurring in the rule resulting from step (b), where $f : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$, replace $f(t_0, \ldots, t_n) = t_{n+1}$ by $val(f(t_0, \ldots, t_n), t_{n+1})$ and add the atom $instance(t_{n+1}, c_{n+1})$ to the body of the rule.

As mentioned above, we assume that the interpretation of classes and attributes from the signature of $\mathcal{D}$ is fixed by the structure of $\mathcal{D}$. Hence, for every complete interpretation $\langle \mathcal{U}, \mathcal{I} \rangle$ of the symbols in the signature of $\mathcal{D}$, $\mathcal{I}(c)$ is fixed if $c \in \mathcal{C}$ or $c \in \mathcal{P}$, and so is $\mathcal{I}(a)$ if a is an attribute from $\mathcal{F}$.

Definition of $\Pi(\mathcal{D})$

We define $\Pi(\mathcal{D})$ as follows:

1. For every element $x \in \mathcal{U}$ and every class $c \in \mathcal{C}$:

   If $x \in \mathcal{I}(c)$, then $\Pi(\mathcal{D})$ contains the fact

   $$instance(x, c).$$

   Otherwise, $\Pi(\mathcal{D})$ contains the fact

   $$\neg instance(x, c).$$

2. For every attribute $a : c \times c_0 \times \cdots \times c_n \rightarrow c_{n+1}$ in $\mathcal{F}$ of a class c, and for every $x \in \mathcal{I}(c)$, $x_0 \in \mathcal{I}(c_0)$, $\ldots$, $x_n \in \mathcal{I}(c_n)$:

   If $\mathcal{I}(a)(x, x_0, \ldots, x_n) = x_{n+1}$, then $\Pi(\mathcal{D})$ contains the fact

   $$val(a(x, x_0, \ldots, x_n), x_{n+1}).$$

3. For every function $f : c_0 \times \cdots \times c_n \rightarrow c_{n+1}$ in $\mathcal{F}$ of type t, $\Pi(\mathcal{D})$ contains:

$$\begin{aligned} function(f(X_0, \ldots, X_n), t) \quad &\leftarrow \quad instance(X_0, c_0), \\ &\qquad \ldots, \\ &\qquad instance(X_n, c_n). \\ range(f(X_0, \ldots, X_n), X_{n+1}) \quad &\leftarrow \quad instance(X_0, c_0), \\ &\qquad \ldots, \\ &\qquad instance(X_n, c_n), \\ &\qquad instance(X_{n+1}, c_{n+1}). \end{aligned}$$

4. $\Pi(\mathcal{D})$ contains a rule defining the sort $function$:

$$function(F) \leftarrow function(F, Type).$$

5. $\Pi(\mathcal{D})$ contains a rule expressing the "uniqueness of value" condition for functions:

$$\begin{aligned} \neg val(F, Y) \quad \leftarrow \quad & val(F, Z), \\ & Y \neq Z, \\ & function(F), \\ & range(F, Y), \\ & range(F, Z). \end{aligned}$$

6. $\Pi(\mathcal{D})$ contains axioms specifying the "existence of a value" condition for functions:

   First, an axiom describing what it means for the value of a function to be defined:

$$\begin{aligned} total(F) \quad \leftarrow \quad & val(F, V), \\ & function(F), \\ & range(F, V). \end{aligned}$$

   Second, a constraint stating that every non-attribute function must be defined:

$$\begin{aligned} \leftarrow \quad & not\ total(F), \\ & function(F), \\ & not\ attribute(F). \end{aligned}$$

7. $\Pi(\mathcal{D})$ contains the Closed World Assumption for defined fluents:

$$\begin{aligned} val(F, false) \quad \leftarrow \quad & not\ val(F, true), \\ & function(F, defined). \end{aligned}$$

8. For every state constraint " l **if** p " in $\mathcal{T}$, $\Pi(\mathcal{D})$ contains:

$$lp(l \textbf{ if } p).$$

We will now show how we use the program $\Pi(\mathcal{D})$ to define the states of $\tau(\mathcal{D})$. Let $\langle \mathcal{U}, \mathcal{I} \rangle$ be a complete interpretation of the symbols in the signature of $\mathcal{D}$ and let σ be the restriction of $\mathcal{I}$ to the function symbols of $\mathcal{F}^*$. By σ_{nd} we denote the restriction of σ to non-defined (i.e., static or inertial) function symbols. Let

$$\Pi(\mathcal{D}, \sigma) =_{def} \Pi(\mathcal{D}) \cup \{val(f, v) : f = v \in \sigma_{nd}\}$$

Definition 10. [State] The restriction σ of $\mathcal{I}$ to $\mathcal{F}^*$, where $\langle \mathcal{U}, \mathcal{I} \rangle$ is a complete interpretation of the symbols in the signature of $\mathcal{D}$, is a *state* of the transition diagram $\tau(\mathcal{D})$ if $\Pi(\mathcal{D}, \sigma)$ has a unique answer set, A, and

$$\sigma = \{f = v : val(f, v) \in A\}.$$

Example 22. [State] The transition diagram of the system description *basic_travel* from Example 21 contains, for instance, a state defining the assignments: $holding(john,\ suitcase) = true$, $is\_held(suitcase) = true$, $loc\_in(john) = paris$, $loc\_in(suitcase) = paris$, $loc\_in(bob) = london$, and assigning the value $false$ to fluents *holding* and *is_held* for all the remaining possible inputs.

However, any σ in which $holding(john, suitcase) = true$ and $is\_held(suitcase) = false$ is not a state of the transition diagram of *basic_travel* because it does not satisfy the state constraint that gives the definition of the defined fluent *is_held*, which can be seen in Example 17:

$is\_held(X)$ **if** $holding(T, X)$.

6.2 Transitions of $\tau(\mathcal{D})$

Let us now assume that we have two complete interpretations $\langle \mathcal{U}, \mathcal{I}_0 \rangle$ and $\langle \mathcal{U}, \mathcal{I}_1 \rangle$ coinciding on the interpretation of classes and attributes, which was fixed by the structure of $\mathcal{D}$, and that $\mathcal{I}_1(occurs) = \emptyset$. By σ_0 we denote the restriction $\mathcal{I}_0|_{\mathcal{F}^*}$, by σ_1 the restriction $\mathcal{I}_1|_{\mathcal{F}^*}$, and by a the collection of actions $\{e : occurs(e) = true \in \mathcal{I}_0(occurs)\}$. Intuitively, $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition of $\tau(\mathcal{D})$ if σ_0 and σ_1 are states and

$\mathcal{I}_0$ and $\mathcal{I}_1$ satisfy the dynamic causal laws, executability conditions, and triggers of the theory $\mathcal{T}$ of $\mathcal{D}$.

In order to precisely define the transitions of $\tau(\mathcal{D})$ we construct a logic program $\Pi_1(\mathcal{D})$ obtained from the previously described program $\Pi(\mathcal{D})$ from Section 6.1. The signature of $\Pi_1(\mathcal{D})$ extends the one of $\Pi(\mathcal{D})$ by a new sort called *step* with constants 0 and 1, a relation $occurs(action, step)$, and a relation $holds(fluent, step)$, where $fluent$ is either a relation of the type $val(f(c_0, \ldots, c_n), c_{n+1})$ or of the type $total(f(c_0, \ldots, c_n))$ such that f is a symbol for a fluent of $\mathcal{F}^*$.

We use the term *fluent atoms of a logic program* to denote expressions of the form

$$val(f(t_0, \ldots, t_n), t_{n+1})$$

and

$$total(f(t_0, \ldots, t_n))$$

where t_i, $0 \leq i \leq n+1$, are either variables or constants and f is either an inertial or a defined fluent symbol. In our definition of $\Pi_1(\mathcal{D})$ we use the variable I for steps.

Definition of $\Pi_1(\mathcal{D})$

We define $\Pi_1(\mathcal{D})$ as follows:

1. For every rule r in $\Pi(\mathcal{D})$, replace every fluent atom fl in r by

   $$holds(fl, I)$$

   and add the resulting rule to $\Pi_1(\mathcal{D})$.

2. For every dynamic causal law, executability condition, or trigger "l **if** p" in $\mathcal{T}$, $\Pi_1(\mathcal{D})$ contains a rule obtained as follows:

   (a) Apply the transformation lp to "l **if** p".

   (b) Replace every fluent atom fl in $lp(l$ **if** $p)$ by:

   $$holds(fl, I+1)$$

if fl occurs in the head of a dynamic causal law, and by:

$$holds(fl, I)$$

otherwise.

(c) Replace every action atom of the type $occurs(X)$ in the resulting rule by an atom $occurs(X, I)$. Then add the obtained rule to $\Pi_1(\mathcal{D})$.

(We assume that an action literal $occurs(X) = true$ is written as $occurs(X)$ and $occurs(X) = false$ is written as $\neg occurs(X)$ in " l **if** p ".)

3. $\Pi_1(\mathcal{D})$ contains the Inertia Axioms for inertial fluents:

$$\begin{array}{lcl} holds(val(F,V), I+1) & \leftarrow & holds(val(F,V), I), \\ & & not\ \neg holds(val(F,V), I+1), \\ & & function(F, inertial), \\ & & range(F,V). \\ \neg holds(val(F,V), I+1) & \leftarrow & \neg holds(val(F,V), I), \\ & & not\ holds(val(F,V), I+1), \\ & & function(F, inertial), \\ & & range(F,V). \end{array}$$

We can now describe how we use the program $\Pi_1(\mathcal{D})$ to define the transitions of $\tau(\mathcal{D})$. But first, we define the logic programming encodings of states and of a collection of actions. For every state σ we denote by σ_s the restriction of σ to static functions. For every state σ and step i, if $f = v \in \sigma_s$, then

$$h(val(f,v),\ i)\ =_{def}\ val(f,v)$$
$$h(total(f),\ i)\ =_{def}\ total(f)$$

and otherwise

$$h(val(f,v),\ i)\ =_{def}\ holds(val(f,v), i)$$
$$h(total(f),\ i)\ =_{def}\ holds(total(f), i)$$

As well,

$$h(\sigma, i)\ =_{def}\ \{h(l,i) : l \in \sigma\}$$

If i is a step and a is a collection of actions, then

$$occurs(a, i) =_{def} \{occurs(e, i) : e \in a\}$$

If σ_0 is a state and a is a collection of actions, then

$$\Pi_1(\mathcal{D}, \sigma_0, a) =_{def} \Pi_!(\mathcal{D}) \cup h(\sigma_0, 0) \cup occurs(a, 0)$$

Definition 11. [Transition] A *transition* $\langle \sigma_0,\ a,\ \sigma_1 \rangle$ is in $\tau(\mathcal{D})$ if $\Pi_1(\mathcal{D}, \sigma_0, a)$ has an answer set A such that

$$a = \{e : occurs(e, 0) \in A\}$$
$$\sigma_1 = \{f = v : h(val(f, v), 1) \in A\}.$$

Example 23. [Transition] The transition diagram of the system description *basic_travel* from Example 21 contains, for instance, a transition $\langle \sigma_0, a, \sigma_1 \rangle$ where:

- σ_0 is the state in which $loc\_in(john) = paris$, $loc\_in(bob) = london$, $loc\_in(suitcase) = london$, and fluents *holding* and *is_held* have the value $false$ for all their possible inputs.

- $a = \{move(john, paris, london)\}$

- σ_1 is the state in which $loc\_in(john) = london$, $loc\_in(bob) = london$, $loc\_in(suitcase) = london$, and fluents *holding* and *is_held* have the value $false$ for all their possible inputs.

On the other hand, the triple $\langle \sigma_0, a, \sigma_1' \rangle$ where $loc\_in(john) = paris \in \sigma_1'$ is not a transition, because it does not satisfy the dynamic causal law for actions of class *move*, which appears in Example 16.

A system description $\mathcal{D}$ is called *consistent* if the transition diagram it defines contains at least one non-empty state. This completes the description of $\mathcal{ALM}$. In the next chapter we present methodology for representing knowledge in $\mathcal{ALM}$.

CHAPTER VII
METHODOLOGY OF REPRESENTING KNOWLEDGE IN $\mathcal{ALM}$

The next step after designing our language was to develop a methodology of its use. For that purpose, we needed to experiment with representing knowledge about various domains. For our first experiments, which followed the design of the first version of $\mathcal{ALM}$, we chose two benchmark examples from the field of reasoning about actions: Blocks World and Monkey and Banana.[1] After the design of the current version of $\mathcal{ALM}$, we extended our practice to other known domains: River Crossing (a domain inspired by the Missionaries and Cannibals puzzle), Pednault's Briefcase, and Towers of Hanoi.

We wanted our methodology for knowledge representation to be characterized by the reuse of information, elaboration tolerance, simplicity, and computability. Hence, in our practice we started with the knowledge already represented in $\mathcal{ALM}$— the motion modules from Chapter V. These modules were created with the purpose of illustrating the syntax of our language and were not initially intended to be library modules. We transformed them so that the knowledge they describe is general and can be used to formalize various different domains. We considered the resulting modules to be part of a commonsense library about motion. We reused the library in formalizing the above mentioned sample domains. We were able to do so in a straightforward manner for the River Crossing problem and for Pednault's Briefcase domain. However, the ontology of the initial library was not rich enough to cover domains like Blocks World, Towers of Hanoi, and Monkey and Banana. We extended the library in an elaboration tolerant way so that the remaining scenarios could also be captured in a direct manner. These examples gave us some insights on how $\mathcal{ALM}$ should be used for representing knowledge and for the creation of library modules.

A final example that helped us develop our methodology was a generalization of

[1] We also modeled specialized knowledge about a biological phenomenon that we will describe in Chapter IX.

the domain described in [Todorova & Gelfond, 2012]. This is a more complex motion domain in which objects belonging to different categories can enter and leave different locations that are not simple points in space. To model it, we introduced an alternative representation of motion. Additionally, modeling this example required some knowledge about tree-hierarchies. We showed how such knowledge could be added to the knowledge base, this time in a separate library, and be easily incorporated in the domain representation.

In the remainder of the chapter we describe our formalization of the experimental examples. This will give us the opportunity to discuss our knowledge representation methodology. A summary of the methodology will be given in Section 7.4.

7.1 Generalizing the Initial Motion Formalization

We assume that we have access to the modules *move_between_points*, *carrying_things*, and *grasp_and_release* from Chapter V. We modify and expand these modules by knowledge expressed in more general terms. As a consequence, we produce our first library modules.

We start with the transformation of module *move_between_points*. We remind the reader that this module contains three classes: *points* denoting locations that have no parts, *things* denoting separate and self-contained entities, and *movers* (a special case of *things*) denoting things that can move by themselves. We replace the class *movers* by a more general class called *vehicles* referring to "things that can change locations and can be used to transport other things." As a consequence, the *actor* attribute of class *move* is now of class *vehicles*. A vehicle has a power train — a collection of components necessary for it to move. Vehicles have two attributes, *capacity* (how many things it can transport) and *needs_driver* (whether it needs a driver or can move by itself):

vehicles : *things*

attributes

capacity : 0..*maxint*

We add a new class called *drivers* referring to "things that can direct the motion and course of vehicles":

$drivers : things$

We consider some new properties of the domain: points can be occupied, vehicles can have a driver, vehicles can be unusable or not. (A vehicle is unusable, for example, if some component of the vehicle's power train is not in working condition or if the capacity of the vehicle is exceeded.) To model these properties, we introduce three new functions in module *move_between_points*:

inertial $occupied : points \rightarrow booleans$

inertial $has\_driver : vehicles \rightarrow booleans$

inertial $unusable : vehicles \rightarrow booleans$

together with two new executability conditions for action class *move*:

$\neg occurs(X)$ **if** $instance(X, move)$,
$dest(X) = D$,
$occupied(D)$.

$\neg occurs(X)$ **if** $instance(X, move)$,
$actor(X) = A$,
$unusable(A)$.

and a state constraint:

$unusable(V)$ **if** $needs\_driver(V)$,
$\neg has\_driver(V)$.

We add further state constraints defining *unusable* and *has_driver* in module *carrying_things*, based on the function *holding* defined there:

$unusable(V)$ **if** $capacity(V) = N$,
$N_1 = \#count\{T : holding(V, T)\}$,
$N_1 > N$.

$has\_driver(V)$ **if** $holding(V, T)$,
$instance(T, driver)$.

In module *grasp_and_release*, we replace the attributes *actor* and *patient* of action *grasp* by *grasper* and *grasped_thing*, respectively, to make them more general; similarly for *release* where we replace these attributes by *releaser* and *released_thing*. The *grasper* and *releaser* are now of sort *vehicles*. We update the axioms of this module accordingly. We add a new relation *can_reach* to express that a vehicle has a thing within its reach. We make it a $defined$ fluent and expect the user to define it for particular domains.

defined $can\_reach : vehicles \times things \rightarrow booleans$

We add a new executability condition to this module to state that a vehicle cannot grasp another thing that is not within its reach.

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, grasp),$$
$$grasper(X) = G,$$
$$grasped\_thing(X) = T,$$
$$\neg can\_reach(G, T).$$

The resulting modules *move_between_points*, *carrying_things*, and *grasp_and_release* describe general knowledge. We store them in a library called *commonsense_motion*. The dependency relations between these modules are represented graphically in Figure 7.1.

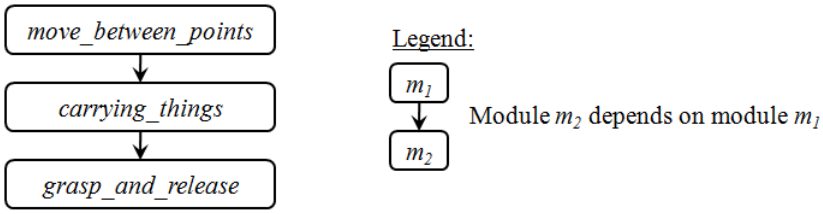

Figure 7.1: The Initial *commonsense_motion* Library

In the following two subsections, we show how we created system descriptions that formalize some benchmark examples by reusing the knowledge captured in the *commonsense_motion* library in a straightforward way.

7.1.1 Crossing a River

We first consider the River Crossing problem, a simplification of the Missionaries and Cannibals puzzle [Amarel, 1968] presented in [Erdoğan, 2008]: *Three persons want to cross a river with a boat that can hold at most two persons at a time.*

To model this domain, we reuse the information in the *commonsense_motion* library by viewing: the crossing of the river as a *carry* action performed by the boat; a person's action of boarding the boat as the boat "grasping hold" of the person; and a person's action of disembarking the boat as the boat "releasing hold" of the person. We need a new state constraint defining fluent *can_reach*: a vehicle can reach things located at the same point as itself. This axiom is not valid for all domains; it is particular to our domain. Hence, we place it in a new module of our theory, called *main*, which will depend on modules imported from the library. Note that this practice is part of our methodology of representing knowledge in $\mathcal{ALM}$: class declarations, function declarations, and axioms that are not imported from the library, but are rather particular to a scenario, are collected in module *main*. The knowledge in the *main* module (or parts of it) may finally be stored in some library if we realize that we could reuse it in modeling many other domains. In general, it is preferred to keep the *main* module small and reuse as much knowledge as possible from existing libraries.

Additionally, we specify that the boat is a vehicle with capacity 2 and that it needs a driver. The three people are drivers and carriables. This information is captured by the structure. The resulting system description for this domain, called *river_crossing*, looks as follows:

system description *river_crossing*
  theory *river_crossing*
    import *commonsense_motion*
    module *main*
      depends on *grasp_and_release*
      axioms

state constraints

$can\_reach(X, Y)$ **if** $loc\_in(X) = loc\_in(Y)$.

structure

$pers(X)$ **in** $drivers, carriables$ **where** $instance(X, 1..3)$

$boat$ **in** $vehicles$

$capacity = 2$

$needs\_driver = true$

$bank_1, bank_2$ **in** $points$

$move(V, P_1, P_2)$ **in** $carry$

$actor = V$

$origin = P_1$

$dest = P_2$

$embark(P, V)$ **in** $grasp$

$grasper = V$

$grasped\_thing = P$

$disembark(P, V)$ **in** $release$

$releaser = V$

$released\_thing = P$

7.1.2 Pednault's Briefcase Domain

The briefcase domain is described by Pednault in [Pednault, 1988] as follows: *Suppose that we have a world that consists of three objects—a briefcase, a dictionary, and a paycheck—each of which may be situated in one of two locations: the home or the office. Actions are available for putting objects in the briefcase, and for taking objects out, as well as for carrying the briefcase between two locations. Initially, the briefcase, the dictionary, and the paycheck are at home; the paycheck is in the briefcase, but the dictionary is not. The goal is to have the briefcase and dictionary at the office and the paycheck at home.*

To model this domain, we import the *commonsense_motion* library, and view the briefcase as an instance of class *vehicles*, the dictionary and paycheck as instances of *carriables*, and the two locations as instances of *points*. The action of putting objects in the briefcase is an instance of *grasp*, where the briefcase "takes hold" of one of the two objects (dictionary or paycheck). Similarly, taking objects out of the briefcase is an instance of *release*, where the briefcase "releases hold" of the object. Pednault does not consider a person carrying the briefcase in his problem description, so we do not either. We view the carrying of the suitcase between points as an instance of *move* with the briefcase as its actor. As in the formalization of the River Crossing domain, we need to add a state constraint defining *can_reach* as before. Following our methodology for knowledge representation, this executability condition would be part of a module called *main*. The resulting system description looks as follows:

system description $briefcase$

theory $briefcase$

import $commonsense\_motion$

module $main$

depends on $grasp\_and\_release$

axioms

state constraints

$can\_reach(X, Y)$ **if** $loc\_in(X) = loc\_in(Y)$.

structure

$briefcase$ **in** $vehicles$

$dictionary, paycheck$ **in** $carriables$

$home, office$ **in** $points$

$carry\_briefcase(P_1, P_2)$ **in** $move$

$actor = briefcase$

$origin = P_1$

$dest = P_2$

put_in_briefcase(*C*) **in** *grasp*

grasper = *briefcase*

grasped_thing = *C*

take_out_from_briefcase(*C*) **in** *release*

releaser = *briefcase*

released_thing = *C*

Note that the specification of the initial situation and goal particular to a scenario are not part of system description of $\mathcal{ALM}$. Hence, they do not appear in the above system description.

7.2 Extending our Motion Library: Supporters

We would now like to address other simple domains: Blocks World, Towers of Hanoi, and Monkey and Banana. Representing them by just importing the existing *commonsense_motion* library is not as straightforward as previously because these domains refer to some general knowledge that is not yet captured by the motion library. In particular, a special type of things is referenced in these domains: things that have a flat top where other things can be placed. We will call such objects *supporters*. As *supporters* are a recurrent class of objects, we create a new module describing it, called *supporter*, and add it to our *commonsense_motion* library. In general, our methodology recommends storing modules that capture general and reusable knowledge in the appropriate library, based on the theme of the module.

module *supporter*

depends on *grasp_and_release*

class declarations

supporters : *things*

attributes

top : *points*

constraints

$$top(Y) \neq U \quad \textbf{if} \quad top(X) = U,$$
$$instance(X, supporters),$$
$$instance(Y, supporters),$$
$$X \neq Y.$$

axioms

executability conditions

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),$$
$$actor(X) = A,$$
$$holding(A, T),$$
$$dest(X) = top(T).$$

The attribute constraint in this module says that two supporters cannot share the same top. The executability condition says that an actor cannot move on top of a thing that he is holding.

The extended version of the *commonsense_motion* library is illustrated graphically in Figure 7.2. In the next subsections, we show how the extended version of the library can be used to model some domains.

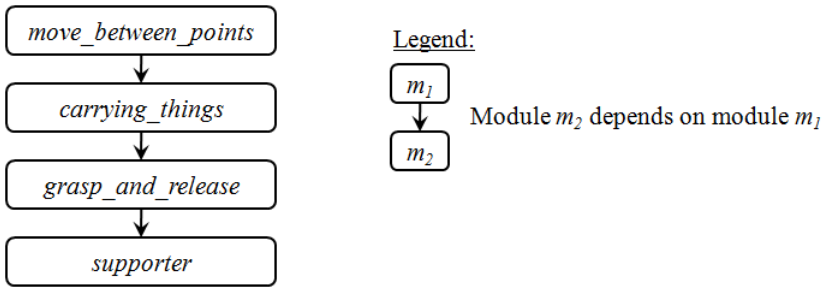

Figure 7.2: The Extended *commonsense_motion* Library

7.2.1 Blocks World

The Blocks World problem says the following: *There are some cubic blocks of the same size sitting on a table. The goal is to build different configurations of vertical stacks of blocks. Only one block may be moved at a time: it may either be placed on*

the table or placed atop another block. Only one block can sit on top of another block. Any blocks that are, at a given time, under another block cannot be moved.

To model this domain, we import the *commonsense_motion* library, now extended with the *supporter* module. For simplicity, blocks are seen as *vehicles*; they are also *supporters*, as other blocks can be stacked on top of them. The *main* module contains knowledge particular to this domain: a state constraint saying that the top of a thing is occupied if there is something located in it, and executability conditions saying that a thing with an occupied top cannot move and two things cannot move simultaneously.

system description *blocks_world*

theory *blocks_world*

import *commonsense_motion*

module *main*

depends on *supporter*

axioms

state constraints

$$occupied(P) \quad \textbf{if} \quad top(T) = P,\ loc\_in(X) = P.$$

executability conditions

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = B,\ top(B) = P,\ occupied(P).$$

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ instance(Y, move),\ occurs(Y),\ X \neq Y.$$

This was the theory of our system description. We now define the structure corresponding to a particular domain in which there are n blocks, where n is some numerical constant. The first block is denoted by the constant $b(1)$, the second block

by $b(2)$, and so on. For each of the n blocks there is an instance of class *points* that represents the top of that block. The top of block $b(X)$, where X ranges from 1 to n, is denoted by $t(X)$ (see the third instance definition below). Hence, the top of the first block is represented by the constant $t(1)$, the top of the second block by $t(2)$, etc. The table itself is also an instance of *points*. Note the use of "where" statements in instance definitions, which allows us to define multiple instances in a compact way.

structure

$table$ **in** $points$

$t(X)$ **in** $points$ **where** $instance(X, 1..n)$

$b(X)$ **in** $vehicles, supporters$ **where** $instance(X, 1..n)$

$top = t(X)$

$move(B, P_1, P_2)$ **in** $move$

$actor = B$

$origin = P_1$

$dest = P_2$

7.2.2 Towers of Hanoi

The classical puzzle of the Towers of Hanoi can be modeled in a similar way to the Blocks World. This puzzle says the following: *There are three pegs and several disks of different sizes. Initially the disks are stacked in the ascending order of their size on one peg, with the smallest disk at the top. The objective is to move the entire stack to another peg, by moving disks one at the time, with the constraint that no disk may be placed on top of a smaller disk.*

To represent this domain, we introduce a class *disks* that is a special case of *vehicles* and *supporters* and has an additional attribute: its size. Notice that the creation of the new class is needed because of the added attribute. State constraints specify that only one thing can be located in each point in space and that the disk on top of another disk has to have a smaller size. An executability condition prevents

disks with an occupied top from moving and another one prohibits bigger disks to be moved on top of smaller disks.

system description $towers\_of\_hanoi$

theory $towers\_of\_hanoi$

import $commonsense\_motion$

module $main$

depends on $supporter$

class declarations

$disk : vehicles, supporters$

attributes

$size : 1..maxint$

axioms

state constraints

$occupied(P)$ **if** $loc\_in(D) = P.$

$loc\_in(D) \neq top(D_1)$ **if** $size(D) \geq size(D_1).$

executability conditions

$\neg occurs(X)$ **if** $instance(X, move),$
$actor(X) = A,$
$top(A) = P,$
$occupied(P).$

$\neg occurs(X)$ **if** $instance(X, move),$
$actor(X) = D,$
$dest(X) = top(D_1),$
$size(D) \geq size(D_1).$

In the structure below, we assume that there are n disks, where n is some numerical constant. Disks are denoted by terms of the type $d(X)$ where X ranges from 1 to n. Each disk will have an instance of sort *points* as its top. The top of disk $d(1)$ will be denoted by $t(1)$, and so on. The size of a disk will be equal to the index of the disk. For instance, the size of $d(1)$ is 1. The three pegs are instances of *points*.

structure

peg_1, peg_2, peg_3 **in** *points*

$t(X)$ **in** *points* **where** $instance(X, 1..n)$

$d(X)$ **in** *disks* **where** $instance(X, 1..n)$

$top = t(X)$

$size = X$

$move(D, P_1, P_2)$ **in** *move*

$actor = D$

$origin = P_1$

$dest = P_2$

7.2.3 Monkey and Banana

Let us now consider the Monkey and Banana problem: *A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. In the room there is also a box. The ceiling is just the right height so that a monkey standing on the box under the bananas can reach the bananas. The monkey can move around, carry other things around, climb on the box, and reach for the bananas. What is the best sequence of actions for the monkey to get the bananas?*

We model this domain by importing the *commonsense_motion* library, and declaring in the *main* module three new sorts specific to the problem: *monkeys*, which are *vehicles*; *boxes* that are both *supporters* and *carriables*; and *bananas*, which are carriables. We also declare a new function: the static $under : points \times points \rightarrow booleans$ specifying whether one point in space is exactly under another one. The definition of *can_reach* in module *main* says that a monkey can reach a box if it is located where the box is and a monkey can reach the bananas if it is located on top of a box that is located exactly under the bananas. The executability condition says that a vehicle can only get on top of supporters located in the same point as him.

system description *monkey_and_bananas*

theory *monkey_and_bananas*

import *commonsense_motion*

module *main*

depends on *supporter*

class declarations

$monkeys : vehicles$

$boxes : supporters, carriables$

$bananas : carriables$

function declarations

static $under : points \times points \rightarrow booleans$

axioms

state constraints

$can\_reach(M, B)$ **if** $instance(M, monkeys)$,
$instance(B, boxes)$,
$loc\_in(M) = loc\_in(B)$.

$can\_reach(M, Ban)$ **if** $instance(M, monkeys)$,
$instance(Ban, bananas)$,
$instance(B, boxes)$,
$loc\_in(M) = top(B)$,
$under(loc\_in(B), loc\_in(Ban))$.

executability conditions

$\neg occurs(X)$ **if** $instance(X, move)$,
$actor(X) = M$,
$dest(X) = top(B)$,
$loc\_in(M) \neq loc\_in(B)$.

This was the theory of our system description. We now define its structure. In particular, we define five points in space, which we name: *box_top* to denote the top

of the box; *ceiling* for the initial location of the bananas; *under_bananas* for the point exactly under *ceiling*; *initial_monkey* for the initial location of the monkey; and *initial_box* for the initial location of the box.

structure

box_top, *ceiling*, *under_bananas*, *initial_monkey*, *initial_box* **in** *points*

m **in** *monkeys*

banana_bunch **in** *bananas*

box **in** *boxes*

top = *box_top*

move(*A*, *P*) **in** *move*

actor = *A*

dest = *P*

carry(*A*, *C*, *P*) **in** *carry*

actor = *A*

carried_thing = *C*

dest = *P*

grasp(*A*, *T*) **in** *grasp*

grasper = *A*

grasped_thing = *T*

7.3 Extending our Motion Library: Areas

Notice that so far, the spatial layout of the domain was characterized by a collection of points in space, where a thing could only be located in one point at a time. This kind of layout may be too simplistic for some scenarios, for instance when dealing with areas that are part of other areas (e.g., a city that is part of a country that is part of a continent, etc.). To characterize such a spatial layout, we introduce a class *areas* (referring to roughly bounded parts of space or surface having some specific characteristic or function) and two relations, *within* and *disjoint*, defined on areas. Module *basic_geography* below encodes this information.

module *basic_geography*

class declarations

areas

function declarations

static $within : areas \times areas \rightarrow booleans$

static $disjoint : areas \times areas \rightarrow booleans$

axioms

state constraints

$within(A_1, A_2)$ **if** $within(A_1, A)$, $within(A, A_2)$.

$\neg within(A_2, A_1)$ **if** $within(A_1, A_2)$, $A_1 \neq A_2$.

$\neg within(A_1, A_2)$ **if** $disjoint(A_1, A_2)$.

$disjoint(A_2, A_1) \equiv disjoint(A_1, A_2)$.

$disjoint(A_1, A_2)$ **if** $within(A_1, A_3)$, $disjoint(A_2, A_3)$.

$\neg disjoint(A_1, A_2)$ **if** $within(A_1, A_2)$.

$\neg disjoint(A, A)$.

To characterize motion between areas, we introduce a new module. This new module, called *move_between_areas*, is similar to *move_between_points* but contains an action class *move* whose origin and destination are *areas*. A new fluent is required to characterize the location of things, as one thing can now be located in several areas if those areas are situated one within another. We call this new fluent *in*, where $in : things \times areas \rightarrow booleans$.

module *move_between_areas*

depends on *basic_geography*

class declarations

things

$vehicles : things$

$move$: **actions**

attributes

$actor : vehicles$

$origin : areas$

$dest : areas$

function declarations

inertial $in : things \times areas \rightarrow booleans$

axioms

dynamic causal laws

$$in(O, A) \quad \textbf{if} \quad instance(X, move),\ occurs(X),\ actor(X) = O,\ dest(X) = A.$$

$$\neg in(O, A) \quad \textbf{if} \quad instance(X, move),\ occurs(X),\ actor(X) = O,\ origin(X) = A.$$

state constraints

$$in(O, A2) \quad \textbf{if} \quad within(A_1, A_2),\ \ in(O, A_1).$$

$$\neg in(O, A2) \quad \textbf{if} \quad disjoint(A_1, A_2),\ \ in(O, A_1).$$

executability conditions

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = O,\ dest(X) = A,\ in(O, A).$$

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = O,\ origin(X) = A,\ \neg in(O, A).$$

The knowledge captured by modules *basic_geography* and *move_between_areas* is quite general. Hence, we assume that the two modules are included in the library *commonsense_motion*. The new structure of the library can be seen in Figure 7.3.

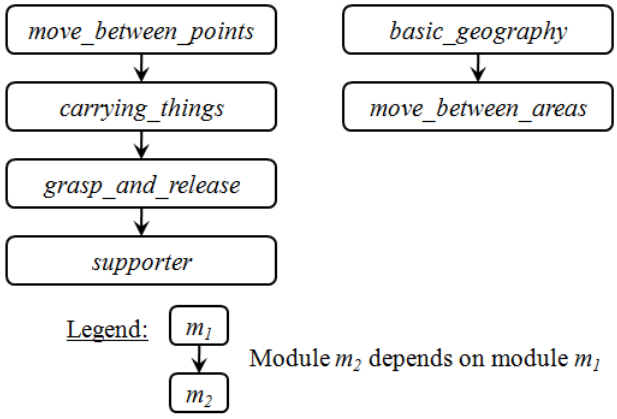

Figure 7.3: The Second Extension of the *commonsense_motion* Library

At this point, the library contains two alternative declarations of action class *move*: one in which the origin and destination are *points* and another one in which they are *areas*. The user may decide which one to select depending on the characteristics of the domain to model. The user should no longer import the whole library in his system description, but rather import only the module(s) corresponding to the desired type of *move*. If a user imported both modules, *move_between_points* and *move_between_areas*, the result would *not* be a theory according to Definition 8 because the class *move* would be declared twice in the resulting collection of modules. Moreover, the declarations of *move* are different in the two modules: its attributes *origin* and *dest* are of type *points* in *move_between_points* and of type *areas* in *move_between_areas*.

Next, we show an application of *move_between_areas* to modeling travel domains.

7.3.1 Travel Domain

Let us consider a travel domain described in [Todorova & Gelfond, 2012]: *There are several areas and a hierarchy of classes of movable objects. The movement of an*

object into an area changes not only its position, but also the cardinality of the sets of objects from different classes located in this (and other) areas. Objects of the domain can execute two types of actions: enter into an area or leave an area.

The modules *basic_geography* and *move_between_areas* are appropriate for representing this domain. In addition, this domain requires some representation of hierarchies of classes—in this case, tree hierarchies: "a collection of classes connected by arrows, which indicate the subclass relation between classes." Tree hierarchies are a pervasive type of data structure, so we decided to create a separate module that describes them. The module, called *tree_hierarchy*, is part of a second library, *data_structures*. We create a class *nodes* to refer to nodes of the tree hierarchy; nodes have an attribute $parent : nodes \rightarrow nodes$ that describes arcs of the hierarchy. Several statics are used to describe other properties of the tree hierarchy.

module *tree_hierarchy*

class declarations

nodes

attributes

$parent : nodes$

function declarations

static $root : nodes \rightarrow booleans$

static $leaf : nodes \rightarrow booleans$

static $descendant : nodes \times nodes \rightarrow booleans$

axioms

state constraints

$root(X)$ **if** $N = \#count\{Y : parent(X) = Y\}$, $N = 0$.

$\neg root(X)$ **if** $N = \#count\{Y : parent(X) = Y\}$, $N > 0$.

$leaf(X)$ **if** $N = \#count\{Y : parent(Y) = X\}$, $N = 0$.

$$\neg leaf(X) \quad \textbf{if} \quad N = \#count\{Y : parent(Y) = X\},$$
$$N > 0.$$
$$descendant(C_1, C_2) \quad \textbf{if} \quad parent(C_1) = C_2.$$
$$descendant(C_1, C_2) \quad \textbf{if} \quad parent(C_1) = C_3,$$
$$descendant(C_3, C_2).$$

In addition, Todorova and Gelfond define an *instance of a class hierarchy* as "a set of objects called the *universe* of the hierarchy, and a mapping of class names into subsets of the universe, which respects the hierarchy's subclass relation." We place the formalization of an instance of a hierarchy in a separate module called *populated_tree_hierarchy*, which depends on our previous module and is stored in the same library, *data_structures*. (This would allow for the independent use of *tree_hierarchy* when dealing with scenarios that do not mention the instantiation of a hierarchy.) We declare a sort *objects*. To express the mapping of an object into a node of the hierarchy, we add an attribute *m_link* to class *objects*.

module *populated_tree_hierarchy*

depends on *tree_hierarchy*

class declarations

objects

attributes

$m\_link : nodes$

function declarations

static $member : objects \times nodes \rightarrow booleans$

static $is\_defined : objects \rightarrow booleans$

static $universe\_cardinality : 0..maxint$

axioms

state constraints

$member(O, C)$ **if** $m\_link(O) = C.$

$$member(O, C_2) \quad \textbf{if} \quad parent(C_1) = C_2,$$
$$member(O, C_1).$$
$$\neg member(O, C_2) \quad \textbf{if} \quad member(O, C_1),$$
$$parent(C_1) = parent(C_2),$$
$$C_1 \neq C_2.$$
$$is\_defined(O) \quad \textbf{if} \quad leaf(C),$$
$$member(O, C).$$

After introducing the new library called *data_structures*, our knowledge base consists of the two libraries illustrated in Figure 7.4. Of course, both libraries can be expanded in the future by new modules.

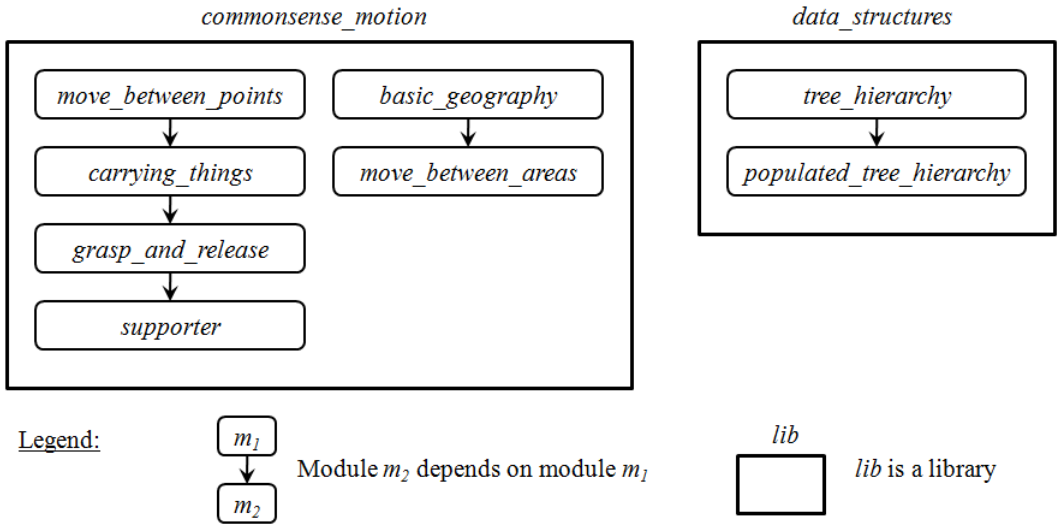

Figure 7.4: Our Knowledge Base

Let us now consider one of the scenarios in [Todorova & Gelfond, 2012] (Example 1, adapted here): *Professor D and two of his students, A and B, entered the empty room. They were immediately followed by professor C. Then, A left the room. Is A in the room? How many people are in the room? How many students? How many professors?* We formalize it by the following system description:

system description *travel_domain*

theory *travel_domain*

import *move_between_areas* **from** *commonsense_motion*

import *populated_tree_hierarchy* **from** *data_structures*

module *main*

depends on *move_between_areas*

depends on *populated_tree_hierarchy*

axioms

state constraints

$disjoint(A_1, A_2)$ **if** $A_1 \neq A_2$.

$\neg within(A, A)$.

This was the theory of our system description. For the particular scenario above, we define the following structure. Note that a, b, c, and d are both *vehicles* and *objects* of the tree hierarchy. The actions *enter* and *leave* are defined as instances of *move* with one unspecified attribute (*origin* and *dest*, respectively).

structure

$room$ **in** $areas$

$people$ **in** $nodes$

$professors$ **in** $nodes$

$parent = people$

$students$ **in** $nodes$

$parent = people$

a, b **in** $vehicles, objects$

$m\_link = students$

c, d **in** $vehicles, objects$

$m\_link = professors$

$enter(X, Y)$ **in** $move$

$actor = X$

$dest = Y$

$leave(X, Y)$ **in** $move$

$actor = X$

$origin = Y$

Let us now connect our $\mathcal{ALM}$ representation of travel domains with the ASP representation presented in [Todorova & Gelfond, 2012]. For that, we will introduce some preliminary definitions and then formulate a proposition.

Let LF be a travel logic form as defined in [Todorova & Gelfond, 2012]. Let $\mathcal{M}^n$ and $\mathcal{P}_1^n(LF)$ be the programs defined in [Todorova & Gelfond, 2012].

By $\mathcal{D}(LF)$ we denote the system description consisting of a theory formed by modules *move_between_areas* and *populated_tree_hierarchy* defined above, and a structure $\mathcal{S}$ defined as follows:

- For every fact

  $$root(c)$$

  in LF, $\mathcal{S}$ contains a statement

  $$c \textbf{ in } nodes$$

- For every fact

  $$c\_link(c_1, c_2)$$

  in LF, $\mathcal{S}$ contains a statement

  $$c_1 \textbf{ in } nodes$$
  $$parent = c_2$$

- For every fact

  $$m\_link(o, c)$$

in LF, $\mathcal{S}$ contains a statement

$$o \textbf{ in } objects, vehicles$$
$$m\_link = c$$

- For every fact

$$area(a)$$

in LF, $\mathcal{S}$ contains a statement

$$a \textbf{ in } areas$$

- $\mathcal{S}$ contains the statements

$$enter(X, Y) \textbf{ in } move$$
$$actor = X$$
$$dest = Y$$
$$leave(X, Y) \textbf{ in } move$$
$$actor = X$$
$$origin = Y$$

Let $\Pi^n(\mathcal{D}(LF))$ be the ASP encoding of $\mathcal{D}(LF)$ described in 6.2, where the variable I for time steps ranges from 0 to n.

Let $\Pi^n(LF)$ be the program obtained from $\Pi^n(\mathcal{D}(LF))$ by: (1) renaming sorts *nodes* as *class*, *objects* as *object*, and *areas* as *area*, and relation *parent* as *c_link*; (2) adding all facts of the form $hpd(a, i)$ or $obs(f, v, i)$ from LF; (3) adding the rules that define predicates *answer* from $\mathcal{M}^n$; and (4) adding the following rules:

$$holds(F, 0) \leftarrow obs(F, true, 0).$$
$$\neg holds(F, 0) \leftarrow obs(F, false, 0).$$

Proposition 1. For any *complete* logic form LF, $\Pi^n(LF)$ and $\mathcal{P}_1^n(LF)$ are equivalent in terms of their answer sets.

This ends our illustration of the methodology of representing knowledge in $\mathcal{ALM}$. In the next section we briefly summarize the key points of our methodology.

7.4 Summary of Knowledge Representation Methodology

The following practices are recommended when representing knowledge in $\mathcal{ALM}$.

- Reuse knowledge from the existing libraries when appropriate.
- Place knowledge particular to a specific class of domains in a module called *main* of the theory of your system description.
- Group knowledge into modules based on the theme of such knowledge.
- Separate knowledge into two or more modules to maintain a manageable module size and to allow for the independent use of knowledge.
- Maintain a low height for the dependency hierarchy corresponding to modules in a library.
- Remember that the initial situation and goal(s) of a scenario are not part of an $\mathcal{ALM}$ system description.

CHAPTER VIII
METHODOLOGY OF $\mathcal{ALM}$'S USE IN SOLVING COMPUTATIONAL TASKS

As mentioned in the introduction, one of the goals of Artificial Intelligence is to learn how to create software components for intelligent agents capable of reasoning about, and acting in dynamic domains. Intelligent agents are assumed to be equipped with mathematical models of the dynamic domains and some reasoning algorithms. In the previous chapter, we showed how concise models of discrete dynamic domains can be created in an elaboration tolerant way using modular action language $\mathcal{ALM}$. We now turn our attention to reasoning algorithms for solving different computational problems and to how these algorithms can be used together with domain descriptions written in our language. In particular, we describe a methodology of using $\mathcal{ALM}$ in solving various tasks such as temporal projection[1], planning[2], diagnosis[3], or question answering [4]. Our methodology is based on previous work [Baral et al., 1997, Lifschitz, 1999, Baral & Gelfond, 2000, Balduccini & Gelfond, 2003a, Balduccini, 2004, Baral et al., 2004, Balduccini, 2005, Baral et al., 2005, Balduccini et al., 2008, Gelfond & Kahl, 2012] in which action languages and ASP, or its extension CR-Prolog [Balduccini & Gelfond, 2003b], were used to perform these tasks.

In the remainder of this chapter, we present the outline of our general methodology for $\mathcal{ALM}$'s use in solving computational tasks in Section 8.1. We then exemplify this methodology for some tasks: temporal projection in Section 8.2, temporal projection with intended actions in Section 8.3, planning in Section 8.4, and diagnosis in Section 8.5. We briefly discuss the particularities of question answering in Section 8.6. An extensive example on question answering will be presented in Chapter IX.

[1]Determining the values of fluents of the domain along a given trajectory.

[2]Finding a sequence of actions which may take the system from the current state to a goal state.

[3] Finding explanations for unexpected observations.

[4]Answering questions posed in natural language about a text also provided in natural language.

8.1 General Methodology

Imagine that we wanted to create an automated and rational intelligent agent capable of solving one or more computational tasks in the context of some dynamic domains. To do that, we would provide the intelligent agent with some background knowledge:

- A collection of $\mathcal{ALM}$ library modules and/or system descriptions. We call this collection the *library*. The library may include both commonsense and expert knowledge.

- A collection of ASP/CR-Prolog programs for solving computational tasks. We call such programs *reasoning modules*[5], based on the terminology in [Balduccini & Gelfond, 2003a, Balduccini, 2005]. Reasoning modules are domain-independent.

We would formalize the knowledge in the library using the methodology described in Chapter VII. To create the ASP/CR-Prolog reasoning modules, we would adopt established methods from the literature.

The input for our intelligent agent would be a scenario, which includes the description of a domain, the evolution of the domain up to some time step (i.e., a *recorded history*), and the task to be solved. For certain tasks (e.g., question answering), the agent may receive the input in the form of a text written in natural language. In this case, the agent would have to automatically extract the relevant information contained by the scenario and translate it into some logic form that it can later use for solving the task. In most cases, however, we would manually process the input for the agent.

In the end, the relevant information in a scenario would be specified as a tuple consisting of:

- a system description $\mathcal{D}$

[5]Reasoning modules should not be confused with modules of $\mathcal{ALM}$.

- a recorded history Γ_n of $\mathcal{D}$ up to the current time step n
- a reasoning module Ω
- additional information $\mathcal{Q}$ defining the goal, query, etc.

$\mathcal{D}$ may either be a system description from the library or a system description obtained by putting together some modules from the library with instance definitions extracted from the scenario.

The recorded history Γ_n of $\mathcal{D}$ up to step n is a collection of *observations*, i.e., statements of the type:

$$obs(eq(f, v), i)$$
$$obs(neq(f, v), i)$$

where $0 \leq i \leq n$, saying that, in the i^{th} state of the trajectory, function f was observed to have (or not to have) value v; and

$$hpd(a, i)$$
$$\neg hpd(a, i)$$

where $0 \leq i < n$, saying that elementary action a was observed to have happened (or not to have happened) at the i^{th} step of the trajectory. For diagnosis, observations must be limited to the state preceding the current one, i.e., $0 < i < n$ in *obs* atoms. For certain scenarios, the history may also include facts of the type

$$intend(\alpha, i)$$

where $0 \leq i < n$, saying that the execution of action or sequence of actions α was intended at step i.

The additional information would depend on the task to be solved. In the case of temporal projection, there may either be no additional information (if we are interested in the values of all fluents along a trajectory) or the additional information $\mathcal{Q}$ may consist of atoms of the type:

$$query(f, i)$$

inquiring about the value of function f at some time step i.

For planning problems, the goal would be provided. A goal is a collection G of user-defined fluent literals over the signature of $\mathcal{D}$. The goal would be encoded by the additional information $\mathcal{Q}$ using the relation h defined in Section 6.2:[6]

$$\begin{aligned} goal(I) \quad \leftarrow \quad & h(G_{pos}, I), \\ & \neg h(G_{neg}, I). \end{aligned}$$

where G_{pos} is the subset of positive literals in G and G_{neg} is the subset of negative literals in G.

Normally, no additional information is specified for diagnosis problems.

A query $\mathcal{Q}$ would be provided for question answering tasks. This may resemble the queries for temporal projection or may have a more complex form.

The intelligent agent is expected to produce as an output the solution(s) to the given task. According to our methodology, the software component emulating the intelligent agent would find the solutions by performing the following steps:

1. Translate the system description $\mathcal{D}$ into a logic program $\Pi_n(\mathcal{D})$ according to the semantics described in Section 6.2, where the sort *step* ranges from 0 to n.

2. Create a logic program $\mathcal{P}$ by putting together the program $\Pi_n(\mathcal{D})$, the recorded history Γ_n, the reasoning module Ω, and the additional information $\mathcal{Q}$:

$$\mathcal{P} =_{def} \Pi_n(\mathcal{D}) \cup \Gamma_n \cup \Omega \cup \mathcal{Q}$$

3. Compute the answer sets of $\mathcal{P}$ using either a general purpose ASP solver (e.g., CLASP [Gebser et al., 2007], DLV [Leone et al., 2006], SMODELS

[6] If $f = v$ is an atom and f is a term obtained from a static fluent, then

$$h(val(f, v),\ i) =_{def} val(f, v) \text{ and } h(has\_val(f),\ i) =_{def} has\_val(f)$$

whereas if f is obtained from an inertial or defined fluent, then

$$h(val(f, v),\ i) =_{def} holds(val(f, v), i) \text{ and } h(has\_val(f),\ i) =_{def} holds(has\_val(f), i).$$

If σ is a collection of function atoms, then $h(\sigma, i) =_{def} \{h(l, i) : l \in \sigma\}$.

[Niemelä & Simons, 1997], CMODELS [Giunchiglia et al., 2004b]) or the inference engine CRMODELS [Balduccini & Gelfond, 2003b, Balduccini, 2007a] for CR-Prolog.

Answer sets of $\mathcal{P}$ will correspond to solutions to the given task.

Note that the only thing that changes for different computational tasks are the reasoning module Ω and the additional information $\mathcal{Q}$ that are part of program $\mathcal{P}$.

In the next sections we show how different problems are solved using this approach. For that, we consider some examples and assume that our intelligent agent has access to a library consisting of modules *move_between_points*, *carrying_things*, and *graps_and_release* from Chapter V.

8.2 Temporal Projection

Let us imagine that our intelligent agent has to solve the temporal projection problem described by the following scenario:

S_1 : *"John was in Paris. He was holding his suitcase. Then, John went to London. Where was his suitcase at the end of the story?*

As we are not focusing on answering questions from natural language, we extract the relevant information from the scenario for the agent. We first need to obtain a system description $\mathcal{D}$ describing this domain. It is clear that the scenario is about things that move between points in space and about things that can be carried around. Hence, we select modules *move_between_points* and *carrying_things* from our library and import them in the theory of $\mathcal{D}$. The scenario talks about a thing that can move by itself (John), a thing that can be carried (the suitcase), some points in space (Paris and London), and a moving action (John going to London). Hence, we add to $\mathcal{D}$ the following structure:

structure

john **in** *movers*

suitcase **in** *carriables*

$paris, london$ **in** $points$

$move(john, london)$ **in** $move$

$actor = john$

$dest = london$

Next, we extract the recorded history $\Gamma_1(S_1)$, where 1 is the current step:

$obs(eq(loc\_in(john), paris), 0)$

$obs(eq(holding(john, suitcase), true), 0)$

$hpd(move(john, london), 0)$

The question at the end of the scenario indicates a temporal projection task. We select the reasoning module Ω_{tp} for temporal projection from the agent's background knowledge. Ω_{tp} connects the predicates *obs* and *hpd* from the history with hypothetical relations *holds* and *occurs*, respectively, which describe the transition diagram of the domain (see Chapter VI). In Ω_{tp}'s definition, we will reuse the relation h from Section 6.2 and use the variable I for steps.

Ω_{tp} is defined, based on previous work in [Baral & Gelfond, 2000], as follows:

- For every action instance a in $\mathcal{D}$, Ω_{tp} contains the rules:

$$\begin{aligned} occurs(A, I) &\leftarrow hpd(A, I). \\ \neg occurs(A, I) &\leftarrow \neg hpd(A, I). \\ h(val(F, V), 0) &\leftarrow obs(eq(F, V), 0). \\ \neg h(val(F, V), 0) &\leftarrow obs(neq(F, V), 0). \end{aligned}$$

  together with the Reality Check Axioms:

$$\begin{aligned} \leftarrow\ & obs(eq(F, V), I), \\ & not\ h(val(F, V), I), \\ & function(F), \\ & range(F, V). \end{aligned}$$

$$\begin{aligned} \leftarrow \quad & obs(neq(F,V),I), \\ & not\ \neg h(val(F,V),I), \\ & function(F), \\ & range(F,V). \end{aligned}$$

and rules to solve queries:

$$\begin{aligned} answer(val(F,V),I) \quad \leftarrow \quad & query(F,I), \\ & h(val(F,V),I). \end{aligned}$$

Finally, we extract the additional information $\mathcal{Q}(S_1)$ about the query from the scenario:

$query(loc\_in(suitcase),1)$

saying that we are interested in the location of the suitcase at 1, the last step of the story.

We give as an input to the intelligent agent the tuple $\langle\ \mathcal{D},\ \Gamma_1(S_1),\ \Omega_{tp},\ \mathcal{Q}(S_1)\ \rangle$. The agent translates $\mathcal{D}$ into a logic program, assembles the program $\mathcal{P}_1 = \Pi_1(\mathcal{D})\ \cup\ \Gamma_1(S_1)\ \cup\ \Omega_{tp}\ \cup\ \mathcal{Q}(S_1)$, and computes the answer sets of $\mathcal{P}_1$ using some ASP solver. The unique answer set of $\mathcal{P}_1$ will contain the atom:

$answer(val(loc\_in(suitcase),london),1)$

indicating that the solution to the given temporal projection task is that the suitcase is located in London at the end of the story.

8.3 Temporal Projection with Intentions

We now consider a different scenario:

S_2 : *"John was in Paris. He was holding his suitcase. John planned to go to London. Where was his suitcase at the end of the story?*

The system description $\mathcal{D}$ extracted from this scenario is the same as the one in the previous section. The history $\Gamma_1(S_2)$ is different in that the observation $hpd(move(john,london),0)$ is replaced by $intend(move(john,london),0)$:

$obs(eq(loc\_in(john), paris), 0)$

$obs(eq(holding(john, suitcase), true), 0)$

$intend(move(john, london), 0)$

A reasoning module Ω_{tpi} for temporal projection with intentions is selected. Ω_{tpi} extends Ω_{tp} from Section 8.2 by an ASP encoding of a theory of intentions (see Appendix C). In particular, the theory of intentions contains the *non-procrastination axiom* saying that *"Normally intended actions are executed the moment such execution becomes possible"*:

$$\begin{aligned} occurs(A, I) \leftarrow\ & instance(A, actions), \\ & intend(A, I), \\ & not\ \neg occurs(A, I). \end{aligned}$$

and the *persistence axiom* saying that *"Unfulfilled intentions persist"*:

$$\begin{aligned} intend(A, I+1) \leftarrow\ & instance(A, actions), \\ & intend(A, I), \\ & \neg occurs(A, I), \\ & not\ \neg intend(A, I+1). \end{aligned}$$

The additional information $\mathcal{Q}(S_2)$ is identical to $\mathcal{Q}(S_1)$, as we are still interested in the location of the suitcase at the end of the story. The intelligent agent assembles a program $\mathcal{P}_2$ by putting together the translation of $\mathcal{D}$, the history $\Gamma_1(S_2)$, the reasoning module Ω_{tpi}, and the additional information $\mathcal{Q}(S_2)$. The answer set of program $\mathcal{P}_2$ will contain

$answer(val(loc\_in(suitcase), london), 1)$

$occurs(move(john, london), 0)$

indicating that the suitcase is located in London at the end of the story (i.e., at time step 1), as John's intended action of going to London was fulfilled at time step 0.

8.4 Planning

We consider the scenario:

S_3. *John is in Paris. He is holding his suitcase. Can he get his suitcase to London?*

This requires solving a planning problem. As before, we use the same system description $\mathcal{D}$. The history $\Gamma_1(S_3)$ looks as follows:

$obs(eq(loc\_in(john), paris), 0)$

$obs(eq(holding(john, suitcase), true), 0)$

(It is similar to the previous history but it contains no *hpd* or *intend* atoms.) Additional information specifying the goal is provided by $\mathcal{Q}(S_3)$:

$$goal(I) \leftarrow holds(val(loc\_in(suitcase), london), I).$$

A reasoning module Ω_{pl} for planning is selected [Lifschitz, 1999, Balduccini, 2004, Gelfond & Kahl, 2012] . Ω_{pl} is a CR-Prolog module extending Ω_{tp} by the following rules:

$$\begin{aligned}
success &\leftarrow goal(I),\\
&\quad\ I \leq n.\\
&\leftarrow not\ success.\\
r_1(A, I) : occurs(A, I) &\stackrel{+}{\leftarrow} instance(A, actions).\\
something_happened(I) &\leftarrow occurs(A, I).\\
&\leftarrow not\ something_happened(I),\\
&\quad\ something_happened(I + 1).
\end{aligned}$$

It computes minimal plans by the use of the cr-rule r_1 and the two regular rules that follow it.

The intelligent agent would use the inference engine CRMODELS to compute answer sets of the program $\mathcal{P}_3$ resulting from putting together all the above mentioned parts. $\mathcal{P}_3$ will have one answer set containing:

$goal(1)$

$occurs(move(john, london), 0)$

This means that John can indeed get his suitcase to London, and that he can achieve his goal at step 1 by executing action $move(john, london)$ at step 0.

Notice that most of the scenarios presented in Chapter VII (River Crossing, Pednault's Briefcase, Blocks World, Towers of Hanoi, Monkey and Banana) require solving a planning problem as they mention a "goal" or an "objective".

8.5 Diagnosis

Finally, let us assume that we want our intelligent agent to solve the task in the scenario:

S_4. *Bob was in Paris. John was also in Paris. John was holding his suitcase. Later on, he noticed that he no longer had his suitcase. What happened? Why was his suitcase missing?*

This scenario describes a diagnosis problem.

Let us assume that for this scenario, we extract a system description $\mathcal{D}(S_4)$, which extends our previous system description $\mathcal{D}$ by the declaration of a special type of actions, *exogenous_actions*, in the theory of $\mathcal{D}(S_4)$:

$exogenous\_actions : actions$

and the definition of mover Bob and of an exogenous grasping action $grasp(bob, suitcase)$ in the structure of $\mathcal{D}(S_4)$:

bob **in** $movers$

$grasp(bob, suitcase)$ **in** $grasp, exogenous\_actions$

$actor = bob$

$patient = suitcase$

We extract from the scenario a history $\Gamma_2(S_4)$, where 2 is the current time step:

$obs(eq(loc\_in(bob), paris), 0)$

$obs(eq(loc\_in(john), paris), 0)$

$obs(eq(holding(john, suitcase), true), 0)$

$obs(eq(holding(john, suitcase), false), 1)$

Notice that observations end at step 1, before the current step 2. No additional information is provided.

A reasoning module Ω_{diag} for diagnosis is selected. This is a CR-Prolog module adopted from [Balduccini & Gelfond, 2003a, Gelfond & Kahl, 2012]. It consists of the rules in Ω_{tp} and the cr-rule:

$$occurs(A, I) \stackrel{+}{\leftarrow} instance(A, exogenous\_actions),$$
$$0 \leq I < n.$$

where n is a numerical constant for the current time step; for scenario S_4, $n = 2$.

The intelligent agent will produce a CR-Prolog program $\mathcal{P}_4$ by putting together all the necessary parts. The program will have a unique answer set containing the atom:

$occurs(grasp(bob, suitcase), 0)$

This says that the explanation for the unexpected observation that John no longer had his suitcase at step 1 is that Bob grasped it at step 0. Solutions to diagnosis problems will depend on the exogenous actions appearing in the selected system description given as an input to the intelligent agent.

8.6 Question Answering

In this section we briefly discuss the particularities of question answering (QA) in comparison with other computational tasks.

An intelligent agent capable of solving QA tasks normally receives as an input a scenario *written in natural language*, consisting of a text and a question. It uses methods of computational linguistics [Balduccini et al., 2008] to analyze this information: it extracts a relevant $\mathcal{ALM}$ system description $\mathcal{D}$ for the text, a reasoning module Ω for the task described in the query, and translates the scenario into a suitable logic form. The text is translated into a recorded history Γ_n and the question is translated into a query $\mathcal{Q}$. Note that a scenario given in natural language and its automated analysis are normally not required in the case of other computational tasks such as temporal projection, planning, or diagnosis.

After this part is accomplished, the QA agent performs the same steps as for other tasks [Baral et al., 2004, Baral et al., 2005, Balduccini et al., 2008]: takes the tuple $\langle\ \mathcal{D}, \Gamma_n, \Omega, \mathcal{Q}\ \rangle$, produces an ASP or CR-Prolog program $\mathcal{P}$ out of it, and computes answer sets of $\mathcal{P}$. These answer sets will correspond to answers to the input question.

In the next chapter, we will present a QA application based on $\mathcal{ALM}$ and ASP.

CHAPTER IX
A CASE STUDY: APPLICATION OF $\mathcal{ALM}$ TO QUESTION ANSWERING

In this chapter we report on the use of $\mathcal{ALM}$ and of our Question Answering (QA) methodology to create a QA system called $\mathcal{ALMAS}$ for a real-life application, the Digital Aristotle developed by Project Halo.[1] Project Halo is a research effort by Vulcan Inc. towards the development of a "Digital Aristotle" – *"an application containing large volumes of scientific knowledge and capable of applying sophisticated problem-solving methods to answer novel questions"* [Gunning et al., 2010]. The initial system was only able to reason and answer questions about *static* domains. It lacked a methodology for answering questions about *dynamic* domains: at that time, it was not clear how to represent and reason about such domains in $\mathcal{F}$LORA-*2* [Kifer, 2005], the language used in the Digital Aristotle. $\mathcal{F}$LORA-*2* is an object-oriented declarative language based on the well-founded semantics of logic programs [Van Gelder et al., 1991]. The inference engine with an identical name is based on XSB [Chen & Warren, 1996]. Our task within Project Halo was to create a $\mathcal{F}$LORA-*2* methodology for answering questions about temporal projection in dynamic domains.

Our goal was to see if $\mathcal{ALM}$ facilitates the design of knowledge bases containing the corresponding background knowledge and if provable correct and efficient logic programming algorithms can be developed that use this knowledge for answering non-trivial questions about the texts.

Our first task within the project was to formalize knowledge about a specific biological process, *cell division*. We represented the cell division domain in our modular language, which allowed us to obtain a concise and elaboration tolerant domain formalization. Note that at the time we only had our initial version of $\mathcal{ALM}$; work on this project allowed us to see possible improvements of the original design that led to the second version of $\mathcal{ALM}$.

Next, we created a QA system, $\mathcal{ALMAS}$, based on $\mathcal{ALM}$ and ASP, using the

[1] www.projecthalo.com/

methodology described in Sections 8.1 and 8.6. The new system was able to answer questions about cell division and other biological domains. Although successful, $\mathcal{ALMAS}$ could not be easily incorporated in the form of the Digital Aristotle current at that time, as this was based on a different logic formalism, the $\mathcal{F}$LORA-*2* declarative language. Hence, we had to create a second QA system compatible with the Digital Aristotle. We called it *the* $\mathcal{F}$LORA-2 *system*. The system is similar to $\mathcal{ALMAS}$, as it is derived from it by modifying the original translation from $\mathcal{ALM}$ into ASP according to the syntactic requirements of $\mathcal{F}$LORA-*2*.

We created a library of system descriptions of $\mathcal{ALM}$ formalizing the cell division domain at different levels of granularity. This library was available to both QA systems. We ignored the processing of the natural language in the input, as that task was assigned to other collaborators on Project Halo. Hence, each text-question pair was manually associated with a system description of $\mathcal{ALM}$. However, automating this task is not difficult because the library of system descriptions is known and the granularity level can be obtained from keywords in the text-question pair. For instance, the presence of the keyword "DNA" indicates the need to use the most detailed system description, *cell_cycle*(2) below. Note that the most detailed formalization produces correct answers for all input pairs, but using a coarser grained representation whenever possible is normally beneficial in terms of readability and efficiency. We also extracted by hand the history and query encodings for each example. We designed and implemented automatic translations of the original version of $\mathcal{ALM}$ into ASP and $\mathcal{F}$LORA-*2*, respectively. The two systems used script files to assemble the necessary logic program for each input pair and to run the appropriate solver to produce answers. The $\mathcal{ALMAS}$ system used the CLASP answer set solver. The $\mathcal{F}$LORA-*2* system used the XSB-based inference engine with the same name.

In what follows, we present the common part of the two QA systems in Section 9.1: the formalization of the cell division domain in $\mathcal{ALM}$. Here, we update the original formalization to the current syntax of $\mathcal{ALM}$. Next, we present the ASP-based QA system $\mathcal{ALMAS}$ and the $\mathcal{F}$LORA-*2* system in more detail in Section 9.2

and 9.3, respectively. We conclude with a comparison between the two in Section 9.4. Our work on the Digital Aristotle was previously presented in [Inclezan, 2010, Inclezan & Gelfond, 2011].

9.1 Formalizing Biological Knowledge in $\mathcal{ALM}$

The representation of the cell division domain is not trivial: it involves the creation of libraries describing *specialized biological knowledge*. It also requires elaborating a formalization of *naturally evolving processes*, i.e., series of phases and sub-phases that follow one another in a specific order, unless interrupted. We represented such processes as *sequences of actions intended by nature* and used a commonsense *theory of intentions* to reason about them. In what follows, we briefly introduce the domain of *cell division* and present our formalization of this process in $\mathcal{ALM}$. We exemplify how we encoded histories in logic form to reflect our intuition that all cells have the natural intention to divide.

9.1.1 The Cell Division Domain

Cell division (or *cell cycle*) refers to the phases a cell goes through from its "birth" to its division into two daughter cells. Cells consist of a number of parts, which in turn consist of other parts. We will focus here on *eukaryotic* cells (i.e., cells that contain a visibly evident nucleus). Eukaryotic cells contain organelles, cytoplasm, and a nucleus; the nucleus contains chromosomes, and the description can continue with more detailed parts. The eukaryotic cell cycle consists of an interphase and a mitotic phase. The interphase is mostly a growth phase, preparing the cell for division. The mitotic phase duplicates the nucleus and then divides the cell and the nuclei within it. This results in two daughter cells that are identical to the original one. The interphase and the mitotic phase are conventionally described as sequences of sub-phases. Depending on the level of detail of the description, these sub-phases may be simple events or sequences of other sub-phases. For example, the more detailed mitotic phase is described as a sequence of two sub-phases: mitosis and cytokinesis.

Mitosis, in turn, can be seen as a sequence of five sub-phases, etc. Certain chemicals, if introduced in the cell, can interfere with the ordered succession of events that is the cell cycle.

Answering questions about cell division requires: (1) knowing what the structure of the cell is at each stage of the cell cycle, and (2) understanding what motivates the succession of phases. This second point is especially interesting, as it calls for a method of representing and reasoning about *naturally evolving processes* in general.

9.1.2 Formalizing Cell Division in $\mathcal{ALM}$

We formalized the knowledge of cell division in $\mathcal{ALM}$ using techniques described in Chapter VII. We created two small library modules. One of them was a general commonsense module describing sequences and sequences of actions in particular. The other module was a specialized one formalizing the biological phenomenon of cell division. These library modules formed a general theory that we used in answering all of the test questions.

We begin with the presentation of our commonsense module describing sequences. This module, called *sequence*, captures our formalization of naturally evolving processes and of cell division in particular. The atom $component(S, N) = E$ appearing in the axioms of module *sequence* is supposed to be read as "the N^{th} component of sequence S is E".

module *sequence*

class declarations

sequences

attributes

$length : 1..maxint$

$component : 1..maxint \rightarrow$ **universe**

constraints

$\neg instance(S, sequences)$ **if** $component(S, N) = E,$
$N > length(S).$

$action\_sequences$: $sequences$

constraints

$\neg instance(S, action\_sequences)$ **if** $component(S, N) = E$,
$\neg instance(E, actions)$,
$\neg instance(E, action\_sequences)$.

Notice that this module can be used to represent a large class of dynamic systems, for instance traveling domains in which a journey is seen as a sequence of embark and disembark actions [Gelfond, 2006]. We assume that the module is stored in a library called *commonsense_lib*.

We now present our formalization of cell division. We start by modeling the eukaryotic cell. As seen above, the eukaryotic cell consists of various parts, which in turn consist of other parts. Together, they form a "*part of*" hierarchy, say H, which can be viewed as a tree. Nodes of this hierarchy will be labeled by elements of a new sort, $types\_of\_parts$, with an attribute is_part_of that maps the sort into itself. An attribute assignment $is\_part\_of(X) = Y$ indicates that Y is the father of X in H (i.e., there is a link from Y to X in H). Some elements of $types\_of\_parts$ are the *cell*, *nucleus*, etc., where $is\_part\_of(nucleus) = cell$, as nuclei are parts of cells. Additionally, we consider the experimental environment, which is usually called *sample*, to be of sort $type\_of\_parts$ and the root of H. To model the transitive closure of is_part_of, we introduce a boolean function $part\_of$ defined on pairs of $types\_of\_parts$; $part\_of(X, Y)$ will be true if X is a descendant of Y in H.

We assume that, at every stage of the cell division process, each cell has the same number of nuclei and similarly for the other inner components. More precisely, we assume that, at every stage and for each pair consisting of a child X and its parent Y in H, this pair is assigned a particular number indicating the number of elements of type X in one element of type Y. Therefore, the states of our domain will be described by an inertial fluent, $num : types\_of\_parts \times types\_of\_parts \rightarrow 0..maxint$, where $num(P_1, P_2) = N$ holds if the number of elements of type P_1 in one element of type P_2 is N. For instance, $num(nucleus, cell) = 2$ indicates that, at the current

stage of the cell cycle, each cell in the environment has two nuclei.

Next, we present the actions of our domain. To describe the cell cycle we will need two action classes: *duplicate* and *split*. *Duplicate*, which has an attribute *object* with values from *types_of_parts*, doubles the number of every part of this kind present in the environment. *Split* also has an attribute *object* ranging over *types_of_parts*. An action a of this type with $object(a) = c_1$, where c_2 is a son of c_1 in H, duplicates the number of elements of type c_1 in the environment and cuts in half the number of elements of type c_2 in one element of type c_1. For example, if the experimental environment consists of one cell with two nuclei, the occurrence of an instance a of action *split* with $object(a) = cell$ will increase the number of cells to two and decrease the number of nuclei per cells to one, thus resulting in an environment consisting of two cells with only one nucleus each. In addition to these two actions we will have an exogenous action, *prevent_duplication*, with an attribute *object* with the range *types_of_parts*. The occurrence of an instance action a of *prevent_duplication* with $object(a) = c$ will nullify the effects of duplication and splitting for the type of parts c. We will make use of this exogenous action in representing external events that interfere with the normal succession of sub-phases of cell division.

The description of the domain is given in a library module called *basic_cell_cycle*, which will eventually become part of a more general *cell_cycle_lib* library.

module *basic_cell_cycle*

class declarations

types_of_parts

attributes

is_part_of : *types_of_parts*

duplicate : **actions**

attributes

object : *types_of_parts*

split : *duplicate*

prevent_duplication : **actions**

attributes

$object : types\_of\_parts$

function declarations

static $part\_of : types\_of\_parts \times types\_of\_parts \rightarrow booleans$

inertial $num : types\_of\_parts \times types\_of\_parts \rightarrow 0..maxint$

inertial $prevented\_dupl(types\_of\_parts) \rightarrow booleans$

axioms

dynamic causal laws

$num(P_2, P_1) = N_2$ **if** $occurs(X)$, $instance(X, duplicate)$, $object(X) = P_2$, $is\_part\_of(P_2) = P_1$, $num(P_2, P_1) = N_1$, $N_1 * 2 = N_2$.

$num(P_2, P_1) = N_2$ **if** $occurs(X)$, $instance(X, split)$, $object(X) = P_1$, $is\_part\_of(P_2) = P_1$, $num(P_2, P_1) = N_1$, $N_2 * 2 = N_1$.

$prevented\_dupl(P)$ **if** $occurs(X)$, $instance(X, prevent\_duplication)$, $object(X) = P$.

state constraints

$part\_of(P_1, P_2)$ **if** $is\_part\_of(P_1) = P2$.

$part\_of(P_1, P_2)$ **if** $is\_part\_of(P_1) = P_3$, $part\_of(P_3, P_2)$.

$num(P, P) = 0.$

$num(P_3, P_1) = N$ **if** $is\_part\_of(P_3) = P_2,$
$part\_of(P_2, P_1),$
$num(P_2, P_1) = N_1,$
$num(P_3, P_2) = N_2,$
$N_1 * N_2 = N.$

executability conditions

$\neg occurs(X)$ **if** $instance(X, duplicate),$
$object(X) = P,$
$prevented\_dupl(P).$

Any model of cell cycle will consist of a theory importing the two library modules presented above and a structure corresponding to the level of detail of that model. Let us consider a first model, in which we view cell division as a sequence consisting of interphase and the mitotic phase. Interphase is considered to be an elementary action, whereas the mitotic phase is a sequence of two elementary actions: mitosis and cytokinesis. As mentioned before, we limit our domain to cells contained in an experimental environment, called *sample*. We remind the reader that an attribute assignment of the type $component(k) = y$ for an instance x below means that the "k^{th} component of sequence x is y".

system description *cell_cycle*(1)

theory *cell_cycle*

import *sequence* **from** *commonsense_lib*

import *basic_cell_cycle* **from** *cell_cycle_lib*

structure

sample **in** *types_of_parts*

cell **in** *types_of_parts*

is_part_of = *sample*

nucleus **in** *types_of_parts*

is_part_of = *cell*

cell_cycle **in** *action_sequences*

length = 2

component(1) = *interphase*

component(2) = *mitotic_phase*

mitotic_phase **in** *action_sequences*

length = 2

component(1) = *mitosis*

component(2) = *cytokinesis*

interphase **in actions**

mitosis **in** *duplicate*

object = *nucleus*

cytokinesis **in** *split*

object = *cell*

This formalization of cell division illustrates the ability of our language to model not only commonsensical dynamic systems, but also highly specialized, non-trivial domains. In addition, it shows the importance of creating and using libraries of knowledge in real-life applications. In the next step, we indicate how we elaborated this initial formalization of cell division to include more detailed information.

9.1.3 Formalizations at Different Granularity Levels in $\mathcal{ALM}$

The initial $\mathcal{ALM}$ model of cell division, *cell_cycle*(1), is quite general. It can be used to answer many of the textbook questions, but not all. Consider, for instance, the following question from [Campbell & Reece, 2001]:

12.15. *Text* : A researcher treats cells with a chemical that prevents DNA synthesis.
Question : This treatment traps the cells in which part of the cell cycle?

To answer it, the system will need to know more about the structure of the cell and that of the interphase and mitosis. Various refinements of our original model

of cell division will contain the same theory as the original formalization; only the structure of our original model will need to be expanded, in an elaboration tolerant way.

We created a finer grained model of the cell cycle, *cell_cycle*(2), which provides the additional knowledge needed to answer question *12.15*. More specifically, the following cell components were added to the structure: the chromosomes inside the nucleus, the chromatids that are part of the chromosomes, and the DNA inside the chromatids. As well, the interphase was now a sequence $\langle g_1, s, g_2 \rangle$ where g_1 and g_2 are elementary actions and s is a sequence of two elementary actions: DNA synthesis, and the creation of sister chromatids. Mitosis was a sequence of five actions: prophase, prometaphase, metaphase, anaphase, and telophase. The treatment of the cells with the chemical was represented by an exogenous action that prevents the duplication of the DNA.

system description *cell_cycle*(2)
  theory *cell_cycle*
    import *sequence* **from** *commonsense_lib*
    import *basic_cell_cycle* **from** *cell_cycle_lib*
  structure
    sample **in** *types_of_parts*
    cell **in** *types_of_parts*
      $is\_part\_of = sample$
    nucleus **in** *types_of_parts*
      $is\_part\_of = cell$
    chromosome **in** *types_of_parts*
      $is\_part\_of = nucleus$
    chromatid **in** *types_of_parts*
      $is\_part\_of = chromosome$
    dna **in** *types_of_parts*
      $is\_part\_of = chromatid$

$cell\_cycle$ **in** $action\_sequences$

$length = 2$

$component(1) = interphase$

$component(2) = mitotic\_phase$

$interphase$ **in** $action\_sequences$

$length = 3$

$component(1) = g1$

$component(2) = s$

$component(3) = g2$

s **in** $action\_sequences$

$length = 2$

$component(1) = dna\_synthesis$

$component(2) = sister\_chromatids$

$mitotic\_phase$ **in** $action\_sequences$

$length = 2$

$component(1) = mitosis$

$component(2) = cytokinesis$

$mitosis$ **in** $action\_sequences$

$length = 5$

$component(1) = prophase$

$component(2) = prometaphase$

$component(3) = metaphase$

$component(4) = anaphase$

$component(5) = telophase$

$g1, g2, prophase, prometaphase, metaphase$ **in actions**

$dna\_synthesis$ **in** $duplicate$

$object = dna$

sister_chromatids **in** *split*

object = *chromatid*

anaphase **in** *split*

object = *chromosome*

telophase **in** *split*

object = *nucleus*

cytokinesis **in** *split*

object = *cell*

treatment **in** *prevent_duplication*

object = *dna*

The system description *cell_cycle*(2) is detailed enough to answer question *12.15*. We created two additional refinements of our initial cell division model. We manually connected each of our test examples to the most appropriate of our final four system descriptions.

9.1.4 Cell Division History Encoding

We have shown how naturally evolving processes, and cell division in particular, can be specified in $\mathcal{ALM}$. We reasoned about naturally evolving processes by treating them as if they were intended by nature. For this purpose, we used a theory of intentions, which contains the logic programming encodings of the following main tenets: *"Normally intended actions are executed the moment such execution becomes possible"* and *"Unfulfilled intentions persist"* (see Appendix C). We used the theory to answer temporal projection questions about cell division in both of our QA systems. In Section 9.2, we will illustrate the use of intentions in reasoning about cell division on some sample questions.

Here, we describe how we incorporated in our histories the view that the purpose of a cell is to divide. System descriptions about the cell division domain, at any level of granularity, contain the *cell_cycle* event, which is viewed either as a simple action or as a sequence of actions. Thus, any history extracted from a text-question pair

that is about the cell cycle domain must contain the fact:

$$intend(cell\_cycle, 0).$$

This says that, ever since the beginning of their existence, cells intend to divide. In addition, some examples refer to exogenous actions that prevent the execution of some sub-phase of cell division. For instance, in the system description $cell\_cycle(2)$, we have the exogenous action $treatment$, which prevents the division of the DNA, and hence the execution of the $dna\_synthesis$ action. We describe the occurrence of such events in the normal way, via the predicate $hpd(a, i)$, as in $hpd(treatment, 0)$.

This concludes our description of the common components between the ASP and $\mathcal{F}$LORA-*2* systems. Next, we will describe some details regarding the ASP system, $\mathcal{ALMAS}$.

9.2 The $\mathcal{ALMAS}$ System

We built the ASP system for question answering, $\mathcal{ALMAS}$, as a proof of concept. We had an automated translation of $\mathcal{ALM}$ system descriptions into ASP theories and an ASP module for performing temporal projection. This module contained a set of rules connecting the predicates in the history to the predicates used in the ASP encoding of a system description. For each input text-question pair, the history and query were obtained as it was described in the previous section. The system used the CLASP solver to compute answer sets of the logic program assembled for each input.

In this section, we illustrate the reasoning performed by the ASP system for a couple of examples. Then, we show our results about the soundness and completeness of our reasoning algorithm, which we call "the ASP algorithm."

9.2.1 Reasoning about Cell Division Using Intentions

Let us illustrate how $\mathcal{ALMAS}$ answered questions about cell division. For simplicity, we limit ourselves to text-question pairs that require the $cell\_cycle(1)$ system description. We will emphasize the role of the theory of intentions in reasoning about naturally evolving process.

As a first example, we consider the following input pair:

S. *Text* : The experimental sample consists of one cell with one nucleus.

Question : How many cells will there be in the sample at the end of the cell cycle, and how many nuclei will each cell contain?

The history, $\Gamma_n(S)$, extracted from the text of S is written as:

$obs(eq(num(cell, sample), 1),\ 0).$

$obs(eq(num(nucleus, cell), 1),\ 0).$

$intend(cell\_cycle, 0).$

As mentioned before, the last statement of $\Gamma_n(S)$ captures our intuition that the purpose of a cell is to divide, expressed as an intention. The query, $\mathcal{Q}(S)$, manually extracted from the question of S looks as follows:

$$\begin{aligned} answer(X, \text{“cells per sample”}) \leftarrow\ & last\_step(I), \\ & holds(val(num(cell, sample), X), I). \\ answer(X, \text{“nuclei per cell”}) \leftarrow\ & last\_step(I), \\ & holds(val(num(nucleus, cell), X), I). \end{aligned}$$

We have enough information to automatize the extraction of query $\mathcal{Q}(S)$, as the expression “how many” in the question of S clearly indicates that the predicate *num* should be used; similarly, the expression “end of the cell cycle” points to the relation *last_step* from our theory of intentions. As described in Section 8.1, the solution to our reasoning problem is obtained from the answer set of a program, $\mathcal{P}(S)$, consisting of the ASP translation of the $cell\_cycle(1)$ system description, the domain history $\Gamma_n(S)$, the reasoning module Ω_{tpi} from Section 8.3 for temporal projection with intentions, and the query encoding $\mathcal{Q}(S)$. The answer set of $\mathcal{P}(S)$ will contain atoms:

$intend(cell\_cycle, 0)$ $occurs(interphase, 0)$

$intend(interphase, 0)$ $occurs(mitosis, 1)$

$intend(mitotic\_phase, 1)$ $occurs(cytokinesis, 2)$

$intend(mitosis, 1)$

$intend(cytokinesis, 2)$

showing that the phases of *cell_cycle* are intended and executed successfully in their natural order. The answer set of $\mathcal{P}(S)$ also contains the last step 3 and the facts:

$answer(2,$ "cells per sample")	$holds(val(num(cell, sample), 2), 3)$
$answer(1,$ "nuclei per cell")	$holds(val(num(nucleus, sample), 2), 3)$
$last\_step(3)$	$holds(val(num(nucleus, cell), 1), 3)$

Hence, the answer to our question is that, at the end of the cycle, the sample contains two cells with one nucleus each.

Next, we target a different kind of example, in which the naturally evolving process is interrupted. The following is an adaptation of an end-of-chapter text-question pair from the biology textbook:

12.9. $Text$: In some organisms mitosis occurs without cytokinesis occurring.
$Question$: How many cells will there be in the sample at the end of the cell cycle, and how many nuclei will each cell contain?

To encode the information given in the text, we simply expand the history $\Gamma_n(S)$ by

$$\neg hpd(cytokinesis, I)$$

for every step I, and name the new history $\Gamma_n(12.9)$. The query encoding, $\mathcal{Q}(12.9)$ will be identical to $\mathcal{Q}(S)$ above. We construct the program $\mathcal{P}(12.9)$ by putting together the ASP translation of the system description $cell\_cycle(1)$, the history $\Gamma_n(12.9)$, the reasoning module Ω_{tpi} for temporal projection with intentions, and the query encoding $\mathcal{Q}(12.9)$. The answer set of $\mathcal{P}(12.9)$ will contain:

$intend(cytokinesis, 2)$	$\neg occurs(cytokinesis, 2)$
$intend(cytokinesis, 3)$	$\neg occurs(cytokinesis, 3)$
$intend(cytokinesis, 4)$	$\neg occurs(cytokinesis, 4)$
...	

showing how the unfulfillable intention of executing action *cytokinesis* persists forever. Additionally, the answer set will include atoms:

$answer(1,$ "cells per sample")	$holds(val(num(cell, sample), 1), 2)$
$answer(2,$ "nuclei per cell")	$holds(val(num(nucleus, sample), 2), 2)$
$last\_step(2)$	$holds(val(num(nucleus, cell), 2), 2)$

which indicate that at the end of the cell cycle there will be one cell in the sample, with two nuclei. This is in fact the correct answer to question *12.9*.

We successfully applied this methodology of reasoning about naturally evolving processes using intentions to other questions about cell division.

9.2.2 The Soundness and Completeness of the ASP Algorithm

In the spirit of good software engineering practices, after testing $\mathcal{ALMAS}$ on several examples, we formally investigated the correctness and completeness of its algorithm. *We showed that the answers produced by our system to temporal projection questions are both sound and complete if the universe of the domain is finite and the initial state is completely specified.*

Remember that at the time our system was based on the initial version of $\mathcal{ALM}$. We proved this result for the original version of our language. Appendix E contains the formal statement of this result and its proof, which also appear in [Inclezan, 2010]. We believe that this result can be easily adapted to our current version of $\mathcal{ALM}$.

9.3 The $\mathcal{F}$LORA-*2* System

$\mathcal{ALMAS}$ was our proof of concept for using $\mathcal{ALM}$ in answering complex questions. Our next task was to develop a QA system that would be compatible with the Digital Aristotle. For that, we adapted our methodology of building the ASP system by replacing the translation from $\mathcal{ALM}$ into ASP with a translation into $\mathcal{F}$LORA-*2* that we created and by adapting the reasoning module Ω_{tpi} to the syntax of $\mathcal{F}$LORA-*2*. (We remind the reader that these transformations were done for the first version of $\mathcal{ALM}$; for details see Appendix D and B.) For instance, we replaced the original encoding of the Inertia Axioms in the translation of $\mathcal{ALM}$ system descriptions by

rules of the type:[2]

$$\begin{aligned} holds(val(F,V),I+1) \quad &\leftarrow \quad function(F,inertial),\\ &\qquad\quad range(F,V),\\ &\qquad\quad holds(val(F,V),I),\\ &\qquad\quad not\ defeated(val(F,V),I+1).\\ \neg holds(val(F,V),I+1) \quad &\leftarrow \quad function(F,inertial),\\ &\qquad\quad range(F,V),\\ &\qquad\quad \neg holds(val(F,V),I),\\ &\qquad\quad not\ defeated(n\_val(F,V),I+1). \end{aligned}$$

This allowed us to simplify the proofs of some mathematical results as we avoided loops through negation. As another example, we replaced the reality check

$$\begin{aligned} \leftarrow \quad &obs(eq(F,V),I),\\ &not\ holds(val(F,V),I),\\ &function(F),\\ &range(F,V). \end{aligned}$$

by the rule:

$$\begin{aligned} inconsistency(I) \quad \leftarrow \quad &function(F),\\ &range(F,V),\\ &obs(eq(F,V),I),\\ &not\ holds(val(F,V),I). \end{aligned}$$

as rules with empty heads are not allowed in $\mathcal{F}$LORA-*2*. Note that in the case of consistent histories with a completely specified initial situation, the well-founded model will not contain any $inconsistency(i)$ atom. Also note that the order of literals in the body of a rule matters in $\mathcal{F}$LORA-*2* while this is not important in ASP.

The resulting $\mathcal{F}$LORA-*2* system was able to answer questions about cell division successfully and was compatible with the Digital Aristotle. Finally, we investigated

[2]The symbol for classical negation in $\mathcal{F}$LORA-*2* is "neg". Here, we will use ¬ instead of "neg" for simplicity. Similarly, variables begin with a question mark in $\mathcal{F}$LORA-*2*, but here we will denote them by identifiers starting with capital letters.

the correctness and completeness of our second QA algorithm. *We showed that the answers produced by the* $\mathcal{F}$LORA-2 *system to temporal projection questions are sound if the universe of the domain is finite and the initial state is completely specified.* However, we could not show that the answers were also complete in the general case because $\mathcal{F}$LORA-*2* is based on the well-founded semantics and hence on the idea of a unique model for every program. We formulated a syntactic condition for system descriptions of $\mathcal{ALM}$ that guarantees that the answers produced by the $\mathcal{F}$LORA-*2* system are also complete. (The above results are formally stated and proved in Appendix F for the first version of $\mathcal{ALM}$.) The system descriptions for the cell division domain all belong to this category.

This mathematical result allowed us to compare our two QA systems. We had an additional comparison in terms of efficiency and applications of the two QA systems. In the next section, we present our conclusions derived from such comparisons.

9.4 Comparison between $\mathcal{ALMAS}$ and the $\mathcal{F}$LORA-*2* System

The two non-monotonic formalisms used by our two QA algorithms correspond to different intuitions. On the one hand, we have the answer set semantics based on the principle of *belief*, which implies the possibility of multiple valid models. On the other hand, we have the well-founded semantics and hence the idea of a *unique* set of conclusions for *every* program. This distinction has direct implications for the properties of the two algorithms. In particular, it determines the completeness of the ASP algorithm and the incompleteness, in the general case, of the $\mathcal{F}$LORA-*2* algorithm. Depending on the reasoning task to be performed, the incompleteness of the $\mathcal{F}$LORA-*2* algorithm can be seen as a limitation. This is especially the case when reasoning about the effects of non-deterministic actions, if we desire to know the consistent effects of these actions over all possible trajectories. Such effects are not detected by the $\mathcal{F}$LORA-*2* algorithm. The ASP algorithm has an advantage here, as each possible trajectory of the system is defined by an answer set.

The $\mathcal{F}$LORA-*2* system might, however, have an advantage over the ASP system

in terms of efficiency. The SLG algorithm of XSB, the basis of the $\mathcal{F}$LORA-*2* inference engine, has polynomial time complexity for function-free programs [Chen & Warren, 1996]. The logic programs described in this paper are indeed function-free. However, with the efficiency of ASP solvers constantly improving, we cannot make strong claims here. We ran tests on 30 examples using the inference engine $\mathcal{F}$LORA-*2* and the answer set solver CLASP. In most cases the two systems were equally efficient; only a minimal difference[3] was noted on three examples containing numerical constraints, where numbers ranged over a large set. New solvers integrating answer set reasoning with constraint solving techniques may annul this minimal advantage of $\mathcal{F}$LORA-*2*. This remains to be explored in the future.

In terms of applications of the two methods, we see an advantage for the ASP algorithm. Substantial work has been done to investigate the suitability of ASP for solving reasoning problems related to dynamic domains and used in answering questions from natural language, for example planning [Baral, 2003] or diagnosis [Balduccini & Gelfond, 2003b]. Our ASP system can be easily extended with a planning or diagnosis module in order to accomplish those tasks. As far as we know, work in these directions has not been done yet for $\mathcal{F}$LORA-*2*. Furthermore, although we were concerned in this paper only with consistent histories with a complete initial situation, the ASP approach can easily perform temporal projection for other types of histories. However, in the case of histories with an incomplete initial situation, the set of conclusions produced by $\mathcal{F}$LORA-*2* would be limited, due to the underlying formalism.

The use of $\mathcal{ALM}$ as a specification language in both methods has two advantages. First of all, it determines a smaller class of logic programs for which it is easy to compare ASP and $\mathcal{F}$LORA-*2*. Focusing on this small class can, however, shed some light on the relationship between ASP and $\mathcal{F}$LORA-*2* in general. The idea of using action

[3] 30 seconds versus 60 seconds on a machine with 1.73 GHz CPU and 1GB RAM running 32-bit Windows.

languages for comparing different formalisms was first explored by [Kartha, 1993]. Secondly, the $\mathcal{F}$LORA-*2* programs used by our QA system receive a specification independent from the computational technique that is used: in terms of a system description of $\mathcal{ALM}$ and the transition diagram it describes. This gives trustworthiness to the system, as it ensures that whatever is computed is indeed correct.

CHAPTER X
RELATED WORK

The idea of modularity in knowledge representation was first explored in the context of logic programming; from there it expanded to action languages during the last decade.

A first paper to survey the different modular endeavors in logic programming was [Bugliesi et al., 1994]. We will mention here some of the approaches for ASP that followed. In [Eiter et al., 1997], for instance, modules are viewed as *generalized quantifiers* that can be nested. Other authors created modular extensions of ASP by means of *templates* [Calimeri et al., 2004], [Ianni et al., 2004], *macros* [Baral et al., 2006], or *signature declarations* and *module definitions* [Balduccini, 2007b]. A more recent approach [Janhunen et al., 2009] makes use of the concept of an *lp-function* [Gabaldon & Gelfond, 1997].

In what concerns action languages, or more broadly languages dedicated to representing knowledge about dynamic domains, a first effort towards modularity led to the design of TAL-C [Gustafsson & Kvarnström, 2004]. TAL-C is an object-oriented extension of Temporal Action Logics [Doherty et al., 1998], [Doherty & Kvarnström, 2009]. It allows definitions of classes of objects that are somewhat similar to those in $\mathcal{ALM}$. TAL-C, however, seems to have more ambitious goals: the language is used to describe and reason about various dynamic scenarios, whereas in $\mathcal{ALM}$ the description of a scenario and that of reasoning tasks are not viewed as part of the language. Modular BAT [Gu & Soutchanski, 2007] is a modular language with similar goals to ours. However, it is based on Situation Calculus [McCarthy & Hayes, 1969, Shanahan, 1997, Reiter, 2001]. Language $\mathcal{M}$ [Gelfond, 2006] for creating knowledge modules is not exactly an action language as it extends CR-Prolog. However, it served as an inspiration for $\mathcal{ALM}$.

The language MAD [Lifschitz & Ren, 2006, Erdoğan & Lifschitz, 2006] is a modular expansion of action language $\mathcal{C}$ [Giunchiglia & Lifschitz, 1998]. Even though

MAD and $\mathcal{ALM}$ have a lot in common, they differ significantly in the underlying assumptions incorporated in their semantics. For example, the semantics of $\mathcal{ALM}$ incorporates the Inertia Axiom, which says that "*Things normally stay the same.*" Language $\mathcal{C}$ expanded by MAD is based on a different assumption – the Causality Principle – which says that "*Everything true in the world must be caused.*" Its underlying logical basis is causal logic [McCain & Turner, 1997, Giunchiglia et al., 2004a]. In $\mathcal{C}$ the inertia axiom for a literal l is expressed by a statement

caused l **if** l **after** l,

read as "there is a cause for l to hold after a transition if l holds both before and after the transition" [Gelfond & Lifschitz, 2012]. There is a close relationship between ASP and $\mathcal{C}$ but, in our judgment, the distance between ASP and $\mathcal{ALM}$ is much smaller than that between ASP and $\mathcal{C}$. There is also a substantial difference between modules of $\mathcal{ALM}$ and MAD, which leads to theories of MAD being constructed from a much larger number of modules than those of $\mathcal{ALM}$.

The differences in the underlying languages and in the way structure is incorporated into $\mathcal{ALM}$, MAD, TAL-C, and Modular BAT lead to very different knowledge representation styles. We believe that this is a good thing. Much more research and experience of use is needed to discover if one of these languages has some advantages over the others, or if different languages simply correspond to and enhance different habits of thought.

In the remainder of this chapter, we will focus on comparing our language with MAD in more detail, as MAD is the modular language with which $\mathcal{ALM}$ has the most in common.

10.1 Comparison between $\mathcal{ALM}$ and MAD

In order to compare the two modular languages, we translated system descriptions of $\mathcal{ALM}$ into action descriptions of MAD and vice-versa. Based on these experiments, we were able to notice the differences between the two languages as well as some of

their strengths and weaknesses. We will discuss the most interesting points of the two translations in the following two subsections, 10.1.1 and 10.1.2. We will summarize the conclusions of our comparison in Subsection 10.1.3.

We remind the reader that in MAD variables are identifiers starting with lower-case letter and constants are identifiers starting with upper-case letter, the opposite of $\mathcal{ALM}$.

10.1.1 Discussion about the Translation from $\mathcal{ALM}$ to MAD

To illustrate our translation of *system descriptions*[1] of $\mathcal{ALM}$ into action descriptions of MAD, let us start with an example. The precise description of this translation appears in Appendix G.

Example 24. [$\mathcal{ALM}$ to MAD Translation] Let us consider an extension of the system description *basic_travel* from Example 21 by some new action definitions: $move(john, paris)$ and $carry(A, C, P)$ as you will see below. For simplicity, we assume that the two modules in the theory (*move_between_points* and *carrying_things*), are replaced by a single closed module called *main*, equivalent to them. To make it easier to follow the translation, we show it little by little and add a marker ($\mathcal{ALM}$ or MAD) in front of statements in one language or the other.

The theory of *basic_travel* defines the following classes of primitive objects:

($\mathcal{ALM}$:)
$points$
$things$
$movers$: $things$
$carriables$: $things$

In MAD, we capture these declarations by adding the following sections to the action description $AD(basic\_travel)$:

[1] We translate system descriptions of $\mathcal{ALM}$ rather than just theories because the translation of a theory would result into a MAD collection of modules without definitions of objects and such a collection of modules does not have a meaning in MAD. By translating the structure as well, we ensure that object definitions are part of the resulting MAD action description.

(MAD :) **sorts**

$Points; Things; Movers; Carriables; Universe;$

inclusions

$Points << Universe;$

$Things << Universe;$

$Movers << Things;$

$Carriables << Things;$

The first part declares the four classes together with the pre-defined class *universe* of $\mathcal{ALM}$ and the second part describes the specialization relations between them. Other classes defined in the theory are the action classes *move* and *carry*, but we will show their translations later. Let us now look at the function declarations in our theory:

($\mathcal{ALM}$:) **function declarations**

inertial $loc\_in : things \rightarrow points$

inertial $holding : things \times things \rightarrow booleans$

defined $is\_held : things \rightarrow booleans$

To declare them in MAD, we create a module called $Basic\_travel$, following the name of the system description. We add to this module the following statements:

(MAD :) **fluents**

$Loc\_in(Things) : simple(Points);$

$Holding(Things, Things) : simple;$

$Is\_held(Things) : statically Determined;$

variables

$t, t_1, t_2 : Things;$

axioms

inertial $Loc\_in(t);$

inertial $Holding(t_1, t_2);$

Note that we need two axioms and variable definitions to say that our first two fluents are inertial. We also include in this module the translations of state constraints:

$(\mathcal{ALM}:)$ $loc\_in(C) = P \equiv loc\_in(T) = P$ **if** $holding(T, C)$.

$is\_held(X)$ **if** $holding(T, X)$.

For that purpose, we add to MAD module *Basic_travel* some new variables:

$(MAD:)$ $c, t_3, x, t_4 : Things$;

$p : Points$;

and the axioms:

$(MAD:)$ $Loc\_in(c) = p \equiv Loc\_in(t_3) = p$ **if** $Holding(t_3, c)$;

$Is\_held(x)$ **if** $Holding(t_4, x)$;

We can now go back and look at our declaration of action class *move*:

$(\mathcal{ALM}:)$ $move$: **actions**

attributes

$actor : movers$

$origin : points$

$dest : points$

We place its MAD translation into a separate module called *Basic_travel_move*. We do so to allow the first module to be used independently if our structure does not contain definitions of action instances. Note that in MAD actions are described as terms. To model attributes, we introduce variables with the same names as our attributes. This will facilitate referring to those attributes later. We also order attributes alphabetically as arguments of the action term to ease the translation of special case action classes of *move*:

$(MAD:)$ **module** *Basic_travel_move*;

actions

$Move(Movers, Points, Points)$;

variables

$actor : Movers$;

$origin, dest : Points$;

axioms

exogenous $Move(actor, dest, origin)$;

We also place in this module the dynamic causal laws and executability conditions that are about instances of class *move*:

$$(\mathcal{ALM}:) \qquad loc\_in(A) = D \quad \textbf{if} \quad instance(X, move),\ occurs(X),\ actor(X) = A,\ dest(X) = D.$$

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = A,\ origin(X) = O,\ loc\_in(A) \neq O.$$

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = A,\ dest(X) = D,\ loc\_in(A) = D.$$

$$\neg occurs(X) \quad \textbf{if} \quad instance(X, move),\ actor(X) = A,\ is\_held(A).$$

We first import the MAD module containing the declarations of functions, *Basic_travel*, into *Basic_travel_move*:

$$(MAD:) \qquad \textbf{import}\ Basic\_travel;$$

Then, we add the translation of the above laws to the **axioms** section of *Basic_travel_move*:

$$(MAD:) \qquad Move(actor, dest, origin)\ \textbf{causes}\ Loc\_in(a) = d\ \textbf{if}\ actor = a,\ dest = d;$$

$$\textbf{nonexecutable}\ Move(actor, dest, origin) \quad \textbf{if} \quad actor = a_1,\ origin = o,\ Loc\_in(a_1) \neq o;$$

$$\textbf{nonexecutable}\ Move(actor, dest, origin)\ \ \textbf{if}\ \ actor = a_2,\ dest = d_1,\ Loc\_in(a_2) = d_1;$$

$$\textbf{nonexecutable}\ Move(actor, dest, origin)\ \ \textbf{if}\ \ actor = a_3,\ Is\_held(a_3);$$

and declare the new variables used in these axioms

$$(MAD:)\qquad a, a_1, a_2, a_3 : Movers;$$
$$origin, d, o, d_1 : Points;$$

Our theory contains the declaration of another action class, *carry*, which is a special case of *move*:

$$(\mathcal{ALM}:)\qquad carry\ :\ move$$
$$\textbf{attributes}$$
$$carried\_thing : carriables$$

and has an additional executability condition:

$$(\mathcal{ALM}:)\qquad \neg occurs(X)\ \ \textbf{if}\ \ instance(X, carry),\ actor(X) = A,\ carried\_thing(X) = C,\ \neg holding(A, C).$$

Note that special case actions are declared in MAD by importing the module containing the original action and renaming that action as the special case action. To do that, we place the MAD translation of *carry* in a new module called *Basic_travel_carry* in which we import *Basic_travel_move* while renaming $Move(actor, dest, origin)$ as $Carry(actor, carried\_thing, dest, origin)$. Here is where the alphabetical ordering of variables corresponding to attributes becomes handy: it helps match attributes of the original class to attributes of the special case class.

$(MAD:)$ **module** $Basic\_travel\_carry$;

actions

$Carry(Movers, Carriables, Points, Points)$;

variables

$actor, a : Movers$;

$dest, origin : Points$;

$carried\_thing, c : Carriables$;

import $Basic\_motion\_move$;

$Move(actor, dest, origin)$ **is**

$Carry(actor, carried\_thing, dest, origin)$;

axioms

exogenous $Carry(actor, carried\_thing, dest, origin)$;

nonexecutable $Carry(actor, carried\_thing, dest, origin)$

if $actor = a$,

$carried\_thing = c$,

$\neg Holding(a, c)$;

Finally, we consider the structure of our $\mathcal{ALM}$ system description:

$(\mathcal{ALM}:)$ **structure**

$john, bob$ **in** $movers$

$london, paris$ **in** $points$

$suitcase$ **in** $carriables$

$move(A, P_1, P_2)$ **in** $move$

$actor = A$

$origin = P_1$

$dest = P_2$

$move(john, paris)$ **in** $move$

$actor = john$

$dest = paris$

$carry(A, C, P)$ **in** $carry$

$actor = A$

$carried\_thing = C$

$dest = P$

To translate it into MAD, we add a new module called S to the action description $AD(Basic\_travel)$. This module imports module $Basic\_travel$, which contains the declarations of functions and state constraints. It also imports the modules declaring the action classes whose instances are defined in the structure; instances of action classes are defined by using renaming clauses.

$(MAD:)$ **module** S;

objects

$John, Bob : Movers$;

$London, Paris : Points$;

$Suitcase : Carriables$;

actions

$Move(Movers, Points, Points)$;

$Move(Movers, Points)$;

$Carry(Movers, Carriables, Points)$;

variables

$actor, a, a_1 : Movers$;

$origin, dest, p_1, p_2, p : Points$;

$carried\_thing, c : Carriables$;

import $Basic\_motion$;

import $Basic\_motion\_move$;

$Move(actor, dest, origin)$ **is**

case $actor = a$ & $origin = p_1$ & $dest = p_2$:

$Move(a, p_1, p_2)$;

import $Basic\_motion\_move$;

$Move(actor, dest, origin)$ **is**

case $actor = John$ & $dest = Paris$:

$Move(John, Paris)$;

import $Basic\_motion\_carry$;

$Carry(actor, carried\_thing, dest, origin)$ **is**

case $actor = a_1$ & $carried\_thing = c$ & $dest = p$:

$Carry(a_1, c, p)$;

This ends our translation example. Note that the number of variables could be reduced and the definition of action instances in module S could be simplified. However, the version shown here would be the one produced by our translation algorithm described in Appendix G.

In the remainder of this section we will discuss certain features or concepts of $\mathcal{ALM}$ that either were not straightforwardly translatable into MAD or produced a translation that was substantially different from the original.

- *Recursive definitions*

  The translation of state constraints is not straightforward if the collection of state constraints defines a *cyclic* fluent dependency graph [Gelfond & Lifschitz, 2012]. For instance, the $\mathcal{ALM}$ state constraint:

  $$p \textbf{ if } p.$$

  is not equivalent to the same axiom in MAD. The $\mathcal{ALM}$ axiom can be eliminated without modifying the meaning of the system description; it says that "in every state in which p holds, p must hold." Eliminating the same axiom from a MAD action description would not produce an equivalent action description; in MAD, the axiom says that "p holds by default."

- *Action class attributes with arity* > 0

The name of an action in MAD is a term with one parameter for each of its possible attributes. If a class c of $\mathcal{ALM}$ has an attribute

$$attr\_name : c \times c_1 \times \cdots \times c_n \rightarrow c_{n+1}$$

where $n > 0$, then the number of possible attributes of an action of this class depends on the interpretations of classes $c_1, \ldots, c_n$. In the current translation of $\mathcal{ALM}$ into MAD, we wanted our translation of a theory to be independent from its interpretation, which seems to be unavoidable here.

Another possible translation of $\mathcal{ALM}$ system descriptions into MAD can be created starting from the interpretation (i.e., the structure) of the system description. In that case, we would replace variables that stand for action instances in axioms (e.g., X in $occurs(X)$, $instance(X, move)$) by their possible instantiations. However, such a translation would produce a unique module of MAD. This would be inconvenient, as it would not allow us to compare the modular structure of $\mathcal{ALM}$ with that of MAD.

- *Functions defined on, or ranging over action classes*

  Fluents of MAD must be defined on, and range over either primitive sorts or the built-in class *action*. A possible translation of functions defined on or ranging over *action* classes would be to transform each such function

  $$f : c_1 \times \cdots \times c_n \rightarrow c_{n+1}$$

  of $\mathcal{ALM}$ into a boolean function

  $$f(c'_1, \ldots, c'_n, c'_{n+1})$$

  of MAD, where $\forall 1 \leq i \leq n+1$, $c'_i =_{def} action$ if $c \leq_{\mathcal{C}} actions$ and $c'_i =_{def} c_i$ otherwise, and to add the following axiom:

  $$\textbf{default } \neg f(X_1, \ldots, X_n, X_{n+1});$$

However, this implies that f will be defined on all actions, whereas it may happen that the $\mathcal{ALM}$ counterpart of f is defined only on some special case class of actions. Hence, there will be more fluents in MAD than in $\mathcal{ALM}$ and states of the MAD transition diagram will be larger than for $\mathcal{ALM}$.

- *Action classes with attributes and attribute constraints*

  An action class is declared as a term in MAD. In our translation, that term contains one argument for each attribute of the $\mathcal{ALM}$ action class. An action term of MAD stands for the execution of that action, which would correspond to our *occurs* relation. There is no immediate correspondent for atoms of the type $instance(X, c)$ where c is an action class. Such atoms may appear, for example, in attribute constraints, as in:

  $$\neg instance(X, move) \textbf{ if } origin(X) = dest(X)$$

  Representing such axioms is not straightforward. It may require declaring a rigid fluent defined on the built-in sort *action* for each of the attributes. These fluents would be false by default, except for the particular action. Attribute constraints would then be expressed using these rigid fluents. Notice that such an approach would duplicate the encoding of attributes: both as arguments (needed to distinguish between the names of different instances) and as rigid fluents.

- *Variables in axioms*

  In $\mathcal{ALM}$, we do not define the sorts of variables, this information is evident from the atoms in which they appear. In MAD, variables need to be defined. This produces larger modules and makes axioms not self-contained in the sense that the reader has to look at the **variables** section to completely understand the meaning of an axiom. However, this may sometimes produce shorter axioms than in $\mathcal{ALM}$. For example, if we have the function $f : c_1 \rightarrow booleans$, c_2 is a

special case of c_1, and we want to say that f is true for all instances of c_2, in $\mathcal{ALM}$ we would add an *instance* atom to the body of the rule:

$$f(X) \textbf{ if } instance(X, c_2)$$

whereas in MAD the axiom would be just:

$$f(X)$$

and variable X would be defined to range over sort c_2.

- *The specialization construct on action classes*

  In $\mathcal{ALM}$, an action class and its specialization can be part of the same module. This is not the case in MAD. As a consequence, our MAD translation contains more modules that are smaller in size than the $\mathcal{ALM}$ counterpart. On the other hand, note that $\mathcal{ALM}$ modules are not required to be large; they can be as small as a user desires.

10.1.2 Discussion about the Translation from MAD to $\mathcal{ALM}$

Let us start by illustrating the translation via some simple examples.

Example 25. [Simple Action Description] We start by considering an action description of MAD from [Erdoğan, 2008], which consists of a single module that does not contain *import* statements:

```
(MAD :)   sorts
              Domain;
              Range;
          module ASSIGN;
              actions
                  Assign(Domain, Range);
              fluents
                  Value(Domain) : simple(Range);
```

variables

$x : Domain$;

$y : Range$;

axioms

inertial $Value(x)$;

exogenous $Assign(x, y)$;

$Assign(x, y)$ **causes** $Value(x) = y$;

As this action description does not contain definitions of objects, it does not correspond to a *system description* of $\mathcal{ALM}$, but rather to a single *module*. In our translation of the action *assign*, we give names to its arguments, which will thus become attributes in $\mathcal{ALM}$. As a consequence, the dynamic causal law will also look different:

($\mathcal{ALM}$:) **module** *assign*

class declarations

domain

range

assign : **actions**

attributes

$attr_1 : domain$

$attr_2 : range$

function declarations

inertial $value : domain \rightarrow range$

axioms

dynamic causal laws

$value(X) = Y$ **if** $instance(A, assign)$,
$occurs(A)$,
$attr_1(A) = X$,
$attr_2(A) = Y$.

Example 26. [Action Description with Import Statements] Let us now consider a more complex action description with two new sorts, *Thing* and *Place*. It consists of module *ASSIGN* from Example 25 and a new module, *MOVE* from [Erdoğan, 2008], which imports the previous module and renames its sorts, fluent, and action:

$(MAD:)$ **module** $MOVE$;

actions

$Move(Thing, Place)$;

fluents

$Location(Thing) : simple(Place)$;

variables

$x : Thing$;

$y : Place$;

import $ASSIGN$

$Domain$ **is** $Thing$;

$Range$ **is** $Place$;

$Value(x)$ **is** $Location(x)$;

$Assign(x, p)$ **is** $Move(x, p)$;

axioms

nonexecutable $Move(x, p)$ **if** $Location(x) = p$;

To translate this action description in $\mathcal{ALM}$, we could either expand our $\mathcal{ALM}$ module from the previous example or we could create a new module. We illustrate here the second option. Notice that we translate renaming clauses for sorts and actions via the specialization construct of $\mathcal{ALM}$ and renaming clauses for fluents via state constraints using the shorthand $\equiv$. Additionally, if an action is renamed and the sorts of its attributes are also renamed in the same import statement, we add some attribute constraints to the $\mathcal{ALM}$ translation of the new action class.

$(\mathcal{ALM}:)$ **module** *move*

depends on *assign*

class declarations

$thing : domain$

$place : range$

$move : assign$

constraints

$\neg instance(X, move)$ **if** $attr_1(X) = A$, $\neg instance(A, thing)$.

$\neg instance(X, move)$ **if** $attr_2(X) = A$, $\neg instance(A, place)$.

function declarations

inertial $location : thing \rightarrow place$

axioms

state constraints

$location(X_1) = X_2 \equiv value(X_1) = X_2$

executability constraints

$\neg occurs(A)$ **if** $instance(A, move)$, $location(attr_1(A)) = attr_2(A)$.

Example 27. [Action Description with Object Definitions] We consider an action description consisting of modules $ASSIGN$ and $MOVE$ from Examples 25 and 26 respectively, together with a module defining objects as follows:

$(MAD :)$ **module** $ROBOT$

objects

$Robot : Thing$;

$P_1, P_2 : Place$;

actions

$Walk(Place)$;

variables

$p : Place$;

import $MOVE$;

$Move(Robot, p)$ **is** $Walk(p)$;

This action description would be translated into a system description of $\mathcal{ALM}$ consisting of the translation of the two modules, $ASSIGN$ and $MOVE$, and a structure that is the translation of module $ROBOT$ into $\mathcal{ALM}$:

($\mathcal{ALM}$:) **structure**

$robot$ **in** $thing$;

p_1, p_2 **in** $place$;

$walk(P)$ **in** $move$

$attr_1 = robot$

$attr_2 = P$

Example 28. [Problematic Translation] Applying the translation illustrated by the above examples to the action description in Example 3 from Section 2.6 is not a simple task. Particular problems are caused by the renaming statement:

(MAD :) **import** $MOVE$;

$Location(Monkey) = l$ **is**

case $l = BoxTop : OnBox$;

case $l = Floor : \neg OnBox$;

We cannot apply our usual methodology for translating renaming clauses for fluents because the renaming clause above makes reference to particular object constants; in $\mathcal{ALM}$, object constants that are not pre-defined are not part of the signature of a module. Hence, we cannot place the translation of this statement in a module of $\mathcal{ALM}$. An option would be to state this clause using more general terms acceptable for the syntax of $\mathcal{ALM}$ (e.g., by referring to arbitrary instances of some class(es)). However, the resulting translation would not be straightforward nor easy to automate. Another option would be to place the $\mathcal{ALM}$ counterpart of this statement in the structure, but structures do not contain axioms in the current version of our language. A final option would be to delegate it to the description of the history, which lies outside the scope of $\mathcal{ALM}$.

Similar problems occur in relation to axioms:

$(MAD:)$ **nonexecutable** $ClimbOn$ **if** $Location(Monkey) \neq Location(Box)$;
nonexecutable $ClimbOff \wedge Walk(p)$;

For example, action *ClimbOn* would be defined as an action instance in the structure of the resulting system description. However, it is not clear where the translation of the second axiom above would be placed. In order to place it in the theory it would have to be stated in more general terms, without using object constants. It is not obvious how such a translation could be automated for the general case.

In what follows, we will discuss those features of MAD that are not easily translatable into $\mathcal{ALM}$:

- *Renaming clauses for sorts*

  We translate such clauses using the specialization construct of $\mathcal{ALM}$, whereas the meaning of a renaming clause is that the two sorts are synonyms.

- *Renaming clauses for actions*

  If the sorts of the original action are also renamed, we need to add attribute constraints to the new action in $\mathcal{ALM}$, according to our translation.

- *Renaming clauses for (partially) ground fluents*

  It is not clear how such statements would be translated in $\mathcal{ALM}$. (This problem was illustrated in Example 28.)

- *Axioms containing user-defined object constants*

  There is no obvious translation for such axioms nor an obvious place for their translations in an $\mathcal{ALM}$ system description. Most probably, axioms of this type would have to be transformed into more general statements in $\mathcal{ALM}$. (This problem was illustrated in Example 28.)

- *Certain types of axioms of MAD*

Axioms of the type:

$$formula \textbf{ may cause } formula \text{ [} \textbf{if } formula \text{]}$$

or

$$\textbf{default } formula \text{ [} \textbf{if } formula \text{] [} \textbf{after } formula \text{]}$$

are not directly translatable into $\mathcal{ALM}$. They may be expressible (see Example 2), but more rules and extra fluents may be needed in their $\mathcal{ALM}$ counterparts.

10.1.3 Summary of Conclusions

Based on the translation from $\mathcal{ALM}$ to MAD and vice-versa, we can summarize the main differences between the two languages:

- *Underlying assumptions: Inertia Axiom in $\mathcal{ALM}$ vs. Causality Principle in MAD*

  As a consequence, recursive definitions in $\mathcal{ALM}$ are not obviously translatable into MAD.

- *Basic concepts: class in $\mathcal{ALM}$ vs. sort and action in MAD*

  This causes problems in the translation of functions of $\mathcal{ALM}$ that are defined on, or range over action classes, the translation of attribute constraints for action classes, etc.

- *Declaration of special case knowledge: specialization construct in $\mathcal{ALM}$ vs. import statements and renaming clauses in MAD*

  This leads to a different structuring of knowledge: generally, fewer but larger modules in $\mathcal{ALM}$ and more but smaller modules in MAD.

- *Distinction between classes and their instances in $\mathcal{ALM}$*

  As a result, the translation of *renaming clauses*—a basic concept of MAD— is not straightforward when user-defined object constants appear in them. The same is true of axioms of MAD that contain user-defined object constants.

CHAPTER XI
CONCLUSIONS AND FUTURE WORK

11.1 Conclusions

In this dissertation, we presented an action language, $\mathcal{ALM}$, characterized by a modular structure and the ability to separate the definition of *classes of objects* of the domain from the definition of *instances* of these classes. This, together with the means for defining objects of the domain as special cases of previously defined ones facilitates the stepwise development, testing, and readability of a knowledge base, as well as the development of knowledge representation libraries.

We illustrated the methodology of representing knowledge in $\mathcal{ALM}$ by extending our initial library motion by knowledge about means of transportation and about more complex spatial layouts. We exemplified how our libraries can be used to create descriptions of several benchmark domains. We presented a methodology of $\mathcal{ALM}$'s use for solving computational tasks.

We reported on an application of $\mathcal{ALM}$ for the development of question answering systems in the framework of our collaboration on Project Halo. This case study provided evidence of the adequacy of $\mathcal{ALM}$ for knowledge representation. The use of $\mathcal{ALM}$ clearly facilitated the design of provenly correct systems, and was essential in comparing such systems, which are based on different nonmonotonic formalisms. In particular, we proved the equivalence of our question answering systems for a specific class of system descriptions. As part of this project, we designed and implemented automatic translations of the original version of $\mathcal{ALM}$ into ASP and $\mathcal{F}$LORA-*2*, respectively.

We compared $\mathcal{ALM}$ with a modular action language with similar goals to ours, MAD, by creating a translation from $\mathcal{ALM}$ into MAD. This translation was instrumental in noticing differences between the two languages.

In the course of designing $\mathcal{ALM}$, we had an initial version that was an extension of action language $\mathcal{AL}$. Based on experiments that followed, we realized some drawbacks

of our original design. This led to a second version of $\mathcal{ALM}$, the one presented here, which substantially improves the syntax and semantics of the language and the methodology of its use. Finally, we adapted the automated translator from $\mathcal{ALM}$ into ASP to match the changes in our language.

11.2 Future Work

We see different directions in which the work presented here can be extended:

- We plan to precisely formulate the translation from MAD into $\mathcal{ALM}$ and to formally investigate the relation between the two languages.

- It would be important to develop various general $\mathcal{ALM}$ libraries about commonsense domains. This would imply further expanding our library about motion and adding similar libraries about other recurrent domains. Such practice would also be beneficial in testing (and possibly improving) the current design of $\mathcal{ALM}$.

- We believe that the new theory of intentions described in [Blount & Gelfond, 2012] should become a module of $\mathcal{ALM}$. For that purpose, our language should be expanded to allow fluents to range over atoms of the language. This is needed in order to model the notion of a goal of an activity. A goal is a ground fluent, as in:

  $$goal(m, loc\_in(john, paris))$$

  This says that the goal of activity m is to have John located in Paris.

- We would like to extend $\mathcal{ALM}$ with the capability of representing knowledge about *hybrid* domains, i.e., domains which allow both discrete and continuous change. We intend to achieve this goal by combing $\mathcal{ALM}$ with action language $\mathcal{H}$ [Chintabathina et al., 2005, Chintabathina, 2010].

- In order to facilitate the use of $\mathcal{ALM}$ in applications, a development environment should be created. Such an environment would facilitate the creation and storage of libraries, the testing and debugging of modules, etc.

- Finally, we would like to create a more efficient implementation of $\mathcal{ALM}$. We plan to achieve it in the future by substituting our automated translation from $\mathcal{ALM}$ into ASP with a translation into extensions of ASP with non-Herbrand functions, ASP{f} [Balduccini, 2012], and with sorted signatures, $\mathcal{SPARC}$ [Balai et al., 2012].

BIBLIOGRAPHY

[Amarel, 1968] Amarel, S. (1968). On Representations of Problems of Reasoning about Actions. In D. Michie (Ed.), *Machine Intelligence*, volume 3 (pp. 131–171). Edinburgh University Press.

[Apt & Pellegrini, 1994] Apt, K. R. & Pellegrini, A. (1994). On the Occur-check Free Prolog Programs. In *ACM Transactions on Programming Languages and Systems* (pp. 1–19).: Springer-Verlag.

[Balai et al., 2012] Balai, Gelfond, M., & Zhang, Y. (2012). SPARC: Sorted ASP with Consistency Restoring Rules. (unpublished).

[Balduccini, 2004] Balduccini, M. (2004). USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL'04*, Lecture Notes in Artificial Intelligence (LNCS).

[Balduccini, 2005] Balduccini, M. (2005). *Answer Set Based Design of Highly Autonomous, Rational Agents*. PhD thesis, Texas Tech University.

[Balduccini, 2007a] Balduccini, M. (2007a). CR-MODELS: An Inference Engine for CR-Prolog. In *Proceedings of LPNMR-07* (pp. 18–30).

[Balduccini, 2007b] Balduccini, M. (2007b). Modules and Signature Declarations for A-Prolog: Progress Report. In *Proceedings of Workshop of Software Engineering for Answer Set Programming (SEA)*.

[Balduccini, 2012] Balduccini, M. (2012). A "Conservative" Approach to Extending Answer Set Programming with Non-Herbrand Functions. In *Correct Reasoning* (pp. 24–39).

[Balduccini et al., 2008] Balduccini, M., Baral, C., & Lierler, Y. (2008). Knowledge Representation and Question Answering. In F. van Harmelen, V. Lifschitz, & B. Porter (Eds.), *Handbook of Knowledge Representation* (pp. 779–820). Elsevier.

[Balduccini & Gelfond, 2003a] Balduccini, M. & Gelfond, M. (2003a). Diagnostic Reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3, 425–461.

[Balduccini & Gelfond, 2003b] Balduccini, M. & Gelfond, M. (2003b). Logic Programs with Consistency-Restoring Rules. In P. Doherty, J. McCarthy, & M.-A. Williams (Eds.), *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series (pp. 9–18).

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press.

[Baral et al., 2006] Baral, C., Dzifcak, J., & Takahashi, H. (2006). Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming. In *Proceedings of International Conference on Logic Programming (ICLP)*. 376–390.

[Baral et al., 2005] Baral, C., Gelfond, G., Gelfond, M., & Scherl, R. B. (2005). Textual Inference by Combining Multiple Logic Programming Paradigms. In *Proceedings of the AAAI'05 Workshop on Inference for Textual Question Answering* (pp. 1–5). Pittsburgh, PA, USA: AAAI Press.

[Baral & Gelfond, 1994] Baral, C. & Gelfond, M. (1994). Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19(20), 73–148.

[Baral & Gelfond, 2000] Baral, C. & Gelfond, M. (2000). *Reasoning Agents in Dynamic Domains*, (pp. 257–279). Kluwer Academic Publishers: Norwell, MA.

[Baral & Gelfond, 2005] Baral, C. & Gelfond, M. (2005). Reasoning about Intended Actions. In *AAAI-05: Proceedings of the 20th National Conference on Artificial Intelligence* (pp. 689–694).: AAAI Press.

[Baral et al., 1997] Baral, C., Gelfond, M., & Provetti, A. (1997). Representing Actions: Laws, Observations and Hypotheses. *Journal of Logic Programming*, 31(1–3), 201–243.

[Baral et al., 2004] Baral, C., Gelfond, M., & Scherl, R. (2004). Using Answer Set Programming to Answer Complex Queries. In *Workshop on Pragmatics of Question Answering at HLT-NAAC2004*.

[Baral & Lobo, 1997] Baral, C. & Lobo, J. (1997). Defeasible Specifications in Action Theories. In *Proceedings of IJCAI-97* (pp. 1441–1446).

[Blount & Gelfond, 2012] Blount, J. & Gelfond, M. (2012). Reasoning about the Intentions of Agents. In A. Artikis, R. Craven, N. Kesim Çiçekli, B. Sadighi, & K. Stathis (Eds.), *Logic Programs, Norms and Action*, volume 7360 of *Lecture Notes in Computer Science* (pp. 147–171). Springer Berlin / Heidelberg.

[Bugliesi et al., 1994] Bugliesi, M., Lamma, E., & Mello, P. (1994). Modularity in Logic Programming. *Journal of Logic Programming*, 19/20, 443–502.

[Calimeri et al., 2004] Calimeri, F., Ianni, G., Ielpa, G., Pietramala, A., & Santoro, M. C. (2004). A System with Template Answer Set Programs. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)* (pp. 693–697).

[Campbell & Reece, 2001] Campbell, N. A. & Reece, J. B. (2001). *Biology*. Benjamin Cummings, 6th edition.

[Chen & Warren, 1996] Chen, W. & Warren, D. S. (1996). Tabled Evaluation with Delaying for General Logic Pprograms. *Journal of the ACM*, 43, 20–74.

[Chintabathina, 2010] Chintabathina, S. (2010). *Towards Answer Set Programming Based Architectures For Intelligent Agents*. PhD thesis, Texas Tech University, Lubbock, TX, USA.

[Chintabathina et al., 2005] Chintabathina, S., Gelfond, M., & Watson, R. (2005). Modeling Hybrid Domains Using Process Description Language. In *Proceedings of ASP '05 Answer Set Programming: Advances in Theory and Implementation* (pp. 303–317).

[Dell'Armi et al., 2003] Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., & Pfeifer, G. (2003). Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 03)* (pp. 847–852).: Morgan Kaufmann.

[Dembinski & Maluszynski, 1985] Dembinski, P. & Maluszynski, J. (1985). And-Parallelism with Intelligent Backtracking for Annotated Logic Programs. In *SLP* (pp. 29–38).

[Doherty et al., 1998] Doherty, P., Gustafsson, J., Karlsson, L., & Kvarnström, J. (1998). TAL: Temporal action logics language specification and tutorial. *Linköping Electronic Articles in Computer and Information Science*, 3.

[Doherty & Kvarnström, 2009] Doherty, P. & Kvarnström, J. (2009). Temporal Action Logics. In V. Lifschitz, F. van Harmelen, & B. Porter (Eds.), *Handbook of Temporal Reasoning in Artificial Intelligence*, volume 3 of *Foundations of Artificial Intelligence* (pp. 709–757). Elsevier.

[Eiter et al., 1997] Eiter, T., Gottlob, G., & Veith, H. (1997). Modular Logic Programming and Generalized Quantifiers. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)* (pp. 290–309).: Springer.

[Erdoğan & Lifschitz, 2006] Erdoğan, S. & Lifschitz, V. (2006). Actions as Special Cases. In *Principles of Knowledge Representation and Reasoning: Proceedings of the International Conference* (pp. 377–387).

[Erdoğan, 2008] Erdoğan, S. T. (2008). *A Library of General-Purpose Action Descriptions*. PhD thesis, University of Texas at Austin, Austin, TX, USA.

[Fikes & Nilsson, 1971] Fikes, R. & Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3–4), 189–208.

[Finger, 1986] Finger, J. (1986). *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford University.

[Gabaldon & Gelfond, 1997] Gabaldon, A. & Gelfond, M. (1997). From Functional Specifications to Logic Programs. In *Proceedings of the International Logic Programming Symposium (ILPS'97)*.

[Gebser et al., 2007] Gebser, M., Kaufmann, B., Neumann, A., & Schaub, T. (2007). Conflict-Driven Answer Set Solving. In *IJCAI-07: Proceedings of the 20th International Joint Conference on Artifical Intelligence* (pp. 386–392). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Gelfond, 2006] Gelfond, M. (2006). Going Places – Notes on a Modular Development of Knowledge about Travel. AAAI 2006 Spring Symposium Series (pp. 56–66).

[Gelfond & Inclezan, 2009] Gelfond, M. & Inclezan, D. (2009). Yet Another Modular Action Language. In *Proceedings of SEA-09* (pp. 64–78).: University of Bath Opus: Online Publications Store.

[Gelfond & Kahl, 2012] Gelfond, M. & Kahl, Y. (2012). *Knowledge Representation, Reasoning, and the Design of Intelligent Agents*. Available at URL: http://www.cs.ttu.edu/~mgelfond/ai/book.pdf.

[Gelfond & Lifschitz, 1988] Gelfond, M. & Lifschitz, V. (1988). The Stable Model Semantics for Logic Programming. In *Proceedings of ICLP-88* (pp. 1070–1080).

[Gelfond & Lifschitz, 1991] Gelfond, M. & Lifschitz, V. (1991). Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4), 365–386.

[Gelfond & Lifschitz, 1998] Gelfond, M. & Lifschitz, V. (1998). Action Languages. *Electronic Transactions on AI*, 3(16), 193–210.

[Gelfond & Lifschitz, 2012] Gelfond, M. & Lifschitz, V. (2012). The Common Core of Action Languages B and C. Available at URL: http://www.cs.utexas.edu/users/vl/papers/BC.pdf. (Submitted to NMR-12).

[Giunchiglia et al., 2004a] Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., & Turner, H. (2004a). Nonmonotonic Causal Theories. *Artificial Intelligence*, 153(1–2), 105–140.

[Giunchiglia et al., 2004b] Giunchiglia, E., Lierler, Y., & Maratea, M. (2004b). SAT-Based Answer Set Programming. AAAI 2004 Nineteenth National Conference on Artificial Intelligence.

[Giunchiglia & Lifschitz, 1998] Giunchiglia, E. & Lifschitz, V. (1998). An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)* (pp. 623–630).: AAAI Press.

[Gu & Soutchanski, 2007] Gu, Y. & Soutchanski, M. (2007). Modular Basic Action Theories. In *Proceedings of the 7th IJCAI International Workshop on Nonmontonic Reasoning, Action and Change (NRAC-07)* (pp. 73–78).

[Gunning et al., 2010] Gunning, D., Chaudhri, V. K., Clark, P., Barker, K., Chaw, S.-Y., Greaves, M., Grosof, B., Leung, A., McDonald, D., Mishra, S., Pacheco, J., Porter, B., Spaulding, A., Tecuci, D., & Tien, J. (2010). Project Halo–Progress Toward Digital Aristotle. *AI Magazine*, 31(3), 33–58.

[Gustafsson & Kvarnström, 2004] Gustafsson, J. & Kvarnström, J. (2004). Elaboration Tolerance through Object-Orientation. *Artificial Intelligence*, 153, 239–285.

[Hanks & McDermott, 1987] Hanks, S. & McDermott, D. (1987). Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence*, 33(3), 379–412.

[Ianni et al., 2004] Ianni, G., Ielpa, G., Pietramala, A., Santoro, M. C., & Calimeri, F. (2004). Enhancing Answer Set Programming with Templates. In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)* (pp. 233–239).

[Inclezan, 2010] Inclezan, D. (2010). Computing Trajectories of Dynamic Systems Using ASP and Flora-2. Available at URL: http://www.cs.uky.edu/~marek/ nonmonat30.dir/inclezan.pdf.

[Inclezan & Gelfond, 2011] Inclezan, D. & Gelfond, M. (2011). Representing Biological Processes in Modular Action Language ALM. In *Proceedings of the 2011 AAAI Spring Symposium on Formalizing Commonsense* (pp. 49–55).: AAAI Press.

[Janhunen et al., 2009] Janhunen, T., Oikarinen, E., Tompits, H., & Woltran, S. (2009). Modularity Aspects of Disjunctive Stable Models. *Artificial Intelligence Research*, 35, 813–857.

[Kartha, 1993] Kartha, N. (1993). Soundness and Completeness Theorems for Three Formalizations of Action. In *Proceedings of IJCAI-93* (pp. 724–729).: Morgan Kaufmann.

[Kifer, 2005] Kifer, M. (2005). Nonmonotonic Reasoning in FLORA-2. In C. Baral, G. Greco, N. Leone, & G. Terracina (Eds.), *Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *Lecture Notes in Computer Science* (pp. 1–12). Springer Berlin / Heidelberg.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006). The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7, 499–562.

[Lifschitz, 1999] Lifschitz, V. (1999). *Action Languages, Answer Sets, and Planning*, (pp. 357–373). The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin.

[Lifschitz & Ren, 2006] Lifschitz, V. & Ren, W. (2006). A Modular Action Description Language. Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI) (pp. 853–859).

[Lin, 1995] Lin, F. (1995). Embracing Causality in Specifying the Indirect Effects of Actions. In C. Mellish (Ed.), *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-95)* (pp. 1985–1993).: Morgan Kaufmann.

[McCain & Turner, 1995] McCain, N. & Turner, H. (1995). A Causal Theory of Ramifications and Qualifications. *Artificial Intelligence*, 32, 57–95.

[McCain & Turner, 1997] McCain, N. & Turner, H. (1997). Causal Theories of Action and Change. In *Proceedings of AAAI-97* (pp. 460–465).

[McCarthy, 1977] McCarthy, J. (1977). Epistemological Problems of Artificial Intelligence. Proceedings of the IJCAI (pp. 1038–1044).

[McCarthy & Hayes, 1969] McCarthy, J. & Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 4* (pp. 463–502). Edinburgh University Press.

[Niemelä & Simons, 1997] Niemelä, I. & Simons, P. (1997). Smodels - An Implementation of the Stable Model and Well-founded Semantics for Normal Logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-97)*, volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)* (pp. 420–429).

[Pednault, 1987] Pednault, E. (1987). Formulating Multi-Agent Dynamic-World Problems in the Classical Planning Framework. In M. Georgeff & A. Lansky (Eds.), *Reasoning about actions and plans* (pp. 47–82). Morgan Kaufmann.

[Pednault, 1988] Pednault, E. (1988). Synthesizing Plans that Contain Actions with Context-Dependent Effects. *Computational Intelligence*, 4(4), 356–372.

[Reiter, 1978] Reiter, R. (1978). On Closed World Data Bases. In H. Gallaire & J. Minker (Eds.), *Logic and Data Bases* (pp. 119–140). Plenum Press.

[Reiter, 2001] Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press.

[Shanahan, 1997] Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press.

[Stroetman, 1993] Stroetman, K. (1993). A Completeness Result for SLDNF Resolution. *Journal of Logic Programming*, 15, 337–357.

[Todorova & Gelfond, 2012] Todorova, Y. & Gelfond, M. (2012). Toward Question Answering in Travel Domains. In *Correct Reasoning* (pp. 311–326).

[Turner, 1997] Turner, H. (1997). Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *Journal of Logic Programming*, 31(1–3), 245–298.

[Van Gelder et al., 1991] Van Gelder, A., Ross, K. A., & Schlipf, J. S. (1991). The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38, 619–649.

APPENDIX A
MATHEMATICAL PROPERTIES OF $\mathcal{AL}$

The semantics of $\mathcal{AL}$ with defined fluents is non-monotonic and hence, in principle, the addition of a new definition could substantially change the diagram of $\mathcal{D}$. The theorem we present in this subsection shows that this is generally not the case. In order to formulate it precisely, we need the following definition from [Erdoğan, 2008].

Definition 12 (Residue)**.** Let $\mathcal{D}$ and $\mathcal{D}'$ be system descriptions of $\mathcal{AL}$ such that the signature of $\mathcal{D}'$ is part of the signature of $\mathcal{D}$. $\mathcal{D}'$ is a *residue* of $\mathcal{D}$ if restricting the states and actions of $\mathcal{T}(\mathcal{D})$ to the signature of $\mathcal{D}'$ establishes an isomorphism between $\mathcal{T}(\mathcal{D}')$ and $\mathcal{T}(\mathcal{D})$.

We also use the following definitions:

Definition 13 (Fluent dependency graph)**.** The *fluent dependency graph* of a system description $\mathcal{D}$ of $\mathcal{AL}_d$ is the directed graph such that

- its vertices are arbitrary fluent literals,
- it has an edge from
  - l to l' if l is formed by an inertial fluent and $\mathcal{D}$ contains a static law with head l and the body containing l',
  - from f to l' if f is a defined fluent and $\mathcal{D}$ contains a static law with head f and the body containing l' and not containing f,
  - from $\neg f$ to f for every defined fluent f.

Definition 14 (Weakly Acyclic System Description)**.** A system description $\mathcal{D}$ of $\mathcal{AL}$ is called *weakly acyclic* if its fluent dependency graph contains no paths from defined fluents to their negations.

Theorem 1. For every *weakly acyclic* system description $\mathcal{D}$ of $\mathcal{AL}$ such that the signature of $\mathcal{D}$ contains defined fluents, there is a system description $\mathcal{D}'$ such that the signature of $\mathcal{D}'$ does not contain defined fluents and $\mathcal{D}'$ is a residue of $\mathcal{D}$.

The following lemma will be useful in proving our theorem. It states that any defined fluent from $\mathcal{D}$ can be eliminated in such a way that the resulting system description is a residue of the original one.

Lemma 1. If $\mathcal{D}$ is a *weakly acyclic* system description of $\mathcal{AL}$ and f is a defined fluent from the signature of $\mathcal{D}$, then there is a mapping from $\mathcal{D}$ and f into a system description $\mathcal{E}(\mathcal{D}, f)$ not containing f, such that $\mathcal{E}(\mathcal{D}, f)$ is a residue of $\mathcal{D}$.

Proof of Lemma 1. We prove this lemma by construction.
(a) We use the following notation: If Δ is the set of rules

$$\begin{array}{lll} \{ \quad l & \textbf{if} & \beta_1 \\ & \ldots & \\ \quad l & \textbf{if} & \beta_n \quad \} \end{array}$$

then by $unsat(\Delta)$ we denote the collection of all minimal sets of literals such that each of these minimal sets would make the body of every rule in Δ unsatisfiable:

$$unsat(\Delta) = \{ \ \{\bar{l_1}, \ldots, \bar{l_n}\} \ : l_1 \in \beta_1, \ \ldots, \ l_n \in \beta_n\}$$

where by $\bar{l}$ we denote the complement of l.
(b) $\mathcal{E}$ can be seen as a function that takes as inputs a weakly acyclic system description of $\mathcal{AL}$ and a defined fluent f from the signature of $\mathcal{D}$. The output of $\mathcal{E}$ is another weakly acyclic system description of $\mathcal{AL}$.
(c) Let $\mathcal{D}$ be a weakly acyclic system description of $\mathcal{AL}$. $\mathcal{E}(\mathcal{D}, f)$ is constructed from $\mathcal{D}$ as follows:

(i) Eliminate every statement of the type "f **if** f, Γ". (Note that Γ cannot contain $\neg f$ because $\mathcal{D}$ is weakly acyclic.)

Let's assume that the remaining definition of f is the set of rules $\delta(f)$:

$$\begin{array}{rcl} \delta(f) & =_{def} & \{ \quad f \quad \textbf{if} \quad \beta_1 \\ & & \qquad \ldots \\ & & \qquad f \quad \textbf{if} \quad \beta_k \quad \} \end{array} \tag{0.1}$$

Note that the sets of literals $\beta_1, \ldots, \beta_k$ do not contain f; they do not contain $\neg f$ either because $\mathcal{D}$ is weakly acyclic.

(ii) Replace every statement "α **if** f, Γ" by the set of statements:

$$\begin{array}{l} \alpha \quad \textbf{if} \quad \beta_1, \Gamma \\ \ldots \\ \alpha \quad \textbf{if} \quad \beta_k, \Gamma \end{array} \tag{0.2}$$

(iii) Replace every statement "α **if** $\neg f, \Gamma$" by the set of statements:

$$\begin{array}{l} \alpha \quad \textbf{if} \quad x_1, \Gamma \\ \ldots \\ \alpha \quad \textbf{if} \quad x_m, \Gamma \end{array} \tag{0.3}$$

where $x_1, \ldots, x_m$ are sets of literals such that $\{x_1, \ldots, x_m\} = unsat(\delta(f))$.

(The intuition here is that $\neg f$ is true if no statement in the definition of f is satisfied, i.e. the body of every statement from the definition of f contains at least one literal that is false.)

(iv) Eliminate the definition $\delta(f)$ of f from the system description.

(v) Eliminate f from the signature of the system description.

(d) It can be shown via standard techniques that, given a state σ_0 and a collection of actions e, the answer sets of $\Pi(\mathcal{D}', \sigma_0, e)$ are identical to the answer sets of $\Pi(\mathcal{D}, \sigma_0, e)$ modulo the set $\{holds(f, i), \neg holds(f, i)\}$ for every step i. (The construction of program Π was described in Chapter II.)

(e) By (d) and Definitions 10, 11, and 12, $\mathcal{E}(\mathcal{D}, f)$ is a residue of $\mathcal{D}$. □

We can now prove Theorem 1.

Proof of Theorem 1. We prove this theorem by construction.

(a). Let $\mathcal{F}_d$ be the set of all defined fluents from the signature of $\mathcal{D}$.

(b). For every fluent $f \in \mathcal{F}_d$, let $\mathcal{E}(\mathcal{D}, f)$ be defined as in Lemma 1.

(c). Let $\pi = (f_1, \ldots, f_n)$ be an arbitrary permutation of $\mathcal{F}_d$.

(d). We denote by $\mathcal{E}(\mathcal{D}, \pi)$ the successive elimination of defined fluents, in the order in which they appear in π:

$$\mathcal{E}(\mathcal{D}, \pi) =_{def} \mathcal{E}(\ldots \mathcal{E}(\mathcal{E}(\mathcal{D}, f_1), f_2) \ldots, f_n)$$

(e). By Lemma 1 applied successively, $\mathcal{E}(\mathcal{D}, \pi)$ is a residue of $\mathcal{D}$. □

Let us state the following corollary of Theorem 1:

Corollary 1. Let $\mathcal{D}$ be a weakly acyclic system description of $\mathcal{AL}$ with signature Σ, $f \notin \Sigma$ be a new symbol for a defined fluent, and $\mathcal{D}'$ be the result of adding to $\mathcal{D}$ the definition of f. Then $\mathcal{D}$ is a residue of $\mathcal{D}'$.

APPENDIX B

$\mathcal{F}$LORA-*2* ENCODING OF $\mathcal{AL}$ SYSTEM DESCRIPTIONS

We present the translation of $\mathcal{AL}$ system descriptions into $\mathcal{F}$LORA-*2* logic programs. For every $\mathcal{AL}$ system description, $\mathcal{D}$, we construct a program $\Pi_F(\mathcal{D})$ consisting of $\mathcal{F}$LORA-*2* encodings of statements from $\mathcal{D}$.

Let $\mathcal{D}$ be a system description of $\mathcal{AL}$. The signature of $\Pi_F(\mathcal{D})$ will consist of the signature of $\Pi(\mathcal{D})$ together with the relations:

- $defeated(fluent, step)$ ($defeated(f, i)$ says that the non-static fluent f is defeated at time step i) and
- $defeated(n(fluent), step)$ ($defeated(n(f), i)$ says that $\neg f$ is defeated at time step i, where f is a non-static fluent).

By $\bar{l}$ we will denote $n(f)$ if $l = f$ or f if $l = \neg f$. The logic program $\Pi_F(\mathcal{D})$ is constructed as follows:[1]

1. For every static causal law "l **if** p" from $\mathcal{D}$, $\Pi_F(\mathcal{D})$ contains:

$$h(l, I) \leftarrow h(p, I). \tag{0.4}$$

   If l is a non-static literal, then $\Pi_F(\mathcal{D})$ also contains:

$$defeated(\bar{l}, I) \leftarrow h(p, I). \tag{0.5}$$

2. For every dynamic causal law "a **causes** l **if** p" from $\mathcal{D}$, $\Pi_F(\mathcal{D})$ contains:

$$\begin{aligned} h(l, I+1) \leftarrow\ & h(p, I), \\ & occurs(a, I). \end{aligned} \tag{0.6}$$

[1] As mentioned previously, the symbol for classical negation in $\mathcal{F}$LORA-*2* is "neg", but we will use $\neg$ instead, for simplicity.

$$defeated(\bar{l}, I+1) \leftarrow h(p, I), \; occurs(a, I). \tag{0.7}$$

(Note that based on the definition of inertial, defined and static fluents, the literal l appearing in the dynamic causal law above can only be an inertial literal).

3. For every executability condition "**impossible** $a_1, \ldots, a_k$ **if** p" from $\mathcal{D}$, and for every i such that $0 \leq i \leq k$, $\Pi_F(\mathcal{D})$ contains:

$$\begin{aligned} \neg occurs(a_i, I) \leftarrow \; & not \; \neg occurs(a_1, I), \\ & \ldots \\ & not \; \neg occurs(a_{i-1}, I), \\ & not \; \neg occurs(a_{i+1}, I), \\ & \ldots \\ & not \; \neg occurs(a_k, I), \\ & h(p, I). \end{aligned} \tag{0.8}$$

4. $\Pi_F(\mathcal{D})$ contains the Inertia Axioms:

$$\begin{aligned} holds(F, I+1) \leftarrow \; & fluent(\textbf{inertial}, F), \\ & holds(F, I), \\ & not \; defeated(F, I+1). \\ \neg holds(F, I+1) \leftarrow \; & fluent(\textbf{inertial}, F), \\ & \neg holds(F, I), \\ & not \; defeated(n(F), I+1). \end{aligned} \tag{0.9}$$

5. $\Pi_F(\mathcal{D})$ contains the CWA for defined fluents:

$$\neg holds(F, I) \leftarrow fluent(\textbf{defined}, F), \; not \; defeated(n(F), I). \tag{0.10}$$

APPENDIX C
THEORY OF INTENTIONS

We present the complete encoding of the theory of intentions [Baral & Gelfond, 2005, Gelfond, 2006] used in this dissertation. This program, called ToI, expects as an input the description of sequences of actions in our domain. Sequences are defined via two predicates:

• $component(S, K, V)$ - which says that the K^{th} component of sequence S is V. Note that V may be either an action or a sequence of actions.

• $length(S, N)$ - which says that the length of sequence S is N.

For instance, the sequence $s = \langle a_1, a_2 \rangle$ is defined as:

$$component(s, 1, a_1). \quad component(s, 2, a_2). \quad length(s, 2).$$

In the program ToI encoding the theory of intentions:

• I, I_1, and I_2 are variables for time steps,

• A is a variable for actions,

• S is a variable for sequences of actions,

• V, V_1, and V_2 are variables for actions or sequences of action (i.e., *elements*), and

• N, K_1, and K_2 are variables for natural numbers.

The program ToI contains the *non-procrastination axiom*, i.e., *"Normally intended actions are executed the moment such execution becomes possible"*:

$$\begin{aligned} occurs(A, I) \quad \leftarrow \quad & intend(A, I), \\ & not\ \neg occurs(A, I). \end{aligned}$$

and the *persistence axiom* (*"Unfulfilled intentions persist"*):

$$\begin{aligned} intend(A, I_1) \quad \leftarrow \quad & intend(A, I), \\ & \neg occurs(A, I), \\ & not\ \neg intend(A, I_1), \\ & I_1 = I + 1. \end{aligned}$$

In addition, ToI includes axioms for intended sequences of actions: *"The first element of a sequence is intended whenever the sequence is intended"*:

$$\begin{aligned} intend(V, I) \quad \leftarrow \quad & intend(S, I), \\ & component(S, 1, V), \\ & elements(V), \\ & sequences(S). \end{aligned}$$

and *"Whenever an element of a sequence is completed, the following element in the sequence is intended"*:

$$\begin{aligned} intend(V_2, I_2) \quad \leftarrow \quad & intend(S, I_1), \\ & component(S, K_1, V_2), \\ & component(S, K, V_1), \\ & K_1 = K + 1, \\ & ends(V_1, I_2), \\ & elements(V_1), \\ & elements(V_2) \\ & sequences(S). \end{aligned}$$

It also contains rules describing the completion of an action or sequence, i.e., the relation $ends(V, I)$:

$$\begin{aligned} ends(S, I) \quad \leftarrow \quad & length(S, N), \\ & component(S, N, V), \\ & ends(V, I). \\ ends(A, I_1) \quad \leftarrow \quad & occurs(A, I), \\ & I_1 = I + 1. \end{aligned}$$

Finally, ToI includes the encoding of the relation *last_step* (i.e., the time step follow-

ing the last occurrence of an action in a trajectory):

$$\begin{aligned} last\_step(I) &\leftarrow not\ \neg last\_step(I). \\ \neg last\_step(I) &\leftarrow last\_step(I1), \\ &\quad\ I1 < I. \\ \neg last\_step(I) &\leftarrow occurs(A, I1), \\ &\quad\ I1 >= I. \end{aligned}$$

APPENDIX D
THE FIRST VERSION OF $\mathcal{ALM}$

This section contains an brief presentation of the initial version of $\mathcal{ALM}$. We assume that the reader is familiar with action description language $\mathcal{AL}$ or is willing to consult Chapter II for details.

Syntax

One of the central concepts of $\mathcal{ALM}$ is that of a *module* — a formal description of a collection of closely connected classes of *elementary objects*, *fluents*, and *actions* of some dynamic system. We start our description of the syntax and semantics of $\mathcal{ALM}$ with the introduction of a simple class of modules called *basic*.

The notion is illustrated by the following example.

Example 29. [Module]

module *move_between_points*
sort declarations
$things, points$: **sort**
$movers$: $things$
fluent declarations
$loc\_in(things, points)$: **inertial fluent**
axioms
$\neg loc\_in(T, A_2)$ **if** $loc\_in(T, A_1), A_1 \neq A_2$.
end of *loc_in*
action declarations
$move$: **action**
attributes
$actor : movers$
$origin, dest : points$

axioms

$move$ **causes** $loc\_in(O, A)$ **if** $actor = O$,
$dest = A$.

impossible $move$ **if** $actor = O$,
$origin = A$,
$\neg loc\_in(O, A)$.

end of $move$

A *basic module* of $\mathcal{ALM}$ consists of three parts: *sort declarations*, *fluent declarations*, and *action declarations*.

The *sort declarations* part of the module contains a hierarchy of names of classes of elementary objects (often referred to as *object classes*). A sort declaration is a statement of the type

$$s_1 \ : \ s_2$$

where s_1 is the name of an object class and s_2 is either the name of an object class or the keyword **sort**. In the latter case, the statement simply declares a new sort s_1. In the former, s_1 is declared as a special case of s_2. Our module *move_between_points* has object classes *points*, *things*, and *movers*. They are organized in a simple hierarchy with the root **sort**, classes *things* and *areas* defined as its children, and *movers* defined as a subclass of *things*.

The *fluent declarations* part of a basic $\mathcal{ALM}$ module consists of descriptions of fluent classes, their parameters, types, and axioms describing the relationship between fluents. Syntactically, a fluent declaration has the form:

$f(s_1, \ldots, s_k)$: *type* **fluent**
axioms
[*state constraint* .]$^+$
end of f

Here f is the name of a fluent class and $s_1, \ldots, s_k$ are object classes for the parameters of f. Similarly to $\mathcal{AL}$, fluents of $\mathcal{ALM}$ are of one of three types: *inertial*, *defined*,

and *static*.

The axiom part of the declaration consists of *state constraints* (also known as static causal laws) specifying the relationships between fluents. To define the syntax of a state constraint we need the notion of an *open fluent atom* of a module of $\mathcal{ALM}$ — an expression of the form $f(X_1, \ldots, X_n)$ where f is a name of a fluent class declared in the module and $X_1, \ldots, X_n$ are variables. An *open fluent literal* is an open atom or its negation. A state constraint of a module M has the form

$$l \textbf{ if } p$$

where: (1) l is an open fluent literal of M and (2) p is a collection of open fluent literals of M and atoms of the form $s(X)$ where s is a sort of M. The informal reading of a state constraint of $\mathcal{ALM}$ is the same as that of state constraints of $\mathcal{AL}$ with variables. If the declaration of a fluent class f contains no state constraints, then the keywords **axioms** and **end of** f should be omitted.

Our module declares a fluent class, *loc_in*, with parameters of the sorts *movers* and *points*, respectively. The fluent is declared as inertial. The state constraint of the fluent declaration says that a thing cannot be located at two points at the same time.

The *action declarations* part of an $\mathcal{ALM}$ basic module contains names of action classes, their attributes, and axioms describing direct effects of actions and their impossibility conditions. Syntactically, an action declaration has the form:

a : *action*
attributes
$[attr : sort]^+$
axioms
$[law .]^+$
end of a

where a is the name of an action class, *attr* is an identifier used to name an attribute of the actions of this class, and *law* is a dynamic causal law or an executability

condition.

A dynamic causal law of a module M describing the direct effect of actions from class a has the form:

$$a \textbf{ causes } l \textbf{ if } p$$

where: (1) a is the name of an action class; (2) l is an open fluent literal; and (3) p is a collection of open fluent literals of M, atoms of the form $s(X)$ where s is a sort of M, and *open attribute atoms* — expressions of the form $attr = Y$, where $attr$ is the name of an attribute of the action and Y is a variable ranging over elements of the corresponding sort. By *open literals* we mean open fluent literals or open attribute atoms. The law says that if an action from the class a with attributes satisfying attribute conditions from p were to be executed in a state satisfying the other conditions of p then l would be true in the resulting state.

The impossibility condition of $\mathcal{ALM}$ is slightly more complex. In its simplest form it looks as follows:

$$\textbf{impossible } a \textbf{ if } p$$

where a is the name of an action class and p is a collection of open literals of M, atoms of the form $s(X)$ where s is a sort of M, and open attribute atoms. Intuitively, the law states that actions from the class a with attributes satisfying attribute conditions from p can not occur in states satisfying the other conditions from p.

For instance, our module *move_between_points* has a simple form of executability condition. The module declares one action class – *move*. It states that actions of this type are allowed to have three attributes: *actor*, with values from the sort *movers*, and *origin* and *dest*, with values from the sort *points*. The first axiom of *move* says that the occurrence of any action that is an instance of *move* and has actor O and destination A causes O to be located in A. The second axiom says that no instance of action class *move* can occur in a state in which its actor is not located at the origin.

A more general executability condition of $\mathcal{ALM}$ has the form:

$$\textbf{impossible } a_1, \ldots, a_n \textbf{ if } p$$

where $a_1, \ldots, a_n$ are names of action classes and p is a collection of open fluent literals, atoms of the form $s(X)$ where s is a sort of M, and expressions of the form $attr(a_i) = Y$, with $1 \leq i \leq n$. The slight complication of the syntax allows to differentiate between attributes of different actions. A typical example may be a condition:

impossible a_1, a_2 **if** $actor(a_1) = Y, actor(a_2) = Y$

which says that actions from classes a_1 and a_2 can not be performed simultaneously by the same actor. An additional complication appears when we want to state the impossibility condition for actions of the same type. This is done by adding a parameter to the name of the action class. For instance, the following (unnecessary) impossibility condition can be added to the action declaration of *move*:

impossible $move(1), move(2)$ **if** $actor(move(1)) = Y, actor(move(2)) = Y$

The condition states that two instances of *move* cannot be simultaneously performed by the same actor. Of course a module cannot contain two declarations of the same class. This completes the description of the syntax of a basic module of $\mathcal{ALM}$.

It is important to notice that, like in the first-order logic, symbols declared in a module of $\mathcal{ALM}$ do not have a fixed interpretation. If the module contains declarations of all its symbols it can, semantically, be viewed as a *mapping of possible interpretations of the symbols of the domain into the transition diagram containing all possible trajectories of the corresponding dynamic system.* This *separation of uninterpreted modules from their interpretations is an essential part of* $\mathcal{ALM}$*, which is, in a large part, responsible for the reusability of its theories.*

To describe a particular interpretation of the symbols of a module M we use the notion of $\mathcal{ALM}$'s *structure*. The following example illustrates such an interpretation for the module *move_between_points*.

Example 30. [Structure of *move_between_points*]

structure

sorts

$john, bob \in movers$

$london, paris \in points$

actions

instance $move(O, A_1, A_2)$ **where** $A_1 \neq A_2$: $move$

$actor := O$

$origin := A_1$

$dest := A_2$

Syntactically, a structure of an $\mathcal{ALM}$ module M starts with the *sort definition*, which has the form:

sorts

$[constants \in s]^+$

where *constants* is a non-empty list of identifiers referred to as *objects* of sort s.

The definition of sorts in a structure of $\mathcal{ALM}$ is followed by the *definition of actions*. Syntactically, it has the form:

actions

$[instance\ description]^+$

where an *instance description* is defined as follows:

instance α **where** $cond$: a

$attr_1 := y_1$

...

$attr_k := y_k$

Here α is a term used to name an action from an action class a that is defined by this instance description, $attr_1, \ldots, attr_k$ are attributes of this action, y's are properly sorted objects of the structure and *cond* is a set of fluent literals formed by static fluents. Note that an instance description of $\mathcal{ALM}$ may also contain variables. In this case it will be referred to as an *action schema*, and viewed as a shorthand for the set of instance descriptions obtained from the schema by replacing its variables $V_1, \ldots, V_k$ by their possibles values $c_1, \ldots, c_k$.

The action definition part of the structure is followed by the *definition of values of static fluents*. Syntactically, it has the form:

statics

[*state constraint* .]$^+$

These state constraints are similar but not identical to the state constraints in the fluent declarations of a module. Both the head and body of a state constraint from the structure must be *static* fluent literals. As well, the static fluent literals appearing in a state constraint from the structure do not have to be *open*; their parameters can be properly sorted *objects* defined in the structure. If the list of state constraints is empty, then the keyword **statics** should be omitted, as in Example 30. The semantics of $\mathcal{ALM}$ require that the static fluents declared in a system description be completely defined. The values of static fluents are obtained by taking the definitions of statics from the structure, closing them under the axioms for statics from the declaration part, and adding the closed world assumption to whatever was derived. The last step ensures the complete definition of statics.

Particular domains are described using *system descriptions*, which consist of a *theory* and a *structure*. A theory is a collection of modules that can be transformed into a single basic module. Hence, in order to give the semantics of system descriptions of our language, we can simply give the semantics of basic modules.

Semantics

Semantically, a basic module M can be viewed as a function which maps its structure, S, into the action description $M(S)$ of $\mathcal{AL}$. Now we will define this mapping.

1. The signature Σ of $M(S)$ contains sorts defined in the structure. We assume that the sorts have no other elements except those specified in their definitions.

2. Actions of $M(S)$ are action instances defined in S.

3. Fluents of $M(S)$ are

- Expressions of the form $f(x_1, \ldots, x_k)$ where $s_1, \ldots, s_k$ are the sorts of the parameters of f in the declaration of f from module M, and for every i, x_i is an object constant of the sort s_i.
- Expressions of the form $attr(\alpha, y)$ where α is an action defined in S and the definition of α contains a statement $attr := y$.

We concluded the definition of the signature of action description $M(S)$ of $\mathcal{AL}$. To conclude the definition of $M(S)$, we need to define its axioms. This is done as follows:

1. State constraints of M are viewed as schemas for state constraints of $M(S)$. Similarly for the state constraints containing variables from the static definitions of S. State constraints of S without variables are state constraints of $M(S)$.

2. For every dynamic causal law a **causes** l **if** p and every action α from the class a defined in S, replace a by α and replace every attribute atom $attr = Y$ in p by a fluent atom $attr(\alpha, Y)$.

3. For every impossibility condition **impossible** $a_1, \ldots, a_n$ **if** p and every collection of different actions $\alpha_1, \ldots, \alpha_n$ where α_i is defined in S and belongs to class a_i for every $1 \leq i \leq n$, replace all occurrences of a_i by α_i and replace every attribute atom $attr(\alpha_i) = Y$ in p by a fluent atom $attr(\alpha_i, Y)$.

APPENDIX E
MATHEMATICAL RESULTS FOR THE $\mathcal{ALMAS}$ SYSTEM

The $\mathcal{ALMAS}$ system was built based on the original version of $\mathcal{ALM}$ described in Appendix 11.2. Here, we will show the theorem that summarizes our theoretical results regarding the ASP algorithm. But before we do that, we introduce some preliminary definitions. We borrowed from [Baral & Gelfond, 2005, Balduccini & Gelfond, 2003b] and adapted the formal definitions for the following terms: *history* of a system description, *model* of a history, *module for temporal projection*, and an *answer set defining a sequence*.

Definition 15 (Observed History)**.** By the *history* Γ_n of a system description $\mathcal{D}$ up to time step n we mean a collection of observations, i.e., facts of the form:

1. $observed(f, true/false, i)$ - fluent f was observed to be true/false at step i, where $0 \leq i \leq n$.

2. $happened(a, i)$ or $\neg happened(a, i)$ - action a was observed to have happened or to not have happened, respectively, at step i, where $0 \leq i < n$.

3. $intend(\alpha, i)$ - the execution of action or sequence of actions α was intended at step i, where $0 \leq i < n$.

If l is a literal, by $obs(l, i)$ we will denote $observed(f, true, i)$ if $l = f$ and $observed(f, false, i)$ if $l = \neg f$. If p is a set of literals, $obs(p, i) = \{obs(l, i) : l \in p\}$. If for every fluent f in the signature either $observed(f, true, 0) \in \Gamma_n$ or $observed(f, false, 0) \in \Gamma_n$, then we say that *the initial situation of* Γ_n *is complete*. We use the notation $\alpha = \langle a_1, a_2, \ldots, a_n \rangle$ for sequences of actions (i.e., the first element of α is the action or sequence a_1, the second is a_2, ..., and the n-th and final element of α is the action or sequence a_n). The semantics of a history Γ_n is given by the following definition:

Definition 16 (Model of a History)**.** Let Γ_n be a history of a system description $\mathcal{D}$ up to time step n. Let $M = \langle \sigma_0, a_0, \sigma_1, \ldots, a_{m-1}, \sigma_m \rangle$ be a trajectory of $\mathcal{T}(\mathcal{D})$.

(a) We first define what it means for M to *satisfy* the statements of Γ_n:

1. M is said to satisfy a statement $intended(a, i)$, where a is an action, if there is $j \geq i$ such that $a \subseteq a_j$ and for every $i \leq k < j$, a is not executable at σ_k (i.e., our system description $\mathcal{D}$ contains an axiom of the type "**impossible** a **if** $l_1, \ldots, l_n$" such that $\{l_1, \ldots, l_n\} \subseteq \sigma_k$). We say that $j+1$ is the point of a's completion, and that a is supported at j.
2. M is said to satisfy a statement $intended(\alpha, i)$ where $\alpha = \langle a'_1, \ldots, a'_n \rangle$, and $n > 1$, if M satisfies $intended(a'_1, i)$ and $intended(\alpha', j)$, where $\alpha' = \langle a'_2, \ldots, a'_n \rangle$ and j is the point of a'_1's completion.
3. M is said to satisfy a statement $obs(l, i)$ if $l \in \sigma_i$.
4. M is said to satisfy a statement $happened(a, i)$ if $a \subseteq a_i$. We say that a is supported at i. M is said to satisfy a statement $\neg happened(a, i)$ if $a \nsubseteq a_i$.

(b) M is a *model* of Γ_n if it satisfies all the statements of Γ_n, and for $0 \leq i < m$, all elements of a_i are supported at i.

(c) Γ_n is *consistent* if it has a model.

(d) A literal l *holds* in a model M of Γ_n at time $i \leq n$ if $l \in \sigma_i$; Γ_n *entails* $h(l, i)$ if, for every model M of Γ_n, $M \models h(l, i)$.

(We use $M \models h(l, i)$ to denote that l holds in model M, and $\Gamma_n \models h(l, i)$ to denote that Γ_n *entails* $h(l, i)$.)

Note that a consistent history may have more than one model if non-deterministic actions are involved. We now formally introduce the program, Ω, that describes the connection between observations in the history and the hypothetical relations *occurs* and *holds* in our ASP encoding of a system description. It can be viewed as a reasoning module for performing temporal projection.

Definition 17 (ASP Module for Temporal Projection)**.** The ASP module for temporal projection, Ω, is a program consisting of the following rules:

$$
\begin{array}{lcl}
occurs(A, I) & \leftarrow & happened(A, I). \\
\neg occurs(A, I) & \leftarrow & \neg happened(A, I). \\
h(F, 0) & \leftarrow & observed(F, true, 0). \\
\neg h(F, 0) & \leftarrow & observed(F, false, 0). \\
 & \leftarrow & observed(F, true, I), \\
 & & not\ h(F, I). \\
 & \leftarrow & observed(F, false, I), \\
 & & not\ \neg h(F, I).
\end{array}
$$

Let $\mathcal{D}$ be a weakly acyclic system description of $\mathcal{ALM}$, Γ_n be a history of $\mathcal{D}$ up to time step n, $\mathcal{I}$ be the theory of intentions from Appendix 11.2, and Ω be the program defined above. By $P(\mathcal{D}, \Gamma_n)$ we denote the logic program obtained by adding Γ_n, $\mathcal{I}$, and Ω to the ASP encoding of the system description $\mathcal{D}$. The following terminology will be useful for describing the relationship between answer sets of $P(\mathcal{D}, \Gamma_n)$ and models of Γ_n.

Definition 18 (Answer Set Defining a Sequence)**.** Let Γ_n be a history of $\mathcal{D}$ and A be a set of literals over $lit(P(\mathcal{D}, \Gamma_n))$. We say that A *defines* the sequence

$$\langle \sigma_0, a_0, \sigma_1, \ldots, a_{m-1}, \sigma_m \rangle$$

if $\sigma_i = \{l \mid h(l, i) \in A\}$ for any $0 \leq i \leq m$, and $a_k = \{a \mid occurs(a, k)\}$ for any $0 \leq k < m$.

We can now formulate our theorem about the ASP algorithm:

Theorem 2. (Soundness and Completeness of the ASP Algorithm) Let $\mathcal{D}$ be a weakly acyclic system description of $\mathcal{ALM}$ defining a transition diagram $\mathcal{T}(\mathcal{D})$.

If the history Γ_n is *consistent* with respect to $\mathcal{T}(\mathcal{D})$ and its initial situation is *complete*, then M is a model of Γ_n **iff** M is defined by some answer set of $P(\mathcal{D}, \Gamma_n)$.

Proof. Previous work [Balduccini & Gelfond, 2003b, Baral & Gelfond, 2005] showed similar results in the context of the *non*-modular action language $\mathcal{AL}$ *without* defined fluents. We will use those results in our proof.

(a) Let $\tau(\mathcal{D})$ be the $\mathcal{AL}$ translation of the system description $\mathcal{D}$ of $\mathcal{ALM}$.

(b) By Theorem 1, there is a system description $\tau'(\mathcal{D})$ with the same signature as $\tau(\mathcal{D})$ minus the defined fluents, such that $\tau'(\mathcal{D})$ is a residue of $\tau(\mathcal{D})$

(c) By Lemma 5 from [Balduccini & Gelfond, 2003b] for $\tau'(\mathcal{D})$ and Theorem 1 from [Baral & Gelfond, 2005], we obtain soundness and completeness.

□

Theorem 2 indicates that *the ASP system will produce correct and complete answers to all questions about temporal projection.* This theorem can be easily extended to cover questions about planning or diagnosis, based on previous results on $\mathcal{AL}$.

APPENDIX F
MATHEMATICAL RESULTS FOR THE $\mathcal{F}$LORA-*2* SYSTEM

In the following theorems, let $\mathcal{D}$ be a weakly acyclic system description of $\mathcal{ALM}$, Γ_n be a history of $\mathcal{D}$, $\mathcal{I}$ be the theory of intentions, and Ω_F be the $\mathcal{F}$LORA-*2* module for temporal projection described in Section 9.3. By $P_F(\mathcal{D}, \Gamma_n)$ we denote the logic program obtained by adding Γ_n, $\mathcal{I}$, and Ω_F to the $\mathcal{F}$LORA-*2* encoding of the system description $\mathcal{D}$.

Lemma 2. Let $\mathcal{D}$ be a weakly acyclic system description of $\mathcal{ALM}$ defining a transition diagram $\mathcal{T}(\mathcal{D})$. If the history Γ_n is *consistent* with respect to $\mathcal{T}(\mathcal{D})$ and its initial situation is *complete*, then:

1. XSB returns f as an answer to the query $holds(F, i)$ **iff** $holds(f, i)$ belongs to the well-founded model of $P_F(\mathcal{D}, \Gamma_n)$.

2. XSB returns f as an answer to the query $\neg holds(F, i)$ **iff** $\neg holds(f, i)$ belongs to the well-founded model of $P_F(\mathcal{D}, \Gamma_n)$.

Proof. We need to show that the SLG algorithm of XSB is sound and complete with respect to the well-founded model for the program $P_F(\mathcal{D}, \Gamma_n)$. Unsoundness can be due to floundering; incompleteness can be due to non-termination [Chen & Warren, 1996]. Hence, we need to show that $P_F(\mathcal{D}, \Gamma_n)$ is occur-check free, does not flounder and the SLG algorithm terminates. Let $\tau(\mathcal{D})$ be the $\mathcal{AL}$ translation of the system description $\mathcal{D}$ of $\mathcal{ALM}$. Hence, $P_F(\mathcal{D}, \Gamma_n)$ is $\Pi_F(\tau(\mathcal{D})) \cup \mathcal{I} \cup \Gamma_n \cup \Omega_F$. For simplicity, we will denote $P_F(\mathcal{D}, \Gamma_n)$ by P_F.

(a) P_F is *occur-check free*: P_F is well-moded [Dembinski & Maluszynski, 1985] for the following input-output specification:

$fluent(+,-)$	$holds(-,+)$	$observed(+,+,+)$
$defeated(+,+)$	$occurs(+,+)$	$happened(+,+)$
		$intend(+,+)$

As there is no rule in P_F whose head contains more than one occurrence of the same variable in its output positions, then, based on the result of Apt and Pellegrini [Apt & Pellegrini, 1994], P_F is occur-check free.

(b) P_F *does not flounder*: P_F is well-moded for the input-output specification in (a) and all predicate symbols occurring under negation as failure (i.e., $defeated$, $occurs$, and $intend$) are moded completely by input. Hence, based on results by Apt and Pellegrini [Apt & Pellegrini, 1994], and Stroetman [Stroetman, 1993], P_F does not flounder.

(c) The SLG resolution of XSB computation *terminates*:

1. P_F has a finite universe. The syntax of system descriptions of $\mathcal{ALM}$ only allows for specifying a finite number of classes, a finite number of parameters/ attributes for fluents/ actions, and a finite number of instances of the declared classes. Hence, the system description $\mathcal{D}$ has a finite signature. Additionally, we assume that the number of time steps declared in P_F is finite, ranging from 0 to m, where $m > n$. Besides atoms describing the signature of $\mathcal{D}$, the program P_F may contain a finite number of new atoms. For example, if there are N non-static fluents in the signature of $\mathcal{D}$, then P_F will contain up to $4 * N * m$ new atoms (i.e., for each non-static fluent f it may contain $holds(f, i)$, $\neg holds(f, i)$, $observed(f, true, i)$, and $observed(f, false, i)$). Note that we view $\neg holds$ as a new predicate symbol. Similarly, for actions and sequences of actions.

2. The SLG resolution of XSB terminates for programs with a finite universe [Chen & Warren, 1996].

3. From 1 and 2, we obtain the termination of the SLG resolution for P_F.

□

We can now state that, for any consistent history with a complete initial situation, the trajectory computed by the $\mathcal{F}$LORA-*2* algorithm is sound with respect to the

specification:[2] In the following theorems, a *history* is defined as in Definition 15, and a *model of a history* as in Definition 16.

Theorem 3. (Soundness of the $\mathcal{F}$LORA-*2* Algorithm) Let $\mathcal{D}$ be a weakly acyclic system description of $\mathcal{ALM}$ defining a transition diagram $\mathcal{T}(\mathcal{D})$. Let the history Γ_n be *consistent* with respect to $\mathcal{T}(\mathcal{D})$ and the initial situation of Γ_n be *complete*. Let l be a fluent literal, and let $0 \leq i \leq n$.

If $h(l, i)$ is in the well-founded model of $P_F(\mathcal{D}, \Gamma_n)$, then

$$\forall \text{ model } M \text{ of } \Gamma_n, M \models h(l, i)$$

Proof. We have to show that the well-founded model W of $P_F(\mathcal{D}, \Gamma_n)$ is sound with respect to the specification. Note that the semantics of an $\mathcal{ALM}$ system description is given by translating it into an equivalent $\mathcal{AL}$ system description. The semantics of an $\mathcal{AL}$ system description is in turn given by an ASP program, Π (see Section 2.5.1).

(a) Let $\tau(\mathcal{D})$ be the $\mathcal{AL}$ translation of the system description $\mathcal{D}$ of $\mathcal{ALM}$.

(b) From (a), $P_F(\mathcal{D}, \Gamma_n)$ is the $\mathcal{F}$LORA-*2* program $\Pi_F(\tau(\mathcal{D})) \cup \mathcal{I} \cup \Gamma_n \cup \Omega_F$. For simplicity, we will denote it by P_F.

(c) Let P_A (A from ASP) denote the ASP program $\Pi(\tau(\mathcal{D})) \cup \mathcal{I} \cup \Gamma_n \cup \Omega$ used in the ASP question answering method.

(d) Under the given conditions (i.e., Γ_n is a consistent history of $\mathcal{D}$ with a complete initial situation), P_F has the same *answer sets* as P_A modulo the common signature.

(We will further develop this part of the proof below).

(e) Based on Corollary 5.7 from [Van Gelder et al., 1991], the well-founded model W of P_F is compatible with every answer set of P_F.

[2]This assumes that the implementation of $\mathcal{F}$LORA-*2* is sound with respect to XSB.

(f) From (d) and (e), W is compatible with every answer set of P_A.

(g) From Theorem 2, every answer set of P_A defines a trajectory of Γ_n.

(h) From (f) and (g), W is compatible with all possible trajectories of Γ_n.

Let us now give some details about point (d) above, which says that P_F has the same *answer sets* as P_A modulo the common signature. Here is a sketch of this part of the proof:

(i) First of all, we introduce some notation.

Let

$$\begin{array}{rcl} Z & =_{def} & \{r : r \text{ is a rule of type 2.6 or 2.7 from } \Pi(\tau(\mathcal{D}))\} \\ & = & \{r : r \text{ is a rule of type 0.4 or 0.6 from } \Pi_F(\tau(\mathcal{D}))\} \end{array} \tag{0.11}$$

(ii) Left-to-right: For every answer set of P_F there is an equivalent answer set of P_A, modulo the common signature.

- Let $\mathcal{S}_F$ be an arbitrary answer set of of P_F.
- Let $\mathcal{S}_A$ be constructed from $\mathcal{S}_F$ as follows:

$$\mathcal{S}_A =_{def} \mathcal{S}_F \backslash \{ defeated(\bar{l}, i) : \exists r \in Z \text{ s.t. } head(r) = h(l, i) \wedge body(r) \subseteq \mathcal{S}_F \} \tag{0.12}$$

- Note that $\mathcal{S}_A$ and $\mathcal{S}_F$ are identical modulo the common signature.
- We show that $\mathcal{S}_A$ is an answer set of P_A.

  This is done by the usual method of showing that:

  (1) $\mathcal{S}_A$ is closed under the rules of $P_A^{\mathcal{S}_A}$, and

  (2) $\mathcal{S}_A$ is a minimal set closed under the rules of $P_A^{\mathcal{S}_A}$ (by induction on the immediate consequence operator).

(iii) Right-to-left: For every answer set of P_A there is an equivalent answer set of P_F, modulo the common signature.

– Let $\mathcal{S}_A$ be an arbitrary answer set of of P_A.

– Let $\mathcal{S}_F$ be constructed from $\mathcal{S}_A$ as follows:

$$\mathcal{S}_F =_{def} \mathcal{S}_A \cup \{ defeated(\bar{l}, i) : \exists r \in Z \text{ s.t. } head(r) = h(l, i) \wedge body(r) \subseteq \mathcal{S}_A \} \tag{0.13}$$

– Note that $\mathcal{S}_F$ and $\mathcal{S}_A$ are identical modulo the common signature.

– We show that $\mathcal{S}_F$ is an answer set of P_F.

As before, this is done by showing that:

(1) $\mathcal{S}_F$ is closed under the rules of $P_F^{\mathcal{S}_F}$, and

(2) $\mathcal{S}_F$ is a minimal set closed under the rules of $P_F^{\mathcal{S}_F}$ (by induction on the immediate consequence operator).

(iv) From (ii) and (iii), we obtain that P_F and P_A have the same answer sets modulo the common signature.

□

We continue discussing the completeness of the $\mathcal{F}$LORA-*2* algorithm. In the general case, this algorithm is not complete. The following well known example illustrates a case in which it is incomplete.

Example 31. [Incompleteness]
Let $\mathcal{D}$ be the following system description:

a	**causes**	f
$\neg g_1$	**if**	f, g_2
$\neg g_2$	**if**	f, g_1
d	**if**	g_1
d	**if**	g_2

with inertial fluents f, g_1, and g_2, and defined fluent d. Let us consider the history:

$$\Gamma_n \;=\; \{\; observed(f, false, 0), observed(g_1, true, 0), observed(g_2, true, 0),$$
$$happened(a, 0)\}.$$

This system has two possible trajectories: in one of them f, g_1, and d are true and g_2 is false at step 1; in the other one f, g_2, and d are true and g_1 is false at step 1. However, the only information about step 1 that the well-founded model of $P_F(\mathcal{D}, \Gamma_n)$ can provide is that f holds in all possible trajectories of the system. The values of the remaining fluents, g_1, g_2, and d, are unknown.

Our goal is to find a class of system descriptions for which the $\mathcal{F}$LORA-*2* algorithm is complete. We first introduce the following definition:

Definition 19 (Simple System Descriptions)**.** A *simple* system description is a system description of $\mathcal{ALM}$ such that:

1. there are no circular dependencies between fluents and

2. all executability conditions "**impossible** $a_1, \ldots, a_k$ **if** p" have $k = 1$, i.e., they are of the form "**impossible** a_1 **if** p."

We can now express the sufficient condition for completeness:

Theorem 4. (Sufficient Condition for Completeness of the $\mathcal{F}$LORA-*2* Algorithm) Let $\mathcal{D}$ be a weakly acyclic *simple* system description of $\mathcal{ALM}$ defining a transition diagram $\mathcal{T}(\mathcal{D})$; let its history Γ_n be *consistent* with respect to $\mathcal{T}(\mathcal{D})$ and its initial situation be *complete*; let l be a fluent literal; and $0 \leq i \leq n$.

If M is a model of Γ_n and $h(l, i) \in M$ then $h(l, i) \in W$, where W is the well-founded model of $P_F(\mathcal{D}, \Gamma_n)$.

Proof. We first introduce some notation:

(a) For any logic program P, let P^* be the *general* logic program obtained from P by replacing all occurrences of $\neg pred$ by n_pred, for every predicate $pred$ from the signature of P.

(b) As before, let $\tau(\mathcal{D})$ be the $\mathcal{AL}$ translation of the system description $\mathcal{D}$ of $\mathcal{ALM}$.

(c) Hence, $P_F(\mathcal{D}, \Gamma_n)$ is the $\mathcal{F}$LORA-*2* program $\Pi_F(\tau(\mathcal{D})) \cup \mathcal{I} \cup \Gamma_n \cup \Omega_F$. For simplicity, we will denote it by P_F.

(d) Let P_A (A from ASP) denote the ASP program $\Pi(\tau(\mathcal{D})) \cup \mathcal{I} \cup \Gamma_n \cup \Omega$ used in the ASP question answering method.

We prove Theorem 4 by showing the following:

(e) The general logic program P_F^* is a *locally stratified program* when $\mathcal{D}$ is a *simple* system description and Γ_n is a consistent history of $\mathcal{D}$ with a complete initial situation.

(We will further develop this part of the proof below.)

(f) From (e), we obtain that P_F has a unique answer set, equivalent to its well-founded model.

(g) We show that the unique answer set of P_F is equivalent to the unique answer set of P_A modulo the common signature.

(This was already shown in the proof of Theorem 3.)

(h) The unique answer set of P_A is complete with respect to the specification (By Theorem 2).

(i) From (g) and (h), we obtain that the well-founded model of P_F is also complete with respect to the specification.

Let us now discuss in more detail the proof for point(e), which says that the program P_F^* is locally stratified:

(i) We first define the *graph of non-static fluents* $\mathcal{G}_{ns}(\mathcal{D})$ *= (V, E)* as a directed graph where:

1. V is the set of all non-static (i.e., inertial or defined) fluent names from the signature of $\mathcal{D}$

2. For every static causal law "l **if** $p_1, \ldots, p_k$" from $\mathcal{D}$, where l is non-static, and for all $1 \leq i \leq k$ such that p_i is non-static, $(p_i, l) \in E$.

(ii) Given that $\mathcal{D}$ is a *simple* system description, $\mathcal{G}_{ns}(\mathcal{D}) = (V, E)$ is acyclic.

(iii) We define the mapping $\alpha : V \rightarrow \{1, 2, \ldots, n\}$ as follows:

1. If $N \in V$ and N is a source, then: $\alpha(N) = 1$
2. For every $N \in V$ such that $(N_1, N) \in E$, ..., $(N_m, N) \in E$: $\alpha(N) = max\{\alpha(N_1), \ldots, \alpha(N_m)\} + 1$
3. $n = max\{\alpha(N) \mid N \text{ is a sink}\}$

(iv) Let $k = n + 1$.

(v) We prove (e) (i.e., P^*_F is locally stratified) by using the mapping α defined above and the ranking, ρ, defined as follows:

$$
\begin{array}{lcll}
\rho(happened(a,i)) & = & \rho(n\_happened(a,i)) = 0 & \\
\rho(observed(f,true,i)) & = & \rho(observed(f,false,i)) = 0 & \text{if } f \text{ is non-static} \\
\rho(n\_intend(\alpha,i)) & = & 0 & \\
\rho(f) & = & \rho(n\_f) = 0 & \text{if } f \text{ is static} \\
\rho(defeated(\bar{l},i)) & = & k*i + (\alpha(l) - 1) & \\
\rho(holds(f,i)) & = & \rho(n\_holds(f,i)) = k*i + \alpha(f) & \\
 & & & \text{if } f \text{ is non-static} \\
\rho(n\_occurs(a,i)) & = & k*i + n & \\
\rho(occurs(a,i)) & = & k*(i+1) & \\
\rho(intend(\alpha,i)) & = & k*(i+1) & \\
\rho(ends(\alpha,i)) & = & k*(i+1) & \\
\rho(inconsistency(i)) & = & k*(i+1) &
\end{array}
$$

(vi) It can be easily checked that ρ is a local stratification for P^*_F.

□

APPENDIX G
TRANSLATION FROM $\mathcal{ALM}$ INTO MAD

In this section, we define a translation of system descriptions of $\mathcal{ALM}$ into action descriptions of MAD. Our goal is to create a translation that would allow us to compare the modular structure of $\mathcal{ALM}$ with that of MAD. We will use the syntax of MAD described in [Erdoğan, 2008]. We will begin by defining a translation for closed modules of $\mathcal{ALM}$, as the theory of any system description of $\mathcal{ALM}$ is equivalent to a closed module. Then, we will discuss the translation of a system description.

G.1 Closed Module

Let us consider a module M of $\mathcal{ALM}$ with the signature $\Sigma = \langle \mathcal{P}, \mathcal{C}, \leq_{\mathcal{C}}, \mathcal{F} \rangle$. We translate it into a MAD action description, $AD(M)$, with the following sections:

sorts
inclusions
module M
module M_c_1
...
module M_c_m

where $c_1, \ldots, c_m$ are action classes in the class hierarchy of Σ. $AD(M)$ may contain some additional modules derived from executability conditions for instances of *several* different actions. Module M may have the following sections:

fluents
variables
axioms

Modules $M\_c_1, \ldots, M\_c_m$, if they exist, will have the sections:

actions

variables

axioms

import ...

where the number of **import** sections in a module M_c_i, $1 \leq i \leq m$, will be equal to the number of parents of c_i in the class hierarchy.

The general idea for the translation of M into $AD(M)$ is as follows:

- For every class $c \not\leq_{\mathcal{C}} actions$ of M:

  Declare it in the **sorts** section of $AD(M)$, declare its specialization relations in **inclusions**, and declare its attributes as rigid fluents in module M.

- For every class $c \leq_{\mathcal{C}} actions$ of M, where c has k attributes:

  Declare c as a term with k arguments in module M_c of $AD(M)$. If c has other parents than *actions* in the class hierarchy, then add import statements with renaming clauses to describe its specialization relations; otherwise import module M without renaming clauses.

- Place state constraints in the **axioms** section of module M.

- Place dynamic causal laws and triggers about an action class c in the **axioms** section of module M_c, i.e., the module in which c is declared.

- Place executability conditions about instances of a single action class, c, in M_c. Place executability conditions about instances of several different action classes, $c_1, \ldots, c_n$, in a new module $M\_c_1\_\ldots\_c_n$, which imports modules $M\_c_1, \ldots, M\_c_n$.

We use the $\mathcal{ALM}$ syntax for variables and constant symbols (initial uppercase letter and initial lowercase letter, respectively) until the last moment; our last step

of the translation will be to rename variables and constants according to the MAD syntax (initial lowercase letter for variables and initial uppercase letter for constants).

We begin with some simplifying assumptions and preliminary definitions. Then, we show how we translate class declarations, function declarations, and axioms of M.

G.1.1 Simplifying Assumptions

We make the following simplifying assumptions about M and explain our reasons for such assumptions:

1. Functions other than attributes are not defined on, and do not range over classes that are special cases of *actions*.

   Why: Fluents of MAD must be defined on, and range over either primitive sorts or the built-in class *action*. A possible solution would be to transform each such function

   $$f : c_1 \times \cdots \times c_n \rightarrow c_{n+1}$$

   of $\mathcal{ALM}$ into a boolean function

   $$f(c'_1, \ldots, c'_n, c'_{n+1})$$

   of MAD, where $\forall 1 \leq i \leq n+1$, $c'_i =_{def} action$ if $c \leq_{\mathcal{C}} actions$ and $c'_i =_{def} c_i$ otherwise, and to add the following axiom:

   $$\textbf{default } \neg f(X_1, \ldots, X_n, X_{n+1});$$

   However, this implies that f will be defined on all actions, whereas the $\mathcal{ALM}$ counterpart of f may only be defined on some special case class of actions.

2. Attributes of every action class (i.e., a class that is a special case of *actions*) are of the form $attr\_name : c \rightarrow c_{n+1}$ where $n = 0$.

   Why: The name of an action in MAD is a term with one parameter for each of its possible attributes. If a class c of $\mathcal{ALM}$ has an attribute $attr\_name : c \times c_1 \times \cdots \times c_n \rightarrow c_{n+1}$ where $n > 0$, then the number of possible attributes of

an action of this class depends on the interpretations of classes $c_1, \ldots, c_n$. In the current translation of $\mathcal{ALM}$ into MAD, we wanted our translation of modules to be independent from the interpretation, which seems to be unavoidable here.

Another possible translation of $\mathcal{ALM}$ system descriptions into MAD can be created starting from the interpretation (i.e., the structure) of the system description. In that case, we would replace variables that stand for action instances in axioms by their possible instantiations. However, such a translation would produce a unique module of MAD. This would be inconvenient, as it would not allow us to compare the modular structure of $\mathcal{ALM}$ with that of MAD.

3. Action classes do not have constraints on attributes.

   Why: Such a constraint could be translated into MAD as an executability condition, but it would not exactly express the same intuition. For now, we do not foresee any better solution.

4. State constraints do not contain literals of the form $instance(V, c)$ or $\neg instance(V, c)$ where c is an action class. Dynamic causal laws, executability conditions and triggers do not contain literals of the form $instance(V, c)$ or $\neg instance(V, c)$ unless they also contain a matching literal of the form $occurs(V)$ or $\neg occurs(V)$.

   Why: In $\mathcal{ALM}$, we distinguish between an instance of an action class and the execution of such an instance by the use of *instance* or *occurs*, respectively. Such a distinction is normally not made in MAD. The only exception of which we are aware is a statement of the form:

   $$\textbf{nonexecutable } a \ \& \ a_1 \textbf{ if } a \ != \ a_1$$

   where: "in the first two occurrences, the variables are treated as the action atoms and the latter occurrences on either side of the inequality sign are treated as the names of the actions" [Erdoğan, 2008]. We remind the reader that an action atom a of MAD (equivalent to $a = true$) says that action a is executed.

G.1.2 Preliminary Definitions

In what follows, we will use the following definitions.

Definition 20. If V is a variable appearing in an axiom r of a module M of $\mathcal{ALM}$, then by the *sort of* V *in* r we mean the most specific class among the classes in the signature of M with whose instances V can be replaced in r.

Example 32. In the axiom:

$$\begin{aligned} loc\_in(A) = D \quad \textbf{if} \quad & instance(X, move), \\ & occurs(X), \\ & actor(X) = A, \\ & dest(X) = D. \end{aligned}$$

where $loc\_in : things \rightarrow points$, $actor : move \rightarrow movers$, and $dest : move \rightarrow points$, the sort of A is $movers$, the sort of D is $points$, and the sort of X is $move$.

Definition 21. If c is an action class (i.e., $c \leq_{\mathcal{C}} actions$), by $attr(c)$ we denote the set of attributes of c, meaning the set of functions of the type $attr\_name : c_0 \rightarrow c_1$ in $\mathcal{F}$ such that $c = c_0$ or $c \leq_{\mathcal{C}} c_0$.

Example 33. Assuming that action classes *move* and *carry* were already introduced,

$$\begin{aligned} attr(move) \quad = \quad \{ \quad & actor : move \rightarrow movers, \\ & origin : move \rightarrow points, \\ & dest : move \rightarrow points\} \end{aligned}$$

and

$$\begin{aligned} attr(carry) \quad = \quad \{ \quad & actor : move \rightarrow movers, \\ & origin : move \rightarrow points, \\ & dest : move \rightarrow points, \\ & carried\_thing : carry \rightarrow carriables\} \end{aligned}$$

Definition 22. Let $<$ be some fixed strict total order defined on the attributes of $\mathcal{F}$, for example, the alphabetical order. By $attr(c, <)$ we denote the tuple obtained by arranging the attribute names in $attr(c)$ from the smallest to the greatest, according

to the order $<$ (i.e., the tuple $(attr_0, \ldots, attr_n)$ such that $attr_i : c_i \rightarrow c_i' \in attr(c)$, $\forall 1 \leq i \leq n$, and if $attr_i < attr_j$ then $i < j$).

Example 34. Assuming that $<$ is the alphabetical order:

$attr(move, <) = (actor, dest, origin)$ and
$attr(carry, <) = (actor, carried\_thing, dest, origin)$.

Definition 23. If x is a string, then by

$$upc(x)$$

we denote the string obtained from x by replacing its first character with its upper-case version; similarly, by

$$lwc(x)$$

we denote the string obtained by replacing the first character of x with its lower-case version. If $t = (x_1, \ldots, x_n)$ is a tuple of strings, then

$$upc(t) =_{def} (upc(x_1), \ldots, upc(x_n))$$

Similarly for $lwc(t)$.

Definition 24. If c is an action class, then:

$$\overline{Attr_c} \ =_{def} \ upc(attr(c, <))$$

$$\alpha(c) \ =_{def} \ c(\overline{Attr_c})$$

Example 35. Assuming that $<$ is the alphabetical order:

$\overline{Attr_{move}} = (Actor, Dest, Origin)$

and

$\alpha(move) = move(Actor, Dest, Origin)$.

Definition 25. If c is an action class, then by

$$l(c)$$

we denote the length of the longest path from c to *actions* in the class hierarchy of Σ.

Example 36. Assuming that action classes *move* and *carry* were already introduced, $l(move) = 1$ and $l(carry) = 2$.

G.1.3 Translation of Class Declarations

We show how we translate a class declaration

c : $pc_1, \ldots, pc_k$
attributes
[$attr\_name : c_1 \times \cdots \times c_n \rightarrow c_{n+1}$]+
constraints
[$attribute\_constraint.$]+

of M into MAD. We distinguish between two cases: classes of objects and classes of actions.

Case 1: $c \not\leq_{\mathcal{C}} actions$

1. Add to the **sorts** of $AD(M)$:

$$c;$$

2. Add to the **inclusions** of $AD(M)$:

$$c << pc_1;$$
$$\ldots$$
$$c << pc_k;$$

3. For every attribute $attr\_name : c \times c_1 \times \cdots \times c_n \rightarrow c_{n+1}$ in $\mathcal{F}$, add to the **fluents** of module M:

$$attr\_name(c, c_1, \ldots, c_n) : rigid(c_{n+1});$$

where "(c_{n+1})" can be omitted if $c_{n+1} = booleans$.

4. For every attribute constraint r of c:

   (a) For every variable γ of sort s in r:

   If γ does not appear in the **variables** section of module M, then add to this section:

   $$\gamma : s;$$

   Otherwise, rename γ with a name γ' not yet used in M, replace γ by γ' in r, and add to the **variables** of M:

   $$\gamma' : s;$$

   (b) Remove literals of the form $instance(\gamma, c)$ and replace literals of the form $\neg instance(\gamma, c)$ by $\neg c(\gamma)$.

   (c) Add the resulting axiom to the **axioms** of module M.

Case 2: $c \leq_{\mathcal{C}} actions$

Let $attr(c, <) = (attr_1, \ldots, attr_n)$ where $attr_i : c'_i \rightarrow c_i$, $1 \leq i \leq n$ and $c \leq_{\mathcal{C}} c'_i$.

1. Add to $AD(M)$ a module called M_c.

2. Add to the **actions** of module M_c:

   $$c(c_1, \ldots, c_n);$$

3. For every attribute $attr_i : c'_i \rightarrow c_i$ in $attr(c)$, if $upc(attr_i) = Attr_i$ then add to the **variables** of module M_c:

   $$Attr_i : c_i;$$

4. Add to the **axioms** of module M_c:

   $$\textbf{exogenous } c(\overline{Attr_c});$$

5. If $l(c) = 1$, then add to module M_c the statement:

$$\textbf{import } M;$$

Otherwise, for every $pc_i \neq actions$, $1 \leq i \leq k$, in the declaration of c, add to module M_c an import statement:

$$\textbf{import } M\_pc_i;$$
$$pc_i(\overline{Attr_{pc_i}}) \textbf{ is } c(\overline{Attr_c});$$

Note that, since c is a special case of pc_i, $\overline{Attr_{pc_i}}$ is a subsequence of $\overline{Attr_c}$.

G.1.4 Translation of Function Declarations

We consider a function declaration:

$$type\ f : c_1 \times \cdots \times c_n \rightarrow c_{n+1}$$

where *type* is one of the keywords **static**, **inertial**, or **defined**.

It will be translated into MAD by adding to the **fluents** of module M:

$$f(c_1, \ldots, c_n) : type'(c_{n+1})$$

where "(c_{n+1})" can be omitted if $c_{n+1} = booleans$ and

$type' = rigid$ if $type =$ **static**

$type' = simple$ if $type =$ **inertial**

$type' = staticallyDetermined$ if $type =$ **defined**

Additionally, if $type =$ **inertial**, then add to the **variables** of M new variables X_1, ..., X_n of sorts c_1, ..., c_n, respectively, and add to the **axioms** of M:

$$\textbf{inertial } f(X_1, \ldots, X_n);$$

G.1.5 Translation of Axioms

Dynamic Causal Laws

We consider a dynamic causal law r of the form:

$$l \textbf{ if } instance(\chi, c),\ occurs(\chi),\ cond$$

We translate r into its MAD counterpart by applying the following consecutive steps:

1. Remove from the body of r the pair of atoms $\{instance(\chi, c), occurs(\chi)\}$ and replace attribute atoms of the type $attr(\chi) = t$ by $upc(attr) = t$.

2. Rewrite r as

$$\alpha(c) \textbf{ causes } l \textbf{ if } cond;$$

3. Remove remaining literals of the type $instance(\delta, c')$ and replace literals of the type $\neg instance(\delta, c')$ by $\neg c'(\delta)$.

4. For every remaining variable γ, where γ is of sort s in the original r:

   If γ does not appear in the **variables** section of module M_c, then add to this section:

$$\gamma : s;$$

   Otherwise, rename γ with a name γ' not yet used in M_c, replace γ by γ' in r, and add to the **variables** of M_c:

$$\gamma' : s;$$

5. Add the resulting rule to module M_c.

State Constraints

We consider a state constraint r of the form:

$$l \textbf{ if } p$$

We translate r into its MAD counterpart by applying the following consecutive steps:

1. Remove literals of the type $instance(\chi, c)$ and replace literals of the type $\neg instance(\chi, c)$ by $\neg c(\chi)$.

2. For every remaining variable γ, where γ is of sort s in the original r:

   If γ does not appear in the **variables** section of module M, then add to this section:

   $$\gamma : s;$$

   Otherwise, rename γ with a name γ' not yet used in M, replace γ by γ' in r, and add to the **variables** of M:

   $$\gamma' : s;$$

3. Add the resulting rule to module M.

Executability Conditions

We consider an executability condition r of the form:

$$\neg occurs(\chi) \textbf{ if } instance(\chi, c),\ cond$$

We translate r into its MAD counterpart by applying the following consecutive steps:

1. If r is about instances of only one action c, then let M_r refer to M_c.

   If r is about instances of several different actions, c_1, ..., c_n, then add a new module $M\_c_1\_ \ldots \_c_n$ to $AD(M)$ (unless it already exists) and add to this module the following import statements without renaming clauses:

   import c_1;

   ...

   import c_n;

   Let M_r refer to $M\_c_1\_ \ldots \_c_n$ in this case.

2. Remove the literals $\neg occurs(\chi)$ and $instance(\chi, c)$ from the head and body of r, respectively, and replace attribute atoms of the type $attr(\chi) = t$ by $upc(attr) = t$.

3. Rewrite r as

$$\textbf{nonexecutable } \alpha(c) \textbf{ if } cond;$$

4. Replace pairs of atoms of the type $\{occurs(\delta), instance(\delta, c')\}$ in p by $\alpha(c')$ and replace attribute atoms of the type $attr(\delta) = t$ by $upc(attr) = t$.

5. Remove remaining literals of the type $instance(\gamma, c')$ and replace literals of the type $\neg instance(\gamma, c')$ by $\neg c'(\gamma)$.

6. For every remaining variable γ, where γ is of sort s in the original r:

   If γ does not appear in the **variables** section of module M_r, then add to this section:

$$\gamma : s;$$

   Otherwise, rename γ with a name γ' not yet used in M_r, replace γ by γ' in r, and add to the **variables** of M_r:

$$\gamma' : s;$$

7. Add the resulting rule to module M_r.

Triggers

We consider a trigger r of the form:

$$occurs(\chi) \textbf{ if } instance(\chi, c),\ cond$$

We translate r into its MAD counterpart by applying the following consecutive steps:

1. Remove the literals $occurs(\chi)$ and $instance(\chi, c)$ from the head and body of r, respectively, and replace attribute atoms of the type $attr(\chi) = t$ by $upc(attr) = t$.

2. Rewrite r as

$$\alpha(c) \textbf{ if } cond;$$

3. Remove remaining literals of the type $instance(\delta, c')$ and replace literals of the type $\neg instance(\delta, c')$ by $\neg c'(V)$.

4. For every remaining variable γ, where γ is of sort s in the original r:

   If γ does not appear in the **variables** section of module M_c, then add to this section:

$$\gamma : s;$$

   Otherwise, rename γ with a name γ' not yet used in M_c, replace γ by γ' in r, and add to the **variables** of M_c:

$$\gamma' : s;$$

5. Add the resulting rule to module M_c.

G.1.6 Completing the Translation

Finally, we need to add to $AD(M)$ the declarations of the predefined symbols in $\mathcal{P}$, rename the class *booleans* as *boolean*, and apply the operation *upc* (upper-case) to variables and the operation *lwc* (lower-case) to constants.

Note that this translation can be easily expanded to theories, which may contain both closed and open modules. The MAD counterpart of a theory T would an action description $AD(T)$. For each module in T, several modules (as shown above) would be added to $AD(T)$. If M is an *open* module of T that depends on modules M_1, ..., M_n, then the MAD module with the same name, M, would import all the MAD modules that resulted from the translation of M_1, ..., M_n.

For simplicity, we will only use the translation of closed modules when addressing the translation of system descriptions. We will do that by considering the closed module that is equivalent to the theory of a system description.

G.2 System Description

Let us consider a system description $\mathcal{D}$ consisting of a theory T and a structure description S and let us assume that T is equivalent to a closed module M. We translate $\mathcal{D}$ into an action description $AD(\mathcal{D})$ as follows:

1. Initialize $AD(\mathcal{D})$ as the action description $AD(M)$ (the translation of M).

2. Add to $AD(\mathcal{D})$ a module called S and add to S the statement:

$$\textbf{import } M;$$

3. For every statement of S of the form:

$$x \textbf{ in } c$$
$$[\ attr[(x_1, \ldots, x_n)]\ =\ x_{n+1}\ ]+$$

where $c \not\leq_{\mathcal{C}} actions$, add to the **objects** of module S:

$$x : c$$

and for each one of the assignments of values to attributes of x, add to the **axioms** of S:

$$attr(x, x_1, \ldots, x_n) = x_{n+1};$$

4. For every statement of S of the form:

$$x \textbf{ in } c$$
$$attr_1\ =\ x_1$$
$$\ldots,$$
$$attr_k\ =\ x_k$$

where $c \leq_{\mathcal{C}} actions$, add to module S:

$$\textbf{import } M\_c;$$
$$\alpha(c) \textbf{ is}$$
$$\textbf{case } upc(attr_1) = x_1\ \&\ \ldots\ \&\ upc(attr_k) = x_k\ :\ x;$$

(* For use in the next point, we introduce the notation $\delta(x)$ as follows:

$\delta(x) \quad =_{def} \quad \alpha(c)$ **is**

case $upc(attr_1) = x_1$ & $\ldots$ & $upc(attr_k) = x_k$: x;

)

5. For every module $M\_c_1\_\ldots\_c_n$ of $AD(M)$, and every possible combination of instances $x_1, \ldots, x_n$ of classes $c_1, \ldots, c_n$, respectively, defined by structure S, add to module S of $AD(M)$:

import $M\_c_1\_\ldots\_c_n$;
$\delta(x_1)$
$\ldots$
$\delta(x_n)$

6. Finally, apply the operation *upc* (upper-case) to variables and the operation *lwc* (lower-case) to constants.

Note that the module M is imported directly in S and is also imported indirectly in S via modules of the type M_c, if such modules are imported in S. This may result in having multiple copies of the same state constraint, for example.