A high-level action semantics for specifying and reasoning about dynamic systems is presented which supports both uncertain knowledge (taken as explicit indeterminism) and contradictory information (taken as implicit indeterminism). We start by developing an action description language for intentionally representing nondeterministic actions in dynamic systems. We then study the different possibilities of interpreting contradictory specifications of concurrent actions. We argue that the most reasonable interpretation which allows for exploiting as much information as possible, is to take such conflicts as implicit indeterminism. As the second major contribution, we present a calculus for our resulting action semantics based on the logic programming paradigm including negation-as-failure and equational theories. Soundness and completeness of this encoding wrt. the notion of entailment in our action language is proved by taking the completion semantics for equational logic programs with negation.

© Elsevier Science Inc., 1997

1. INTRODUCTION

Uncertainty is a general challenge which comes with different faces. If an agent reasons about a given representation of a dynamic system, he or she might be uncertain about the effects of particular actions; one possible reason for such an uncertainty is the designer of this representation has intentionally specified these actions so as to having nondeterministic effects. There are good reasons for doing this: The designer might not know the exact causal relationship between the ac-
tion and the observed effects, or the action might be chaotic, etc. In the very first part of this paper, we develop a high-level representation language and semantics which allows for intentionally specifying actions with nondeterministic, randomized effects.

Our language is based on the *Action Description Language* \( \mathcal{A} \) [12], which is appealing because of the simple, elegant and natural way in which the effects of actions are described. A formal introduction to this language can be found in Section 2, and our extended language dealing with explicit indeterminism, which we call \( \mathcal{A}_I \), is then developed in Section 3.

Aside from being faced with explicitly represented indeterminism, a reasoning agent might also be uncertain about a given specification when the latter turns out to be contradictory. Intelligent beings are most often able to evaluate contradictory information to an appropriate extent. For instance, imagine yourself asking two passers-by for the shortest way to the train station. The first one answers: "Turn right, and you will get there in five minutes," while the second one answers: "Turn right, and you will get there in ten minutes." Reasoning about these answers, you find out that they are contradictory; the provided information is inconsistent and, hence, cannot be true. However, since both passers-by are in agreement about their recommendation to turn right, you would assume this part of the information to be sound; you are just left with uncertainty about the time it takes to reach the station.

One should be aware of the difference between uncertain information explicitly stated as such, like "you will arrive in five or in ten minutes," and contradictory information like the two answers above. Contradictory information cannot be true, so it has to be interpreted appropriately if nonetheless some benefit is to be derived from it. Of course, any such interpretation has to be carefully selected in view of the application at hand. When machines are used to reason about complex domains, it is highly likely that an inconsistency occurs in the corresponding formal specification; e.g., we know from Software Engineering that in general formalizations of non-trivial scenarios are incorrect. Therefore, if a reasoning system detects an inconsistency in the information that has been provided, this only confirms what had to be assumed anyway. But it still remains to be decided how this system should act in such a situation.

A typical problem in the context of reasoning about actions where contradictory specifications are to be expected is the concurrent execution of actions. Most complex dynamic systems include some kind of concurrency, which is why the ability of describing simultaneous actions is of central interest in AI. For instance, to open a door locked by an electric door opener and autonomous robot has to press a button and to push the door concurrently. Thus, knowing the effects of the separate execution of these actions only is not sufficient to be able to open the door. Since it is of course impractical to define the effects of the concurrent execution of each possible combination of actions explicitly, it is necessary to infer these effects from the various descriptions of the individual actions that are involved. In certain cases, some of these descriptions may however propose contradictions. The crucial question then is how to interpret such contradictions.

This question will be discussed in Section 4. To this end, we use a recent extension of the *Action Description Language* \( \mathcal{A} \) which is called \( \mathcal{A}_C \) and supports representing and reasoning about concurrent actions [5]. In Section 4.1, we discuss
different explicit methods which enable the designer of a representation to prevent the aforementioned conflicts by providing more specific information regarding particular combinations of concurrently executed actions. We will argue that $\mathcal{A}_c$ uses the most expressive way and, hence, is most suitable as a basis for our further discussions. The language $\mathcal{A}_c$ is recapitulated in Section 4.2.

In Section 4.3, we then examine the various possibilities to interpret contradictory inferences caused by combining action descriptions. Suggesting a different point of view than the one implicitly underlying $\mathcal{A}_c$, we present a new language called $\mathcal{A}_{NC}$. Thereby we combine our aforementioned development regarding explicit indeterminism, the language $\mathcal{A}_N$, with $\mathcal{A}_c$, and we define a new way of successfully reasoning about inconsistent specifications of concurrently executed actions.¹ The crucial idea is to interpret any such contradiction as implicit indeterminism. To this end, we consider uncertain the pieces of information which cause the contradiction, while all effects on which the involved action descriptions agree are assumed to occur as specified. In so doing, our language enables us to still infer reasonable information from contradictory descriptions, whereas these inferences are neither possible in $\mathcal{A}$ nor in $\mathcal{A}_c$.

As the second major contribution of this paper, in Section 5 we present a sound and complete translation from domains specified in our high-level action language $\mathcal{A}_{NC}$ into logic programs. Our translation follows an approach originally introduced in [17], which is based on the reification of entire state descriptions by formally treating them as terms. In contrast to situation calculus [28, 30], where situation terms are abstract objects, the former approach employs state terms consisting of an explicit collection of those fluents which hold in the situation being represented. Executing actions is then modeled by manipulating these collections of fluents, which is why we call the underlying method fluent calculus (or FE', for short).² An equational logic program suitable for encoding $\mathcal{A}_{NC}$, consequently named FE'$_{NC}$, is developed in Section 5.1. In Section 5.2, we analyze the semantics of this program given by its completion, and in Section 5.3 we prove soundness and completeness wrt. the high-level action semantics given by $\mathcal{A}_{NC}$. Finally, in Section 5.4 we discuss an adequate computation mechanism for our program, namely, SLDENF-resolution [34, 39], which is based on SLD-resolution but with the standard unification procedure replaced by a special equality unification algorithm (E) and negation-as-failure used to treat negative subgoals (NF).

2. DESCRIBING SIMPLE ACTION SCENARIOS

To begin with, we briefly review the concepts underlying the Action Description language $\mathcal{A}$ as defined in [12].

Definition 2.1. A domain description $D$ consists of two disjoint and nonempty sets of symbols $F_D$ and $A_D$ called fluent names and unit actions, respectively. A fluent literal is a fluent name or its negation, the latter of which is denoted by $\neg f$.

¹$\mathcal{A}_N$ denotes an "action formalism supporting nondeterministic (N) actions based on the Action Description Language ($\mathcal{A}$)," and $\mathcal{A}_{NC}$ is as $\mathcal{A}_N$ but in addition supports concurrent actions (C) and conflict solving (C).

²We are grateful to Stuart Russell for suggesting this name.
Furthermore, $D$ consists of a set $V_D$ of \textit{value propositions} ($v$-propositions, for short), which are expressions of the form

$$l \text{ after } [a_1, \ldots, a_m], \quad (2.1)$$

where $a_1, \ldots, a_m$ ($m \geq 0$) are unit actions and $l$ is a fluent literal.

Finally, $D$ includes a set $E_D$ of \textit{effect propositions} ($e$-propositions, for short), which are expressions of the form

$$a \text{ causes } l \text{ if } c_1, \ldots, c_n, \quad (2.2)$$

where $a$ is a unit action and $l$ as well as $c_1, \ldots, c_n$ ($n \geq 0$) are fluent literals.

A $v$-proposition (2.1) should be read as: If the sequence of unit actions $[a_1, \ldots, a_m]$ were performed in the initial state then $l$ would hold in the resulting state. In case $m = 0$, (2.1) is usually written initially $l$. An $e$-proposition (2.2) should be read as: Executing unit action $a$ causes $l$ to hold in the resulting state provided the conditions $c_1, \ldots, c_n$ hold in the current state. In case $n = 0$, (2.2) is usually written $a \text{ causes } l$.

\textbf{Example 2.1.} We model the \textit{Yale Shooting} domain [15] using the fluent names $F_D = \{\text{loaded, alive}\}$ denoting the state of a gun and a turkey, respectively. The effects of the unit actions $A_D = \{\text{load, wait, shoot}\}$ are specified by these three $e$-propositions:

$$\begin{align*}
\text{load} & \text{ causes } \text{loaded} \\
\text{shoot} & \text{ causes } \text{loaded} \\
\text{shoot} & \text{ causes } \overline{\text{alive}} \text{ if } \text{loaded}.
\end{align*} \tag{2.3}$$

In words, loading the gun causes it to be loaded, shooting with the gun causes it to become unloaded and also shoots the turkey provided the gun was loaded. Waiting is assumed to have no effects at all. On this basis, the following two $v$-propositions encode the \textit{Stanford Murder Mystery} instance [3] of the Yale Shooting domain:

$$\begin{align*}
\text{initially } \overline{\text{alive}} \\
\text{alive} \text{ after } [\text{wait, shoot}] \tag{2.4}
\end{align*}$$

In words, the turkey is alive at the beginning but not after executing \textit{wait} followed by \textit{shoot}.

Given a domain description $D$, a \textit{state} $\sigma$ is simply a subset of the set of fluent names $F_D$. For any $f \in F_D$, if $f \in \sigma$ then $f$ is said to \textit{hold} in $\sigma$, otherwise $\overline{f}$ holds. E.g., both \textit{alive} and \textit{loaded} hold in the state $\sigma = \{\text{alive}\}$.

The given effect propositions implicitly determine the causal behavior of the dynamic system being modeled:

\textbf{Definition 2.2.} Let $D$ be a domain description in $\mathcal{A}$. Furthermore, let $a \in A_D$ be a unit action, $l \in F_D \cup \{f \mid f \in F_D\}$ a fluent literal, and $\sigma \subseteq F_D$ a state. Then we say that $a \text{ causes } l \text{ in } \sigma$ iff $E_D$ contains an $e$-proposition $a \text{ causes } l \text{ if } c_1, \ldots, c_n$.
such that each of $c_1, \ldots, c_n$ holds in $\sigma$. Let

$$B_f(a, \sigma) := \{ f \in F_D | a \text{ causes } f \text{ in } \sigma \}$$

$$\overline{B_f}(a, \sigma) := \{ f \in F_D | a \text{ causes } \bar{f} \text{ in } \sigma \}$$

(2.5)

and let $\Phi$ be a mapping from pairs consisting of a unit action and a state into the set of states, that is, $\Phi : A \times 2^{F_0} \rightarrow 2^{F_0}$. Then $\Phi$ is a transition function for $D$ iff, for each $a \in A$ and $\sigma \subseteq F_D$,

1. $B_f(a, \sigma) \cap \overline{B_f}(a, \sigma) = \{ \}$

2. $\Phi(a, \sigma) = (\sigma \setminus \overline{B_f}(a, \sigma)) \cup B_f(a, \sigma)$.

In words, $B_f(a, \sigma)$ contains all fluent names that some e-proposition claims to become true when executing $a$ in $\sigma$, while $\overline{B_f}(a, \sigma)$ contains all fluent names that become false. Apart from considering new truth values for these affected fluent names, the general assumption of persistence is applied to all remaining fluents. In case $\overline{B_f}(a, \sigma)$ and $\overline{B_f}(a, \sigma)$ share one or more elements, no transition function exists and the entire domain is considered inconsistent.

Example 2.1 (continued). Given the e-propositions in (2.3), we have, for instance, $B_f(\text{load}, \{\text{alive}\}) = \{\text{loaded}\}$ and $\overline{B_f}(\text{load}, \{\text{alive}\}) = \{\}$. Hence $\Phi(\text{load}, \{\text{alive}\}) = \{\text{alive, loaded}\}$. The following definition provides a complete description of the transition function $\Phi$ for the Yale Shooting scenario following Definition 2.2:

$$\Phi(\text{wait}, \sigma) = \sigma$$

$$\Phi(\text{load}, \sigma) = \sigma \cup \{\text{loaded}\}$$

$$\Phi(\text{shoot}, \sigma) = \begin{cases} 
\sigma \setminus \{\text{loaded, alive}\}, & \text{if } \text{loaded} \in \sigma \\
\sigma, & \text{otherwise}.
\end{cases}$$

(2.6)

Based on the concept of transition, the semantics for $\mathcal{A}$ provides a notion of entailment given a domain specification:

Definition 2.3. Let $D$ be a domain description in $\mathcal{A}$. A structure $M$ is a pair $(\sigma_0, \Phi)$ where $\sigma_0 \subseteq F_D$—called the initial state—and $\Phi : A \times 2^{F_0} \rightarrow 2^{F_0}$. Let $M[a_1, \ldots, a_m]$ be an abbreviation for $\Phi(a_m, \Phi(a_{m-1}, \ldots, \Phi(a_1, \sigma_0)) \ldots)$, then a $\nu$-proposition $f$ after $[a_1, \ldots, a_m]$ is true in structure $M$ iff $f$ holds in the state $M[a_1, \ldots, a_m]$.

A structure $M = (\sigma_0, \Phi)$ is then called a model of $D$ iff $\Phi$ is a transition function for $D$ and every $\nu$-proposition in $V_D$ is true in $M$. A $\nu$-proposition $\nu$ is entailed by $D$ iff $\nu$ is true in every model of $D$.

Example 2.1 (continued). Let $\Phi$ be as in (2.6). Then both the two structures $M_1 = (\{\text{alive}\}, \Phi)$ and $M_2 = (\{\text{alive, loaded}\}, \Phi)$ are models of the first $\nu$-proposition in (2.4). Since $M_1[\text{wait, shoot}] = \{\text{alive}\}$ our second $\nu$-proposition in (2.4) is not true in $M_1$, whereas $M_2[\text{wait, shoot}] = \{\}$ shows that $M_2$ is a model for our entire example domain. As a matter of fact, $M_2$ is the only model. Therefore, since $M_2[\text{wait, shoot}] = \{\text{alive, loaded}\}$, our domain description entails, among others, the $\nu$-proposition initially loaded. The latter can be taken as a solution to the Stanford Murder Mystery.
3. NONDETERMINISTIC ACTIONS: THE LANGUAGE $\mathcal{A}_N$

A basic assumption underlying $\mathcal{A}$ is that the effects of an action are always completely known and deterministic. As argued in the introduction, however, one cannot adhere to this idealistic view of the real world in general since it is impossible to refine descriptions of the world until the effects of an arbitrary action can always be fully determined. The ability of humans to handle uncertainty, indeterminism, or surprising effects, etc. very flexibly contrasts with the necessity of completely determining the effects of actions. This insight has recently led to several proposals for integrating nondeterministic actions into existing frameworks, e.g. [7, 33, 23, 21, 4, 26]. In this section we extend the action description language $\mathcal{A}$ so that indeterminism can be explicitly represented; we denote the resulting dialect by $\mathcal{A}_N$.

To begin with, expressing nondeterministic actions requires an extended notion of effect propositions:

**Definition 3.1.** Let $F_D$ be a set of fluents and $A_D$ a set of unit actions. An effect proposition is either of the form

$$a \text{ causes } e \text{ if } c_1, \ldots, c_n,$$

(in what follows called *strict* $e$-proposition), or of the form

$$a \text{ alternatively causes } e_1, \ldots, e_m \text{ if } c_1, \ldots, c_n \quad (3.1)$$

(in what follows called *alternative* $e$-proposition), where $a \in A_D$ and $e, e_1, \ldots, e_m$ as well as $c_1, \ldots, c_n$ are fluent literals ($m, n \geq 0$).

**Example 3.1.** We marginally extend the Russian Turkey scenario as formalized in [33] and take this as the running example of this section. To this end, the set of unit actions used in Example 2.1 is augmented by an action called *spin*. The intended meaning is that spinning causes the gun to become randomly loaded or unloaded regardless of its state before, and if it becomes unloaded then the person operating it becomes nervous. \(^3\) The latter is represented by the additional fluent name *nervous*. The effects of the new unit action can be specified in $\mathcal{A}_N$ using these two alternative $e$-propositions:

$$\text{spin } \text{ alternatively causes } \text{ loaded} \quad (3.2)$$

$$\text{spin } \text{ alternatively causes } \text{ loaded, nervous}. \quad (3.3)$$

The intended meaning of a set of alternative $e$-proposition is as follows: If $a$ is a unit action and $\sigma$ a state then let

$$\begin{cases}
\left\{ a \text{ alternatively causes } E_1 \text{ if } C_1 \right\} \\
\left\{ a \text{ alternatively causes } E_2 \text{ if } C_2 \right\} \\
\vdots
\end{cases} \quad (3.3)$$

\(^3\)For sake of simplicity, we assume the gun's cylinder consist of two chambers, exactly one of which contains a bullet. Furthermore, executing the action *load* should be interpreted as manually selecting the chamber that is loaded.
be the (not necessarily finite) set of all alternative e-propositions describing \( a \) such that \( C_1, C_2, \ldots \) simultaneously hold in \( \sigma \), where each of \( E_1, C_1, E_2, C_2, \ldots \) is a finite (possibly empty) sequence of fluent literals. Now, if \( a \) is executed in \( \sigma \) then nondeterministically one \( E \in \{ E_1, E_2, \ldots \} \) becomes true in the resulting state (that is, all fluent literals \( e_1, \ldots, e_m = E \) hold in this state).\(^4\) For instance, if \( \text{spin} \) is executed in the state \( \{ \text{alive}, \text{loaded} \} \) then, following (3.2), either \( \text{loaded} \) or else both \( \text{loaded} \) and \( \text{nervous} \) will be true afterwards. The two possible resulting states are therefore \( \{ \text{alive}, \text{loaded} \} \) and \( \{ \text{alive}, \text{nervous} \} \).

Recall that a set of e-propositions in the language \( \mathcal{A} \) determines a unique transition function \( \Phi \). Now, however, the possibility of alternative effects forces a redefinition of this notion. At first glance one might suggest for allowing the existence of several different transition functions, each of which models one of the various alternative effects of an action. Any particular model \( (\sigma_0, \Phi) \) would then have to select among these possibilities. E.g., given the e-propositions (3.2), \( \Phi \) could be designed such that either \( \Phi(\text{spin}, \sigma) = \sigma \cup \{ \text{loaded} \} \) or \( \Phi(\text{spin}, \sigma) = (\sigma \setminus \{ \text{loaded} \}) \cup \{ \text{nervous} \} \), separately for each \( \sigma \). However, if \( \Phi \) is one of these transition functions in a particular model then the result of spinning the gun will be fixed forever regarding a particular state; e.g., it would be impossible to find a model where initially \( \text{loaded} \), \( \text{loaded} \) after \( \text{[spin]} \), and \( \text{loaded} \) after \( \text{[spin, spin]} \) are simultaneously true. This is of course unintended.

For this reason, we adapt a standard concept for dealing with multiple possible successor states by dropping the idea of \( \Phi \) being a function and instead using the notion of \( \Phi \) as a \textit{relation} between a pair of states and a unit action name such that \( (\sigma, a, \sigma') \in \Phi \) whenever the application of \( a \) to \( \sigma \) might yield \( \sigma' \). The following formal notion of transition in \( \mathcal{A} \) reflects this intuition:

\textbf{Definition 3.2.} Let \( D \) be a domain description in \( \mathcal{A} \) and let \( \Phi \subseteq 2^{F_D} \times A_D \times 2^{F_D} \) be a relation. Then \( \Phi \) is a \textit{transition relation} for \( D \) iff the following condition is satisfied for each state \( \sigma \subseteq F_D \) and each unit action \( a \in A_D \):

Let

\[ \Pi_a^\sigma = \{ a \text{ alternatively causes } E \text{ if } C \in E_D \mid \text{ each fluent literal in } C \text{ holds in } \sigma \} \]

be the set of all alternative e-propositions in \( E_D \) with unit action name \( a \) and which are applicable in \( \sigma \). Then,

1. In case \( \Pi_a^\sigma = \{ \} \): We say that \( a \text{ causes } e \text{ in } \sigma \) iff \( E_D \) contains a strict e-proposition \( a \text{ causes } e \) if \( c_1, \ldots, c_n \) such that each of \( c_1, \ldots, c_n \) holds in \( \sigma \). Define

\[ B_f(a, \sigma) := \{ f \in F_D \mid a \text{ causes } f \text{ in } \sigma \} \]

\[ \overline{B}_f(a, \sigma) := \{ f \in F_D \mid a \text{ causes } \bar{f} \text{ in } \sigma \} \]

\(^4\) Hence, an alternative e-proposition (3.1) with \( m = 0 \) expresses the possibility that the execution of \( a \) has no effects at all aside from what is suggested by all applicable strict e-propositions.

\(^5\) Otherwise no transition relation for \( D \) exists.
then $B_f(a, \sigma) \cap \overline{B_f}(a, \sigma)$ must be empty, and we have $(\sigma, a, \sigma') \in \Phi$ iff 
$\sigma' = (\sigma \setminus \overline{B_f}(a, \sigma)) \cup B_f(a, \sigma)$.

2. In case $\Pi_a^\sigma \neq \emptyset$: For each $\lambda = a$ alternatively causes $E$ if $C$ in $\Pi_a^\sigma$ we say that $a$ causes a fluent literal $e$ wrt. $\lambda$ in $\sigma$ iff $e$ occurs in $E$ or $E_D$ contains a strict $e$-proposition $a$ causes $e$ if $c_1, \ldots, c_n$ such that each of $c_1, \ldots, c_n$ holds in $\sigma$. Define 

$$B_f(a, \lambda, \sigma) := \{ f \in F_D \mid a \text{ causes } f \text{ wrt. } \lambda \text{ in } \sigma \}$$

$$\overline{B_f}(a, \lambda, \sigma) := \{ f \in F_D \mid a \text{ causes } \bar{f} \text{ wrt. } \lambda \text{ in } \sigma \}$$

then $B_f(a, \lambda, \sigma) \cap \overline{B_f}(a, \lambda, \sigma)$ must be empty for each $\lambda \in \Pi_a^\sigma$, and we have $(\sigma, a, \sigma') \in \Phi$ iff there exists some $\lambda \in \Pi_a^\sigma$ such that $\sigma' = (\sigma \setminus \overline{B_f}(a, \lambda, \sigma)) \cup B_f(a, \lambda, \sigma)$.

In words, a possible successor state is constructed by accounting for each strict $e$-proposition; by selecting one $e$-proposition $\lambda \in \Pi_a^\sigma$ among the applicable alternatives; and by applying the persistence assumption to all remaining fluents.

**Example 3.1** (continued). Recall our Russian Turkey scenario. Let $\lambda_1$ and $\lambda_2$ denote the first and second, respectively, $e$-proposition in (3.2). Then we have $\Pi_{\text{spin}} = \{ \lambda_1, \lambda_2 \}$ for each state $\sigma$. From 

$$B_f(\text{spin}, \lambda_1, \sigma) = \{ \text{loaded} \}$$

$$\overline{B_f}(\text{spin}, \lambda_1, \sigma) = \{ \}$$

and 

$$B_f(\text{spin}, \lambda_2, \sigma) = \{ \text{nervous} \}$$

$$\overline{B_f}(\text{spin}, \lambda_2, \sigma) = \{ \text{loaded} \}$$

it follows that $(\sigma, \text{spin}, \sigma') \in \Phi$ iff $\sigma' = \sigma \cup \{ \text{loaded} \}$ or $\sigma' = (\sigma \setminus \{ \text{loaded} \}) \cup \{ \text{nervous} \}$, for each $\sigma$. The following definition provides a complete description of the transition relation $\Phi$ for the $e$-propositions (2.2) and (3.2) following Definition 3.2:

$$(\sigma, \text{spin}, \sigma') \in \Phi \text{ iff } \sigma' = \sigma \cup \{ \text{loaded} \} \text{ or } \sigma' = (\sigma \setminus \{ \text{loaded} \}) \cup \{ \text{nervous} \}$$

$$(\sigma, \text{wait}, \sigma') \in \Phi \text{ iff } \sigma' = \sigma$$

$$(\sigma, \text{load}, \sigma') \in \Phi \text{ iff } \sigma' = \sigma \cup \{ \text{loaded} \}$$

$$(\sigma, \text{shoot}, \sigma') \in \Phi \text{ iff } \sigma' = \begin{cases} \sigma \setminus \{ \text{loaded}, \text{alive} \}, & \text{if loaded} \in \sigma \\ \sigma, & \text{otherwise}. \end{cases}$$

Having defined the notion of transition, we now concentrate on defining the concept of a model in $\mathcal{M}$. The purpose of models is, in general, to provide a possible view of the real world according to given knowledge. In $\mathcal{M}$, where no indeterministic and randomized effects are allowed, models differ only in their initial state once a transition function is fixed. Now, however, any model needs to

---

$^6$If not then, as before, no transition relation for $D$ exists.
Additionally state which particular effect occurs whenever alternatives exist. An additional component for each model, namely, a function \( \varphi \), shall serve this purpose. More precisely, \( \varphi \) maps action sequences \([a_1, \ldots, a_n]\) to states, stating that the actual outcome of applying \([a_1, \ldots, a_n]\) to the initial state in the model at hand would be \( \varphi([a_1, \ldots, a_n]) \). For instance, if the initial state is known to be \{alive\} and we are interested in the consequences of executing the sequence of unit actions \([load, spin, shoot]\) then the set of models of this domain can be divided into two classes: Either the gun remains loaded after spinning, or it becomes unloaded. This is formally captured by requiring that each model satisfy either \( \varphi([load, spin]) = \{alive, loaded\} \) or else \( \varphi([load, spin]) = \{alive, nervous\} \). Suppose that in addition we observe that the turkey is as lively as before after loading, spinning, and shooting then no model of the former class can explain this. Thus, it is reasonable to conclude that the gun was necessarily unloaded and the hunter became nervous after \([load, spin]\). This is illustrated in Figure 1. Note that we would be unable to obtain this conclusion without "recording," by using \( \varphi \), the actual outcome of the nondeterministic action, \( spin \). The formal definition of models for domain descriptions in \( \mathcal{N}_N \) is as follows:

**Definition 3.3.** Let \( D \) be a domain description in \( \mathcal{N}_N \). A **structure** is a triple \( (\sigma_0, \Phi, \varphi) \) where \( \sigma_0 \subseteq F_D \), \( \Phi \subseteq 2^{F_0} \times A_D \times 2^{F_0} \) and \( \varphi : A_D^* \to 2^{F_0} \) such that\(^7\)

1. \( \varphi([\ ])=\sigma_0 \) and
2. \( (\varphi([a_1, \ldots, a_n]), a_{n+1}, \varphi([a_1, \ldots, a_n, a_{n+1}])) \in \Phi \) for each sequence of unit actions \( a_1, \ldots, a_n, a_{n+1} \) \((n \geq 0)\).

A v-proposition \( l \) after \([a_1, \ldots, a_n] \) \((n \geq 0)\) is **true** in a structure \( (\sigma_0, \Phi, \varphi) \) iff \( l \) holds in \( \varphi([a_1, \ldots, a_n]) \). The structure is a **model** of \( D \) iff \( \Phi \) is a transition relation for \( D \) and all v-propositions in \( V_D \) are true. A v-proposition \( \nu \) is **entailed** by \( D \) iff \( \nu \) is true in every model of \( D \).

In words, the third component of a structure, viz. \( \varphi \), both respects the transition relation \( \Phi \) and is now used to validate the given v-propositions. We call a domain description in \( \mathcal{N}_N \) **consistent** if it has a model.

---

\(^7\)By \( A_D^* \) we denote the set of all finite lists, including the empty one, whose elements are chosen from \( A_D \).
Example 3.1 (continued). A structure \((\sigma_0, \Phi, \varphi)\) is a model of (2.2) and (3.2) along with the two \(v\)-propositions

\[
\begin{align*}
\text{initially} & \quad \text{alive} \\
\text{alive after} & \quad \text{[load, spin, shoot]}
\end{align*}
\]

iff transition relation \(\Phi\) is as in (3.4), \(\varphi\) satisfies clauses 1 and 2 of Definition 3.3, and \(\text{alive} \in \sigma_0 = \varphi([])\) as well as \(\text{alive} \in \varphi([\text{load}, \text{spin}, \text{shoot}])\). Let \((\sigma_0, \Phi, \varphi)\) be any of these models, then \(\text{loaded} \in \varphi([\text{load}, \text{spin}])\); hence, \(\text{loaded}\) after \([\text{load}, \text{spin}]\) is entailed (cf. Figure 1).

Like \(\mathcal{A}\), our extended language \(\mathcal{A}_N\) supports reasoning about so-called counterfactual action sequences due to the fact that the model component \(\varphi\) is defined for any sequence of unit actions. To illustrate this, let us consider the following extension of Example 3.1, motivated by a scene in a Pierre Richard movie [31]. An additional fluent name, \(\text{broken}\), which describes the state of a vase. Furthermore, the action \(\text{shoot}\) is replaced by the unit actions \(\text{shoot-at-pierre}\) and \(\text{shoot-at-vase}\), respectively, along with these four \(e\)-propositions:

\[
\begin{align*}
\text{shoot-at-pierre} & \quad \text{causes} \quad \text{loaded} \\
\text{shoot-at-pierre} & \quad \text{causes} \quad \text{alive if} \quad \text{loaded} \\
\text{shoot-at-vase} & \quad \text{causes} \quad \text{loaded} \\
\text{shoot-at-vase} & \quad \text{causes} \quad \text{broken if} \quad \text{loaded}.
\end{align*}
\]

Now, suppose given the \(v\)-propositions

\[
\begin{align*}
\text{initially} & \quad \text{alive} \\
\text{initially} & \quad \text{broken} \\
\text{broken} & \quad \text{after} \quad \text{[spin, shoot-at-vase]}.
\end{align*}
\]

According to the \(e\)-propositions in (3.6), any model \((\sigma_0, \Phi, \varphi)\) must satisfy \(\text{loaded} \in \varphi([\text{spin}])\) since otherwise the vase could not have been destroyed. Thus, it is plausible to conclude that had we shot at Pierre instead then he would not have survived this. Definition 3.3 supports this conclusion formally: The reader is invited to verify that the domain consisting of \(e\)-propositions (3.2) and (3.6) plus the above \(v\)-propositions, (3.7), entails

\[
\begin{align*}
\text{alive after} & \quad \text{[spin, shoot-at-pierre]}.
\end{align*}
\]

Since the two action sequences used respectively in this \(v\)-proposition and in (3.7) are incompatible, this example requires reasoning about counterfactuals.

4. CONCURRENT ACTIONS AND SOLVING CONFLICTS

Since the problems we address in this section become more apparent in the context of the simultaneous execution of actions, we first discuss different ways of inter-
interpreting domain descriptions involving concurrency. To illustrate our exposition, we use the terms of $\mathcal{A}$ (and, later on, $\mathcal{A}_c$); nevertheless, the differences we identify provide a classification of other languages describing concurrent actions as well.

4.1. Explicit Information about Concurrent Execution

Suppose a rather complex description of a part of the world has to be constructed, where arbitrary unit actions may be executed concurrently. Because of the combinatorial explosion it is obviously impractical to describe the effects of all possible combinations of unit actions. It is therefore necessary to infer the effects of compound actions from the descriptions given separately for the various sub-actions involved. Combining these action descriptions may however yield a contradiction among their effects. In terms of the Action Description Language $\mathcal{A}$ this amounts to having $\bigcup_a B_f(a, \sigma) \cap \bigcup_a B_f(a, \sigma) = \emptyset$ (cf. Definition 2.2), where $\bigcup_a a$ contains all unit actions to be executed concurrently.

There are several ways of dealing with and inferring the effects of a compound action from descriptions of the involved unit actions which propose contradictory effects. Languages describing concurrent actions can therefore be classified according to the explicit and implicit methods, respectively, they use to draw these conclusions.

Explicit methods provide further information as to additional effects of certain compound actions; they are also used to state the difference between the actual effects of a concurrent execution of several actions and the effects of these unit actions when executed alone. In terms of the Action Description Language, additional e-propositions may

1. add a fluent to $B_f$ or $\overline{B_f}$: obviously, the set $B_f \cap \overline{B_f}$ will remain nonempty in case of a conflict, hence no conflicts will be solved;
2. remove a fluent from $B_f$ or $\overline{B_f}$: this allows the removal of predicted conflicts, but not the introduction of effects not mentioned by the unit action descriptions (the approach in [24] uses this method by "cancelling" effects in specific cases);
3. add or remove a fluent from $B_f$ or $\overline{B_f}$: this enables one to arbitrarily modify $B_f$ and $\overline{B_f}$ (used, for instance, in $\mathcal{A}_c$, our language $\mathcal{A}_{NCC}$, and in State Event Logic [14]).

Since an extension of $\mathcal{A}$ to concurrent actions, called $\mathcal{A}_c$, has recently been introduced [5], which uses the latter, most powerful method for stating differing effects of actions as regards their concurrent execution, we use this approach to illustrate our following discussion and adopt it when extending our language $\mathcal{A}_N$ to concurrency.

---

9This problem might of course occur even without concurrency involved, viz. if several descriptions, i.e., e-propositions, of the same unit action are used to infer the effects of this single action. If this inference yields a contradiction, the semantics of $\mathcal{A}$ and $\mathcal{A}_N$, for instance, define the whole domain description to be inconsistent as it does not admit a proper notion of transition.

10To be precise, neither $\mathcal{A}_c$ nor $\mathcal{A}_{NCC}$ allows for simply removing an element from one of these two sets via an additional e-proposition. Rather, new, more specific e-propositions may shift a fluent name from $B_f$ to $\overline{B_f}$ or vice versa, which, however, enables one to model any effect obtained by mere removal.
4.2. The Language $\mathcal{A}_C$

We briefly review the concepts underlying the language $\mathcal{A}_C$ as defined in [5] by pointing out the corresponding extensions of $\mathcal{A}$. In either e- or v-propositions of a domain description, actions are now non-empty, finite subsets of the given set of unit actions $A_D$, with the intended meaning that all of the elements are executed concurrently. These actions are also called compound actions to distinguish them from the unit actions.

Example 4.1. Suppose you can open a door by running into it if at the same time you activate the electric door opener; otherwise, you will hurt yourself by running into the door. A dog sleeping beside the door will wake up when the door opener is activated. You can close the door by pulling it. To formalize this scenario in $\mathcal{A}_C$, we take the two sets $A_D = \{ activate, pull, run\_into \}$ and $F_D = \{ open, sleeps, hurt \}$. Suppose that the initial state be partially described by the v-proposition initially sleeps. The effects of the actions can be specified by these five e-propositions:

\[
\begin{align*}
\{ activate \} & \text{ causes } sleeps \\
\{ run\_into \} & \text{ causes } hurt \quad \text{if } open \\
\{ pull \} & \text{ causes } open \\
\{ activate, run\_into \} & \text{ causes } open \\
\{ activate, run\_into \} & \text{ causes } hurt \quad \text{if } hurt.
\end{align*}
\]

Informally, the last e-proposition is needed to limit the application of the second one; this way of restricting applicability of (less specific) e-propositions is called overruling an e-proposition. Let $D_3$ denote the domain description given by these propositions.

The concept of overruling more general action descriptions by more specific ones is formalized by this modification of Definition 2.2.\(11\) If $a$ is a compound action, $l$ a fluent literal and $\sigma$ a state, then we say that $a$ causes $l$ in $\sigma$ iff there is an action $b$ such that $a$ causes $l$ by $b$ in $\sigma$. We say that $a$ causes $l$ by $b$ in $\sigma$ iff

1. $b \subseteq a$;
2. there is an e-proposition $b$ causes $l$ if $c_1, \ldots, c_n$ such that each of $c_1, \ldots, c_n$ holds in $\sigma$; and
3. there is no action $c$ such that $b \subset c$ and $a$ causes $l$ by $c$ in $\sigma$.

If conditions 1 and 2 but not condition 3 hold then the e-proposition in clause 2 is said to be overruled.

Now, if, based on this extended notion, $B_f(a, \sigma)$ and $\overline{B}_f(a, \sigma)$ are defined accordingly (cf. Definition 2.2 but with $a$ being any compound action) and share elements then the corresponding transition function $\Phi$ is taken to be undefined for the argument $(a, \sigma)$; otherwise, $\Phi(a, \sigma) = (\sigma \setminus \overline{B}_f(a, \sigma)) \cup B_f(a, \sigma)$, as before.

\(11\)The following description differs slightly from the definition given in [5], which is circular; we assume that ours is what the authors actually intended.
Example 4.1 (continued). The transition function determined by the e-propositions in our domain description $D_3$, viz. (4.1), is defined as follows. Let $\sigma$ be an arbitrary state then

- $\Phi(\emptyset, \sigma) = \sigma$
- $\Phi(\{\text{run\_into}\}, \sigma) = \sigma$, if $\text{open} \in \sigma$
- $\Phi(\{\text{run\_into}\}, \sigma) = \sigma \cup \{\text{hurt}\}$, if $\text{open} \notin \sigma$
- $\Phi(\{\text{pull}\}, \sigma) = \sigma \setminus \{\text{open}\}$
- $\Phi(\{\text{activate}\}, \sigma) = \sigma \setminus \{\text{sleeps}\}$
- $\Phi(\{\text{activate}, \text{pull}\}, \sigma) = \sigma \setminus \{\text{sleeps, open}\}$
- $\Phi(\{\text{run\_into}, \text{pull}\}, \sigma) = \sigma \setminus \{\text{open}\}$, if $\text{open} \in \sigma$
- $\Phi(\{\text{run\_into}, \text{pull}\}, \sigma) = \sigma \cup \{\text{hurt}\}$, if $\text{open} \notin \sigma$
- $\Phi(\{\text{activate}, \text{run\_into}\}, \sigma) = (\sigma \setminus \{\text{sleeps}\}) \cup \{\text{open}\}$
- $\Phi(\{\text{activate, run\_into, pull}\}, \sigma)$ is undefined.

Function value $\Phi(a, \sigma)$ being undefined for $a = \{\text{activate, run\_into, pull}\}$ and each state $\sigma$ is due to $B_f(a, \sigma) \cap \overline{B_f}(a, \sigma) = \{\text{open}\}$.

Now, following an appropriate adaptation of Definition 2.3 to domain descriptions in $\mathcal{A}_c$, $D_3$ admits four models, each of which satisfies $V_{D_3} = \{\text{initially sleeps}\}$, viz.

$$(\{\text{sleeps}\}, \Phi) \quad (\{\text{open, sleeps}\}, \Phi)$$

$$(\{\text{sleeps, hurt}\}, \Phi) \quad (\{\text{open, sleeps, hurt}\}, \Phi).$$

If, for instance, the v-proposition $\text{hurt}$ after $\{\text{run\_into}\}$ is added to $D_3$ then the only remaining model is $(\{\text{open, sleeps}\}, \Phi)$ since for all other structures in (4.2) we find that $\text{hurt} \in \Phi(\{\text{run\_into}\}, \sigma)$. Hence, for example, the v-proposition initially $\text{open}$ is entailed by this extended domain.

Note that our example domain can be modeled only by allowing both for addition and for removal of elements to and from $B_f$ or $\overline{B_f}$ (cf. Section 4.1):

Example 4.1 (continued). Let $\sigma$ be some state in our example domain. The e-proposition $\{\text{activate, run\_into}\}$ causes $\text{open}$ adds fluent name $\text{open}$ to the set $B_f(\{\text{activate, run\_into}\}, \sigma)$,\footnote{Note that fluent name $\text{open}$ is not mentioned by either of the 'unit' action descriptions $\{\text{activate}\}$ causes $\text{sleeps}$ and $\{\text{run\_into}\}$ causes $\text{hurt}$ if $\text{open}$.} while the e-proposition $\{\text{activate, run\_into}\}$ causes $\text{hurt}$ if $\text{hurt}$ removes fluent name $\text{hurt}$ from $B_f(\{\text{activate, run\_into}\}, \sigma)$ by overruling the e-proposition $\{\text{run\_into}\}$ causes $\text{hurt}$ if $\text{open}$.

4.3. Implicit Indeterminism: Interpreting Contradictions

After having introduced our basic concept for (explicitly) representing indeterminism in Section 3 and after having adopted an adequate formalism for representing concurrent actions, we are now able to discuss and propose a solution to the
problem of contradictory specifications of dynamic systems. Suppose the effects are not defined explicitly for all possible compound actions. In this case, as argued above, it may happen that certain actions are claimed to have contradictory effects.

From the point of view underlying $\mathcal{S}_c$, this indicates that these actions are not executable in the world. A typical example employed to justify this way of interpreting contradictions is the following: The door is open after it has been opened, and the door is not open after it has been closed; since a door cannot be open and not open at the same time, it is impossible to simultaneously open and close the door. An implicit assumption of this argument is that e-propositions do not describe concrete actions but assign (action-) names to the achievement of effects: "to open" means to do something that results in the door being open, likewise "to close" means to do something that results in the door being closed. Then, of course, it is impossible to have both simultaneously.

In contrast, our idea is that e-proposition describe concrete actions, and that all actions (that is, in the end, the mere decision to execute an action) can in principle be performed concurrently in any situation, sometimes, maybe, without being successful in achieving the intended effect. From this point of view, the occurrence of actions which are proposed to have contradictory effects when executed simultaneously only indicates that the descriptions of their effects are incorrect. As argued in the introduction, in many applications it is not desirable that an intelligent agent stops reasoning as soon as he/she detects an error in the description of the scenario he/she is acting in (which happens when the agent uses the semantics of $\mathcal{S}_c$ and observes a compound action—or is asked about the effects of it—which is defined to be impossible in the semantics).

To illustrate this, recall our example domain description $D_3$. The e-propositions describing the effects of the elements of $\{\text{activate, pull, run_into}\}$ claim both $\text{open}$ and $\overline{\text{open}}$. In such cases, depending on the chosen interpretation and the extent of certainty required, one has to regard as unreliable

1. the whole domain description (as in State-Event Logic [14]),
2. the whole situation,
3. all effects of the conflicting actions, or
4. the contradictory effects of the conflicting actions.

It is the latter, weakest condition which we propose in this paper. This follows the idea of still believing in every part of the information which does not cause the contradiction.

Example 4.1 (continued). It is of course conceivable that the door opener is activated, the door is pulled, and somebody runs into it at the same moment. The domain description $D_3$ proposes both $\text{open}$ and $\overline{\text{open}}$ to be an effect of the corresponding compound action, viz. $\{\text{activate, pull, run_into}\}$. Hence, $D_3$ is incomplete with respect to the world it describes. In fact, without further information we cannot say whether the door will be closed after executing this action. However, it is reasonable to assume the dog will not sleep afterwards since we know that $\{\text{activate}\}$ causes $\text{sleeps}$ and there is no proposition contradicting this. Using the semantics of $\mathcal{S}_c$ it cannot be inferred that $\text{sleeps}$ after $\{\text{activate, pull, run_into}\}$ since no successor state $\Phi(\{\text{activate, pull, run_into}\}, \sigma)$ exists (for any state $\sigma$).

In general, whenever a local inconsistency occurs, this causes the entire set of simultaneously executed actions to be contradictory. As an extreme case, imagine
an agent in Germany switching off a light and, concurrently, two agents in China executing the above action. Again, by \( \mathcal{A}c \) it cannot be inferred that the light is switched off in Germany because the description used proposes contradictory states of a door somewhere in China. Yet it seems rational to draw as many conclusions as reasonably possible about the resulting state instead of declaring it to be totally undefined. Preventing global inconsistency in case of local conflicts is our underlying intention here.

We therefore weaken the basic assumption which says that \( \Phi(a, \sigma) \) is undefined whenever the corresponding sets \( B_1(a, \sigma) \) and \( B_2(a, \sigma) \) share one or more elements, and instead we adopt the concept of nondeterminism developed in the preceding section. Informally speaking, if there are conflicts, that is, if the corresponding intersection \( B_1(a, \sigma) \cap B_2(a, \sigma) \) is not empty, then each combination of truth values of the controversial fluent names determines a possible successor state.

The following definition of the language \( \mathcal{A}_{\text{NC}} \) makes these ideas manifest. Syntactically, domain descriptions in our new language are specified using a combination of \( \mathcal{A}_n \) and \( \mathcal{A}_c \); that is, we take the syntax of \( \mathcal{A}_n \) and extend it by allowing to formalize compound actions. With the next definitions, we provide a formal concept of transition determined by a set of e-propositions following the above proposal. To ease readability, the overall definition is split into three parts:

**Definition 4.1.** Let \( D \) be a domain description in \( \mathcal{A}_{\text{NC}} \), and let \( a \subseteq A_D \) be some action and \( \sigma \subseteq F_D \) some state. Furthermore, for each \( b \subseteq a \) let

\[
\Pi_d = \{ b \text{ alternatively causes } E \mid C \subseteq E \}
\]

(4.3)

be the set of all alternative e-propositions in \( E_d \) for action \( b \) and which are applicable in \( \sigma \). Let then \( b_1, \ldots, b_m \) be a (possibly empty) ordered sequence of all actions \( b \subseteq a \) which satisfy \( \Pi_d \neq \{ \} \). A *selection for a wrt. \( \sigma \)*, written \( \Lambda_\sigma \), is a sequence of alternative e-propositions \( \lambda_{b_1}, \ldots, \lambda_{b_m} \) such that \( \lambda_{b_i} \in \Pi_d \) for each \( 1 \leq i \leq m \).

In words, a selection for an action \( a \) consists in exactly one applicable alternative e-proposition \( \lambda_b \), provided there exists any, for each sub-action \( b \subseteq a \).

**Definition 4.2.** Let \( D \) be a domain description, and let \( a \subseteq A_D \) be some action, \( \sigma \subseteq F_D \) some state, and \( e \in F_D \cup \{ \neg f \mid f \in F_D \} \) some fluent literal. Furthermore, let \( \Lambda_\sigma = \lambda_{b_1}, \ldots, \lambda_{b_m} \) be a selection for \( a \) wrt. \( \sigma \). Then we say that \( a \text{ causes } e \text{ wrt. } \Lambda_\sigma \) in \( \sigma \) if there is an action \( b \subseteq A_D \) such that \( a \text{ causes } e \text{ by } b \) (wrt. \( \Lambda_\sigma \) in \( \sigma \)). We say that \( a \text{ causes } e \text{ by } b \text{ wrt. } \Lambda_\sigma \text{ in } \sigma \) iff

1. \( b \subseteq a \);
2. at least one of these two conditions is satisfied:
   a. \( E_D \) contains a strict e-proposition \( b \text{ causes } e \text{ if } c_1, \ldots, c_n \text{ such that } \)
      each of \( c_1, \ldots, c_n \) holds in \( \sigma \), or
   b. \( \Pi_d \neq \{ \} \) and \( e \) occurs in \( E \), where \( \lambda_b = b \) alternatively causes \( E \)
      if \( C \subseteq \Pi_d \) is selected via \( \Lambda_\sigma \), and
3. there is no action \( c \) such that \( b \subseteq c \) and \( a \) causes \( \bar{c} \) by \( c \) wrt. \( \Lambda_a^o \) in \( \sigma \). We then define

\[
B_f(a, \Lambda_a^o, \sigma) := \{ f \in F_D | a \text{ causes } f \text{ wrt. } \Lambda_a^o \text{ in } \sigma \}
\]

\[
\overline{B_f}(a, \Lambda_a^o, \sigma) := \{ f \in F_D | a \text{ causes } \bar{f} \text{ wrt. } \Lambda_a^o \text{ in } \sigma \}.
\]

In words, a fluent literal is caused if it is among the strict or selected alternative effects, provided the corresponding e-proposition is not overruled.

**Definition 4.3.** Let \( D \) be a domain description in \( \mathcal{A}_{NCC} \), and let \( \Phi \subseteq 2^{F_D} \times 2^{A_D} \times 2^{F_D} \) be a relation. Then \( \Phi \) is a transition relation for \( D \) if for each two states \( \sigma, \sigma' \subseteq F_D \) and any action \( a \subseteq A_D \) we have \( (\sigma, a, \sigma') \in \Phi \) iff the following holds: There exists a selection \( \Lambda_a^o \) for \( a \) wrt. \( \sigma \) and a set \( B_f(a, \Lambda_a^o, \sigma) \subseteq \overline{B_f}(a, \Lambda_a^o, \sigma) \cap \overline{B_f}(a, \Lambda_a^o, \sigma) \) such that

\[
\sigma' = \left( \sigma \setminus \overline{B_f}(a, \Lambda_a^o, \sigma) \right) \cup \left( B_f(a, \Lambda_a^o, \sigma) \setminus \overline{B_f}(a, \Lambda_a^o, \sigma) \right).
\]

(4.4)

To summarize, a possible successor state is obtained by first making a random selection among the applicable alternative e-propositions, separately for each compound action \( b \subseteq a \). Afterwards, we proceed as in \( \mathcal{A}_C \) except in case conflicts occur, where we take any truth value distribution \( B_f(a, \Lambda_a^o, \sigma) \) among the disputed fluent names, i.e., \( B_f(a, \Lambda_a^o, \sigma) \cap \overline{B_f}(a, \Lambda_a^o, \sigma) \), when computing a possible \( \sigma' \) via (4.4).

Based on this concept of transition, the notion of model and entailment in \( \mathcal{A}_{NCC} \) are adopted from our language \( \mathcal{A}_N \) (cf. Definition 3.3):

**Definition 4.4.** Let \( D \) be a domain description in \( \mathcal{A}_{NCC} \). A structure is a triple \((\sigma_0, \Phi, \varphi)\) where \( \sigma_0 \subseteq F_D \), \( \Phi \subseteq 2^{F_D} \times 2^{A_D} \times 2^{F_D} \) and \( \varphi: (2^{A_D})^* \mapsto 2^{F_D} \) such that\(^{13}\)

1. \( \varphi([]) = \sigma_0 \) and
2. \( (\varphi([a_1, \ldots, a_n]), a_{n+1}, \varphi([a_1, \ldots, a_n, a_{n+1}])) \in \Phi \) for each sequence of actions
   \( a_1, \ldots, a_n, a_{n+1} \) \((n \geq 0)\).

A v-proposition \( l \) after \( [a_1, \ldots, a_n] \) \((n \geq 0)\) is true in a structure \((\sigma_0, \Phi, \varphi)\) iff \( l \) holds in \( \varphi([a_1, \ldots, a_n]) \). The structure is a model of \( D \) iff \( \Phi \) is a transition relation for \( D \) and all v-propositions in \( V_D \) are true. A v-proposition \( \nu \) is entailed by \( D \) iff \( \nu \) is true in every model of \( D \).

**Example 4.1.** (continued). If our domain description \( D_3 \) is augmented by either the v-proposition open after \( \{ \text{activate, pull, run_into} \} \) or the opposite proposition open after \( \{ \text{activate, pull, run_into} \} \) then both extended domains have models (with different functions \( \varphi \)) according to the semantics of \( \mathcal{A}_{NCC} \). On the other hand, if \( D_3 \) is augmented by sleeps after \( \{ \text{activate, pull, run_into} \} \) then there is no model wrt. \( \mathcal{A}_{NCC} \). Consequently, we can conclude, as intended, that our domain entails sleeps after \( \{ \text{activate, pull, run_into} \} \).

The reader might have noticed that \( \mathcal{A}_{NCC} \) does not distinguish between intentionally expressed nondeterminism of actions and our interpretation of inconsis-

\(^{13}\) The set \((2^{A_D})^*\) contains all finite lists whose elements are finite, non-empty subsets of \( A_D \).
tently specified actions. For instance, $D_3$ could be augmented by the e-propositions $(\text{activate})$ causes $bark$ and $(\text{activate})$ causes $bark$ for describing that the dog possibly starts or stops barking when the door opener is activated. The two alternative e-propositions $(\text{activate})$ alternatively causes $bark$ and $(\text{activate})$ alternatively causes $bark$ together serve the identical purpose. In fact, for someone reasoning about a domain description it makes no difference whether the designer of this domain description was aware of the uncertainty of the described effects or not.

5. TRANSLATING $\mathcal{A}_{\text{NCC}}$ INTO $\mathcal{F}_{\text{NCC}}$

In the second part of this paper, we show how domain descriptions and the notions of transition and entailment in our new language $\mathcal{A}_{\text{NCC}}$ may be encoded as logic programs. While in the preceding sections the sets of elements underlying a domain description were of arbitrary, possibly infinite size, we need to restrict ourselves to finite sets of fluent names, unit actions, and e- and v-propositions in order to obtain a finite logic program. The approach we follow here is based on the reification of whole state descriptions by treating them as single terms [17]. To this end, each fluent that holds in a state is formally represented by a term (a so-called fluent term), and to constitute a state representation these fluent terms are connected by a special binary function symbol, denoted $\circ$. For instance, the term $(\text{open} \circ \text{sleeps}) \circ \text{hurt}$ describes the state wrt. Example 4.1 where the door is open, the dog is sleeping, and the protagonist has hurt himself. Intuitively, the order in which the various fluent terms are connected is irrelevant as regards the state to be represented. Hence our connection function has some special properties, which are formalized using the following equational theory [17]:

\[
\forall X, Y, Z. (X \circ Y) \circ Z = X \circ (Y \circ Z) \quad \text{(associativity)}
\]

\[
\forall X, Y. \quad X \circ Y = Y \circ X \quad \text{(commutativity)}
\]

\[
\forall X. \quad X \circ \emptyset = X \quad \text{(unit element)},
\]

where the constant $\emptyset$ denotes a unit element for $\circ$, which corresponds to an empty collection of fluent terms. These three axioms (AC1, for short) are used as the underlying equational theory for our logic program.\(^\text{14}\) Therefore, the special function symbol $\circ$ will be referred to as the AC1-function, and a term consisting of subterms that are connected by this function will be referred to as an AC1-term. In what follows, we use the equality predicate $\sim_{\text{AC1}}$ in program clauses to illustrate that equality should always be related to the axioms above. Due to the law of associativity, we can omit parenthesis on the level of $\circ$ in any AC1-term.

On the basis of representing a state by a collection of fluent terms, the execution of actions is modeled through manipulation of such collections. For this reason the underlying approach is named fluent calculus. Aside from being closely related, in its basic form, to the Linear Connection Method [6] and reasoning about actions based on Linear Logic [13, 27], the fluent calculus has recently been shown to successfully deal both with the ramification [41] as well as with the qualification problem [38, 40], and with reasoning about continuous change [16].

\(^{14}\) While it suffices to consider these axioms in view of a suitable resolution procedure (see Section 5.4), the standard axioms of equality plus axioms allowing to derive inequalities are additionally required when discussing an adequate semantics for our program (see Section 5.2).
In the following subsection, 5.1, we describe how to construct a fluent calculus-based logic program corresponding to a domain description in $\mathcal{A}_{NCC}$. In Section 5.2, we discuss the semantics of the resulting program by applying the standard completion procedure [9] augmented by a special treatment of the underlying equational theory [19, 34]. In Section 5.3, we then prove soundness and completeness of the equational logic program (by taking the extended completion semantics) wrt. the semantics of $\mathcal{A}_{NCC}$. Finally, in Section 5.4 we discuss the applicability of a special resolution variant, namely, SLDENF-resolution [34, 39], to our logic program. We assume that the reader be familiar with the basic concepts of normal logic programs (i.e., logic programs augmented by negation-as-failure) as described, e.g., in the textbook [25]. We use a PROLOG-like syntax in denoting constants and predicates by lower case letters and variables by upper case letters. Moreover, free variables are assumed to be universally quantified and, as usual, the term $[h|t]$ denotes a list with head $h$ and tail $t$.

5.1. The Equational Logic Program

Let $D$ be a domain description in $\mathcal{A}_{NCC}$ based on fluent names $F_D$. For a proper representation of negative fluent literals, we introduce a unary function whose application to a term representing a fluent name indicates the negation of the latter. We will denote this function illustratively by a bar on top of its argument, like negation has been denoted in the action description languages. Formally, we employ a function $\tau$ mapping sequences of fluent literals to ACI-terms as follows:

$$\tau(l_1, \ldots, l_n) := l_1 \circ \cdots \circ l_n,$$

where $l_i \in F_D \cup \{\bar{f} \mid f \in F_D\} \ (1 \leq i \leq n)$, and in case $n = 0$ the function value of $\tau$ is the unit element $\emptyset$ of $\circ$.

A state $\sigma = \{f_1, \ldots, f_m\}$ over a fixed set of fluent names $F_D \supseteq \{f_1, \ldots, f_m\}$ is represented by an ACI-term as follows:

$$\gamma_D(\{f_1, \ldots, f_m\}) := f_1 \circ \cdots \circ f_m \circ f_{m+1} \circ \cdots \circ f_n,$$

where $\{f_1, \ldots, f_n\} = F_D$.

Finally, we also employ our ACI-function to represent compound actions, viz. by simply connecting the unit action names, which, too, are taken as terms to that end:

$$\mu(\{a_1, \ldots, a_k\}) := a_1 \circ \cdots \circ a_k,$$

where $\{a_1, \ldots, a_k\} \subseteq A_D$.

We are now prepared for translating domain descriptions $D$ in $\mathcal{A}_{NCC}$ into a set of logic program clauses. To begin with, we introduce, for each fluent name $f \in F_D$, a separate unit clause to relate $f$ to its counterpart $\bar{f}$:

$$FLUENT_D := \{\text{complement}(\tau(f, \bar{f})). f \in F_D\}. \quad (5.3)$$

Let $E_D$ be a given set of e-propositions. Then for each strict e-proposition we use an instance of the ternary predicate $eprop$ stating the action name, the effect, and the conditions:

$$EPROP_D := \{eprop(\mu(a), \tau(l), \tau(c_1, \ldots, c_n)). \quad \text{a causes } l \text{ if } c_1, \ldots, c_n \in E_D\}. \quad (5.4)$$
Analogously, alternative e-propositions describing possible effects of nondeterministic actions are encoded using the ternary predicate *alteprop*:

\[
\text{ALTEPROP}_D := \{ \text{alteprop}(\mu(a), \tau(e_1, \ldots, e_m), \tau(c_1, \ldots, c_n)) : \text{\(a\) alternatively causes \(e_1, \ldots, e_m\) if \(c_1, \ldots, c_n \in E_D\)} \}
\]  

(5.5)

*Example 5.1.* Let \(D_4\) denote the amalgamation of the two domains described in Example 3.1 and Example 4.1, respectively. We then have, say, \(\tau_D(sleeps, open) = \text{sleeps} \circ \text{open}, \gamma_D(\{\text{alive, sleeps, open}\}) = \text{alive} \circ \text{sleeps} \circ \text{open} \circ \text{loaded} \circ \text{nervous} \circ \text{hurt},\) and \(\mu_D(\{\text{activate, run\_into}\}) = \text{activate} \circ \text{run\_into}.\) The program clauses \(\text{FLUENT}_{D_4} \cup \text{EPROP}_{D_4} \cup \text{ALTEPROP}_{D_4}\) are as follows:

- \(\text{complement}(\text{loaded} \circ \text{loaded}).\)
- \(\text{complement}(\text{alive} \circ \text{alive}).\)
- \(\text{complement}(\text{nervous} \circ \text{nervous}).\)
- \(\text{complement}(\text{sleeps} \circ \text{sleeps}).\)
- \(\text{complement}(\text{hurt} \circ \text{hurt}).\)
- \(\text{complement}(\text{open} \circ \text{open}).\)
- \(\text{eprop}(\text{load}, \text{loaded}, \emptyset).\)
- \(\text{eprop}(\text{shoot}, \text{loaded}, \emptyset).\)
- \(\text{eprop}(\text{shoot}, \text{alive}, \text{loaded}).\)
- \(\text{alteprop}(\text{spin}, \text{loaded}, \emptyset).\)
- \(\text{alteprop}(\text{spin}, \text{loaded} \circ \text{nervous}, \emptyset).\)
- \(\text{eprop}(\text{activate}, \text{sleeps}, \emptyset).\)
- \(\text{eprop}(\text{run\_into}, \text{hurt}, \text{open}).\)
- \(\text{eprop}(\text{pull}, \text{open}, \emptyset).\)
- \(\text{eprop}(\text{activate} \circ \text{run\_into}, \text{open}, \emptyset).\)
- \(\text{eprop}(\text{activate} \circ \text{run\_into}, \text{hurt}, \text{hurt}).\)

In order to encode the general concept of transition underlying \(\mathcal{A}_{NCC}\), we use a ternary predicate \(\text{action}(i, a, h)\) stating that executing action \(a\) in state \(i\) possibly yields state \(h\). Following Definitions 4.1–4.3, a possible successor state is obtained by taking into account all applicable, strict e-propositions and a selection consisting of exactly one element of each set of applicable alternative e-propositions describing an identical subset of \(a\). This is reflected in the following definition of the \(\text{action}\) predicate:

\[
\text{action}(I, A, H) \leftarrow \text{selection}(A, S, I), \\
\neg \text{impossible}(H, I, A, S), \\
\neg \text{unfounded}(H, I, A, S), \\
\neg \text{inconsistent}(H).
\]

(5.6)
The intended meaning of this clause is the following: Let \( i \) be a term representing a state [cf. (5.1)] and \( a \) be a term representing a (compound) action [cf. (5.2)]\(^{15}\), then:

1. Let \( s \) be a term of the form \((b_1, e_1, c_1) \cdots (b_m, e_m, c_m)\) where each subterm \((b_j, e_j, c_j)\) \((1 \leq j \leq m)\) is a triple representing an alternative e-proposition \( b_j \) alternatively causes \( e_j \) if \( c_j \). An instance \( \text{selection}(a, s, i) \) is then intended to be true if \( s \) represents a selection for \( a \) wrt. state \( i \) (cf. Definition 4.1). To this end, we introduce the following program clause:

\[
\text{selection}(A, S, I) \leftarrow \neg \text{overrepresented}(A, S, I), \\
\neg \text{underrepresented}(A, S, I).
\] (5.7)

In words, the middle argument of \( \text{selection} \) contains a representation of not more than and also at least one element of each set of applicable alternative e-propositions. The predicates \( \text{overrepresented} \) and \( \text{underrepresented} \) are defined as follows:

\[
\text{overrepresented}(A, X \circ R, I) \leftarrow X \neq_{\text{AC1}} \emptyset, \\
\neg \text{underrepresented}(A, R, I).
\]

\[
\text{underrepresented}(A \circ B, S, C \circ J) \leftarrow \text{alt euprop}(A, E, C), \\
\neg \text{represented}(A, S).
\] (5.8)

In words, \( \text{underrepresented}(a \circ b, s, i) \) is true if there exists an applicable (wrt. state \( i \))\(^{16}\) alternative e-proposition for sub-action \( a \) but \( s \) does not include a triple representing an alternative for this particular action—and \( \text{overrepresented}(a, s, i) \) is true if \( s \) contains more than these triples.

2. An instance \( \text{impossible}(h, i, a, s) \) is intended to be true if \( h \), which is intended to represent a possible successor state of \( i \) wrt. action \( a \), contains a fluent literal whose negation (but not the literal itself) is claimed by some non-overruled e-proposition (either a strict one, or an alternative one that has been selected via \( s \)).\(^{17}\) Accordingly, the definition of \( \text{impossible} \) is the following:

\[
\text{impossible}(F \circ H, I, A, S) \leftarrow \text{overruled}(F, I, \emptyset, A, S), \\
\text{complement}(F \circ G), \\
\neg \text{overruled}(G, I, \emptyset, A, S).
\] (5.9)

An instance \( \text{overruled}(h, i, b, a, s) \) is intended to be true if \( h \) contains a fluent literal whose negation is claimed by a (strict or selected) e-proposition for some action \( c \) such that \( c \supseteq b \) and \( c \subseteq a \). The clauses defining \( \text{overruled} \) follow Definition 4.3:

\[
\text{overruled}(F \circ H, C \circ J, A, A \circ B \circ D, S) \\
\leftarrow \text{e prop}(A \circ B, G, C), \\
B \neq_{\text{AC1}} \emptyset.
\] (5.10)

\(^{15}\)For the sake of readability, in the following description we sometimes identify a term \( t \) which represents a state (or a sequence of fluent literals or a compound action, respectively) with the state \( \gamma_D^{-1}(t) \) itself (or with \( \tau^{-1}(t) \) or \( \mu^{-1}(t) \), respectively).

\(^{16}\)Note that applicability means the conditions \( c \) of the e-proposition are true in \( i \), which is guaranteed if the terms \( c \circ J \) and \( i \) are unifiable under the AC1 theory (see Lemma 5.1, below).

\(^{17}\)Notice that in case both a fluent literal and its negation are claimed to be true by two or more non-overruled e-propositions (strict or selected), either of them may hold in a resulting state. This encodes our way of solving conflicts, as formalized in Definition 4.3.
complement(\( F \circ G \)),
\( \neg \overruled(G, C \circ J, A \circ B, A \circ B \circ D, S) \).
\( \overruled(F \circ H, C \circ J, A, A \circ B \circ D, (A \circ B, G \circ E, C) \circ R) \)
\( \leftarrow B \not\in_{AC1}\emptyset, \)
complement(\( F \circ G \)),
\( \neg \overruled(G, C \circ J, A \circ B, A \circ B \circ D, (A \circ B, G \circ E, C) \circ R) \).

In words, the particular effect \( F \) of an action \( A \) is overruled by an \textit{eprop} (first clause) or an \textit{alteprop} that has been selected (second clause) postulating the effect \( G = \overline{F} \) of an action \( A \circ B \supset A \) if this e-proposition is not overruled itself.

3. An instance \textit{unfounded}(h, i, a, s) is intended to be true if \( h \) contains a fluent literal whose negation holds in \( i \) but there is no (strict or selected via \( s \)) e-proposition for action \( a \) wrt. state \( i \) that induces this change. The definition of this predicate is as follows:

\[ \text{unfounded}(F \circ H, G \circ I, A, S) \leftarrow \text{complement}(F \circ G), \]
\[ \neg \overruled(G, G \circ I, \emptyset, A, S). \]

(5.11)

In words, a change from fluent literal \( G = \overline{F} \) to \( F \) is unfounded if there is no (strict or selected) e-proposition that overrules the assumption that \( G \) continues to be true.

4. An instance \textit{inconsistent}(h) is intended to be true if \( AC1 \)-term \( h \) does not represent a state [cf. (5.1)], that is, if \( h \) contains some fluent term twice or more, or it contains a fluent name along with its negation, or there is some fluent name \( f \in F_D \) such that neither \( f \) nor \( \overline{f} \) occur as subterm. These three criteria are encoded by the following clauses:

\[ \text{inconsistent}(G \circ G \circ H) \leftarrow G \not\in_{AC1}\emptyset. \]
\[ \text{inconsistent}(F \circ H) \leftarrow \text{complement}(F). \]
\[ \text{inconsistent}(H) \leftarrow \text{complement}(F \circ G), \]
\[ F \not\in_{AC1}\emptyset, G \not\in_{AC1}\emptyset, \]
\[ \neg \text{holds}(F, H), \neg \text{holds}(G, H). \]

(5.12)

Having encoded the transition relation, we now show how to model the application of an action sequence \([a_1, \ldots, a_m]\) to some initial state. Since the resulting state is model-dependent and, in particular, determined by the associated function \( \varphi \) (cf. Definition 4.4), we need to find a way to encode the latter within the model generation process. To this end, we first introduce the notion of an \textit{action tree} serving as a (minimal) basis for the set of v-propositions underlying the domain description at hand:

\textbf{Definition 5.1.} Let \( D \) be a domain description in \( \mathcal{A}_{NCC} \) with action names \( A_D \) and v-propositions \( V_D \). An \textit{action tree} is a tree \( \mathcal{B} \) whose nodes are finite lists over \( 2^{A_D} \) such that

1. the root of \( \mathcal{B} \) is \([\ ]\), and
2. if \( [a_1, \ldots, a_m, a_{m+1}] \) is a node in \( \mathcal{B} \) then its predecessor is \([a_1, \ldots, a_m]\) (\( m \geq 0 \)).
FIGURE 2. (a) An action tree describing two directions of development, which forms a minimal basis for domains with v-propositions \{(3.7), (3.8)\}, and (b) the same tree augmented by variables to record the outcomes in a concrete model.

An action tree $\mathcal{B}$ is called basis for $D$ iff for each v-proposition $l$ after $[a_1, \ldots, a_m]$ in $V_D$ the sequence $[a_1, \ldots, a_m]$ is a node in $\mathcal{B}$. Moreover, the minimal basis for $D$ is the basis with a minimal number of nodes.

As an example, suppose given the v-propositions \{(3.7), (3.8)\}, then Figure 2(a) depicts the minimal basis for the corresponding domain. Notice that for any finite set of v-propositions a unique minimal basis exists and is finite.

The purpose of a minimal basis is to indicate which arguments of the model component $\varphi$ are of interest—regarding the underlying v-propositions—when searching for models. Now, to record the actual values of $\varphi$ in a particular model, we assign variables, which are intended to be substituted by states, to the edges of the basis as follows. Let $\mathcal{B}_D$ be the minimal basis for $D$ containing $n + 1$ nodes ($n \geq 0$). Furthermore, let $X_1, \ldots, X_n$ be pairwise different variables assigned one-to-one to the edges of the tree, then each node $[a_1, \ldots, a_m, a_{m+1}]$ in $\mathcal{B}_D$ ($m \geq 0$) is replaced by the sequence of pairs $[(a_1, X_1), \ldots, (a_m, X_m), (a_{m+1}, X_i)]$ where

1. the predecessor is replaced by $[(a_1, X_a), \ldots, (a_m, X_\delta)]$, and
2. the edge from the predecessor to the node itself is labeled with $X_i$.

A possible labeling of our example tree is depicted in Figure 2(b).

In order to encode in our logic program the notion of models for domain descriptions in $\mathcal{M}_NCC$, we first introduce the ternary predicate result. Its intended meaning is that $\text{result}(\gamma_D(\sigma_0), [(\mu(a_1), h_1), \ldots, (\mu(a_m), h_m)], \gamma_D(M^{[a_1, \ldots, a_m]}))$ is true iff the application of $[a_1, \ldots, a_m]$ to initial state $\sigma_0$ yields the state $M^{[a_1, \ldots, a_m]}$ in a model which satisfies $\varphi([a_1, \ldots, a_i]) = \gamma_D^{-1}(h_i)$ for each $1 \leq i \leq m$. It is required that each $h_i$ represent a possible successor state of applying $a_i$ to the preceding state,
according to the underlying transition relation. The clauses defining result thus are as follows:

\[
\begin{align*}
\text{result}(I, [\ ], I). \\
\text{result}(I, [(A, H)\{P\}, G]) & \leftarrow \text{action}(I, A, H), \\
\text{result}(H, P, G).
\end{align*}
\]

(5.13)

In words, \( M^{[1]} \) is \( \sigma_0 \), and in case \( m > 0 \), \( M^{[a_1, \ldots, a_m]} \) is obtained by computing a successor state \( H = \gamma_D^{-1}(h_1) \) of executing \( a_1 \) in \( \sigma_0 \) and by applying the remaining sequence \( [a_2, \ldots, a_m] \) to this state.

Finally, to encode the v-propositions \( V_D = \{ l_i \text{ after } [a_i_1, \ldots, a_i_m] \mid 1 < i < n \} \), we use the following clause defining the predicate model. The construction of this clause is grounded on a given minimal basis \( \mathcal{B}_D \) augmented by variables for the domain under consideration:

\[
\begin{align*}
\text{model}(I, X_1, \ldots, X_\beta) \\
& \leftarrow \neg \text{inconsistent}(I), \\
& \quad \text{result}(I, [(\mu(a_{i_1}), X_{a_1}), \ldots, (\mu(a_{i_m}), X_{a_i})], \tau(l_i) \circ G_1), \\
& \quad \vdots \\
& \quad \text{result}(I, [(\mu(a_{n_1}), X_{a_n}), \ldots, (\mu(a_{n_m}), X_{a_n})], \tau(l_n) \circ G_n),
\end{align*}
\]

where \( X_1, \ldots, X_\beta \) are the variables assigned to \( \mathcal{B}_D \) and, for each \( 1 < i < n \), the variables \( X_{a_1}, \ldots, X_{a_i} \) are chosen according to the corresponding node (for the action sequence \( [a_i_1, \ldots, a_i_m] \) in the labeled basis \( \mathcal{B}_D \)). Hence the intended meaning is that \( \text{model}(i, h_1, \ldots, h_\beta) \) is true if \( i \) represents a consistent initial state such that all v-propositions in \( V_D \) are satisfied under the assumption that \( h_1, \ldots, h_\beta \) are the states resulting from executing the respective action sequence.

Example 5.1 (continued). Given the four v-propositions

\[
\begin{align*}
\text{alive} \quad \text{after} \quad [\{\text{load}\}] \\
\text{alive} \quad \text{after} \quad [\{\text{load}\}, \{\text{spin}\}, \{\text{shoot}\}] \\
\text{initially} \quad \text{sleeps} \\
\overline{\text{hurt}} \quad \text{after} \quad [\{\text{activate}, \text{run_into}\}]
\end{align*}
\]

(5.15)

these are encoded by the following program clause if we take the suitably labeled basis shown in Figure 3:

\[
\begin{align*}
\text{model}(I, X_1, X_2, X_3, X_4) \\
& \quad \leftarrow \neg \text{inconsistent}(I), \\
& \quad \text{result}(I, [(\text{load}, X_1)], \text{alive} \circ G_1), \\
& \quad \text{result}(I, [(\text{load}, X_1), (\text{spin}, X_2), (\text{shoot}, X_3)], \text{alive} \circ G_2), \\
& \quad \text{result}(I, [\ ], \text{sleeps} \circ G_3), \\
& \quad \text{result}(I, [(\text{activate} \circ \text{run_into}, X_4)], \overline{\text{hurt}} \circ G_4).
\end{align*}
\]

(5.16)
To summarize, a domain description $D$ in $\mathcal{NCC}$ is translated into the set of clauses $P_D = \text{FLUENT}_D \cup \text{EPROP}_D \cup \text{ALTEPROP}_D \cup \{(5.6)-(5.14)\}$. The resulting equational logic program we denote by $(P_D, \text{AC1})$, and the class of resulting equational logic programs shall be denoted by $\mathcal{EC}_{NCC}$. The reader might have noticed that the major part of this program, clauses (5.6)-(5.13), is domain independent and constitutes an intuitive and direct translation of Definitions 4.1-4.4.

5.2. The Completion Semantics

The equational logic program developed in the previous subsection contains negative literals in the bodies of some clauses. These negative literals are intended to be treated by the (nonmonotonic) negation-as-failure principle. An adequate semantics for such programs, which is based on classical first-order logic, is obtained by applying an extension of Clark's completion procedure [9] to the program. The idea is to consider the set of program clauses which define a predicate $p$ as a complete description of the positive information regarding $p$:

**Definition 5.2.** Let $p(t_1, \ldots, t_n) \rightarrow L_1, \ldots, L_m$ be a program clause, and let $\overline{Y}$ denote a sequence of all variables which occur in this clause. Furthermore, let $X_1, \ldots, X_n$ be pairwise different variables not in $\overline{Y}$. Then the rectified form of this clause is the formula $p(X_1, \ldots, X_n) \leftrightarrow \exists \overline{Y}(X_1 = t_1 \land \cdots \land X_n = t_n \land L_1 \land \cdots \land L_m)$. Let $p$ be an arbitrary predicate symbol and

$$
p(X_1, \ldots, X_n) \leftarrow D_1
$$

$$
\vdots
$$

$$
p(X_1, \ldots, X_n) \leftarrow D_k
$$

be all clauses in a program $P$ defining $p$ in rectified form ($k \geq 0$). The completed definition of $p$ in $P$ is the formula

$$\forall X_1, \ldots, X_n(p(X_1, \ldots, X_n) \leftrightarrow D_1 \lor \cdots \lor D_k)$$

[In case $k = 0$ this reduces to $\forall(\neg p(X_1, \ldots, X_n))$.] The completion $P^*$ of $P$ is the conjunction of the completed definitions of all predicate symbols occurring in the underlying alphabet except for the equality predicate "=".

Given a domain description $D$ in $\mathcal{NCC}$, the entire completion of the corresponding program clauses $P_D$, written $P^*_D$, is shown in the Appendix.
Aside from completing the program clauses, a logic program with an underlying equational theory requires a special kind of completion for the equality predicate since axioms are needed which allow for proving inequalities in order to derive negative information. In case of standard completion [9], some axiom schemata are added to the completed formula which allow for proving inequality of two terms whenever these are not syntactically unifiable. The concept of unification completeness [19, 34] generalizes these axioms schemata for arbitrary equational theories.

Prior to stating the formal definition, we need to introduce some notions and notation related to unification theory, taken from the survey article [2]. The standard axioms of equality are

\[
\begin{align*}
X &= X & \text{(reflexivity)} \\
X &= Y \rightarrow Y = X & \text{(symmetry)} \\
X &= Y \land Y = Z \rightarrow X = Z & \text{(transitivity)} \\
X_i &= Y \rightarrow f(X_1, \ldots, X_i, \ldots, X_n) = f(X_1, \ldots, Y, \ldots, X_n) & \text{(substitutivity I)} \\
X_i &= Y \rightarrow [p(X_1, \ldots, X_i, \ldots, X_n) \equiv p(X_1, \ldots, Y, \ldots, X_n)] & \text{(substitutivity II)}
\end{align*}
\]

for each \(n\)-place function symbol \(f\) and predicate \(p\), and for each \(1 \leq i \leq n\). If \(E\) is an equational theory then two terms \(s, t\) are called \(E\)-equal if the formula \(s = t\) is a logical consequence of \(E\) plus the standard equality axioms. Two terms \(s, t\) are said to be \(E\)-unifiable if there exists a substitution \(\theta\) such that \(s \theta\) and \(t \theta\) are \(E\)-equal; in which case \(\theta\) is called an \(E\)-unifier for \(s, t\). A complete set of \(E\)-unifiers \(cU_E(s, t)\) for two terms \(s, t\) is a set of \(E\)-unifiers for \(s, t\) such that each \(E\)-unifier for \(s, t\) is subsumed by at least one element in \(cU_E(s, t)\). As in [34], given a substitution \(\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}\) we use \(eqn(\theta)\) to denote the formula \(X_1 = t_1 \land \cdots \land X_n = t_n\).

**Definition 5.3.** Let \(E\) be an equational theory. A consistent set of first-order formulas \(E^*\) is called unification complete wrt. \(E\) if it consists of the axioms in \(E\), the standard equality axioms, and a number of equational formulas, i.e., formulas with \("=\) as the only predicate, such that for any two terms \(s, t\) with variables \(\overline{X}\) the following holds:

1. If \(s, t\) are not \(E\)-unifiable then \(E^* \models \exists \overline{X}. s = t\).
2. If \(s, t\) are \(E\)-unifiable then for each complete set of \(E\)-unifiers \(cU_E(s, t)\)

\[
E^* \models \forall \overline{X}\left(s = t \rightarrow \bigvee_{\theta \in cU_E(s, t)} \exists \overline{Y}. eqn(\theta)\right),
\]

(5.17)

where \(\overline{Y}\) denotes the variables which occur in \(eqn(\theta)\) but not in \(\overline{X}\).

In [18], we have proved the existence of a unification complete theory \(AC1^*\) for the equational theory AC1 used in \(\mathcal{FE}_{NCC}\). Since we do not intend to compute with \(AC1^*\), we are only interested in the properties of this theory as given by Definition 5.3; its actual design is irrelevant for our analysis. Given a domain description \(D\) in \(\mathcal{D}_{NCC}\), we take the formulas \(P^*_D \cup AC1^*\) as the logic programming semantics for the corresponding program \((P_D, AC1)\).
5.3. Soundness and Completeness of the Translation

Based on the completion semantics, we now prove soundness and completeness of our equational logic program wrt. the entailment relation defined for $\mathcal{A}_{NCC}$. We start with a number of lemmas concerning specific parts of our program.

**Lemma 5.1.** Let $D$ be a domain description in $\mathcal{A}_{NCC}$ with fluent names $F_D$. Furthermore, let $c_1, \ldots, c_m$ be a sequence of fluent literals ($m \geq 0$) and $\sigma \subseteq F_D$ be a state. Then each of $c_1, \ldots, c_m$ holds in $\sigma$ iff $\tau(c_1, \ldots, c_m) \circ V$ and $\gamma_D(\sigma)$ are ACI-unifiable (where $V$ is an arbitrary variable).

Analogously, let $A_D$ be the underlying set of unit actions and $a, b \subseteq A_D$ two actions then $b \subseteq a$ iff $\mu(b) \circ V$ and $\mu(a)$ are ACI-unifiable.

**Proof.** In case $m = 0$, $\emptyset \circ V$ and $\gamma_D(\sigma)$ are always ACI-unifiable using the substitution $\{V \mapsto \gamma_D(\sigma)\}$. Otherwise, associativity and commutativity of $\circ$ imply that the two terms are ACI-unifiable iff each subterm $\tau(c_i)$ occurs in $\gamma_D(\sigma)$, the latter of which contains exactly the fluent literals that hold in $\sigma$. The second claim follows analogously. $\square$

Notice that moreover unification completeness of ACI* ensures

$$\text{ACI}^* \vDash \forall V. \tau(c_1, \ldots, c_m) \circ V \neq \gamma_D(\sigma)$$

whenever some $c_i$ ($1 \leq i \leq m$) does not hold in $\sigma$ (cf. clause 1 in Definition 5.3).

In what follows, a notation like $(5.3^*)$ refers to the completed definition(s), listed in the appendix, of the clause(s) in $(5.3)$.

**Lemma 5.2.** Let $D$ be a domain description in $\mathcal{A}_{NCC}$. Furthermore, let $\sigma \subseteq F_D$ be a state and $a \subseteq A_D$ an action. If $b'_1, \ldots, b'_n$ are actions and $e'_1, \ldots, e'_n$ and $c'_1, \ldots, c'_n$ both are sequences of fluent literals ($n \geq 0$) then

$$P^*_\sigma \cup \text{ACI}^* \vdash$$

$$\text{selection}(\mu(a), (\mu(b'_1), \tau(e'_1), \tau(c'_1)), \ldots, (\mu(b'_n), \tau(e'_n), \tau(c'_n)), \gamma_D(\sigma))$$

iff the following holds:

For each $b \subseteq a$ let $\Pi^*_b$ be the set of all alternative e-propositions in $E_D$ for action $b$ whose conditions hold in $\sigma$. Then there exists a selection $\Lambda^*_a = \lambda_{b'_1}, \ldots, \lambda_{b'_n}$ for $a$ wrt. $\sigma$ such that $n = m$ and such that there is a one-to-one correspondence between the triples $(\mu(b'_i), \tau(e'_i), \tau(c'_i)), \ldots, (\mu(b'_n), \tau(e'_n), \tau(c'_n))$ and the elements of $\Lambda^*_a$.

$-$ where $(\mu(b'_i), \tau(e'_i), \tau(c'_i))$ corresponding to $\lambda_{b'_i} = \mu$ alternatively causes $E_i \notin C$ means that $b'_i = b_j$, $e'_i = E$, and $c'_i = C$.

**Proof.** Let $s = (\mu(b'_1), \tau(e'_1), \tau(c'_1)), \ldots, (\mu(b'_n), \tau(e'_n), \tau(c'_n))$. From $(5.8^*)$ and $(5.5^*)$ in conjunction with Lemma 5.1 it follows that underrepresented$(\mu(a), s, \gamma_D(i))$ is entailed iff there exists some $b \subseteq a$ such that $\Pi^*_b \neq \emptyset$ but no subterm $(\mu(b), t_1, t_2)$ occurs in $s$. Accordingly, following $(5.8^*)$, overrepresented$(\mu(a), s, \gamma_D(i))$ is entailed iff $s$ includes any other subterms aside from exactly one subterm of the form $(\mu(b), t_1, t_2)$ for each $b \subseteq a$ with $\Pi^*_b \neq \emptyset$. Thus, for each $(\mu(b'), \tau(e'_i), \tau(c'_i))$ in $s$ we can find some $b_j \in \{b_{j_1}, \ldots, b_{m}\}$ and, hence, some $\lambda_{b_j}$ in $\Lambda^*_a$ such that $b_j = b'_i$ and vice versa. Moreover, $(5.8^*)$ in conjunction with $(5.5^*)$ ensures that $e'_i = E$ and $c'_i = C$. The claim then follows from $(5.7^*) \in P^*_\sigma$. $\square$
The following lemma describes the connection between the program clauses defining the predicate \( \text{overruled} \) and Definition 4.2:

**Lemma 5.3.** Let \( D \) be a domain description in \( \mathcal{A}_{\text{NCC}} \). Furthermore, let \( e_1, \ldots, e_n \) be a sequence of fluent literals \((n \geq 1)\), \( \sigma \) a state, \( a, b \) actions such that \( b \subseteq a \), and \( s \) a term representing a selection \( \Lambda^a_\sigma \) for a wrt. \( \sigma \). Then

\[
P^*_D \cup \text{ACI}^* \models \neg \text{overruled}(\tau(e_1, \ldots, e_n), \gamma_D(\sigma), b, a, s) \tag{5.18}
\]

iff there is no \( e_i \) \((1 \leq i \leq n)\) such that \( a \) causes \( e_i \) by some \( c \) wrt. \( \Lambda^a_\sigma \) such that \( c \supset b \) and \( c \subseteq a \).

**Proof.** From \( b \subseteq a \) we know \( |b| \leq |a| \). The proof is by induction on \( m = |a| - |b| \). In case \( m = 0 \) (that is, \( a = b \)), no such \( c \) can possibly exist. Correspondingly, the literal \( B_{AC} : \text{ACI}^* \text{ overruled}(e_1, \ldots, e_n),\gamma_D(\sigma), b, a, s) \) in both disjuncts of (5.10*) guarantees (5.18) to hold.

In case \( m > 0 \), from (5.10*) in conjunction with (5.3*) and (5.4*) and Lemma 5.1 we know that

\[
P^*_D \cup \text{ACI}^* \models \text{overruled}(\tau(e_1, \ldots, e_n), \gamma_D(\sigma), b, a, s) \tag{5.19}
\]

iff some \( e_i \) \((1 \leq i \leq n)\) and some action \( c \) (with \( b \subset c \subseteq a \)) exist such that clause 2 of Definition 4.2 holds for fluent literal \( e_i \) and such that

\[
P^*_D \cup \text{ACI}^* \models \neg \text{overruled}(\tau(e_i), \gamma_D(\sigma), c, a, s) \tag{5.20}
\]

Since \( |b| < |c| \leq |a| \), we have \( 0 \leq |a| - |c| < m \). Hence the induction hypothesis is applicable and ensures (5.20) be true iff \( a \) does not cause \( e_i \) by some \( c' \) wrt. \( \Lambda^a_\sigma \) such that \( c \subset c' \subseteq a \). Thus, according to Definition 4.2, we know that (5.19) is true iff \( a \) causes some \( e_i \) by some \( c \) wrt. \( \Lambda^a_\sigma \) such that \( c \supset b \). Hence the latter (i.e., \( a \) causing some \( e_i \)) being false is equivalent to (5.18) being true. \( \square \)

Finally, we prove the correctness of our definition of consistency as regards ACI-terms that are intended to represent states:

**Lemma 5.4.** Let \( D \) be a domain description with fluents \( F_D \) and \( i \) an ACI-term then

\[
P^*_D \cup \text{ACI}^* \models \neg \text{inconsistent}(i)
\]

iff for each fluent name \( f \in F_D \),

1. either \( f \) or else \( \bar{f} \) occurs in \( i \), and
2. \( i \) does not contain any fluent term more than once.

**Proof.** In conjunction with Lemma 5.1, the first disjunct in (5.12*) ensures that no fluent term occurs twice or more in \( i \), the second disjunct ensures that \( i \) does not contain a fluent name along with its negation, and the third disjunct ensures that each fluent name is represented affirmatively or negatively. \( \square \)

The following theorem concerning transition of states forms the basis of our soundness and completeness result:

**Theorem 5.1.** Let \( D \) be a domain description in \( \mathcal{A}_{\text{NCC}} \). If \( \Phi \) is a transition relation for \( E_D \) then for each state \( \sigma \subseteq F_D \), action \( a \subseteq A_D \), and each term \( h \)

\[
P^*_D \cup \text{ACI}^* \models \text{action}(\gamma_D(\sigma), \mu(a), h) \tag{5.21}
\]

iff \((\sigma, a, \sigma') \in \Phi \) where \( h \) represents state \( \sigma' \).

**Proof.** From Lemma 5.4 and from (5.6*) \( \in P^*_D \) and Lemma 5.2 it follows that (5.21) holds iff \( h \) represents a state, there exists some term \( s \) which represents a
selection $A^e_a$ for $a$ wrt. $\sigma$, and

$$P^*_D \cup ACI^* \models \neg \text{unfounded}(h, \gamma_0(\sigma), \mu(a), s) \land \neg \text{impossible}(h, \gamma_0(\sigma), \mu(a), s)$$

holds. Following (5.11*), (5.9*) and Lemma 5.3 this in turn holds iff

1. for each fluent literal $l$ which is true in $\sigma$ but false in $\sigma'$ there is some applicable (strict or selected in $s$) e-proposition postulating $l$, and
2. there is no fluent literal $l$ that holds in $\sigma'$ such that $a$ causes $l$ in $\sigma$ but not $a$ causes $l$ in $\sigma'.^{18}$

According to Definition 4.3, this is equivalent to $(\sigma, a, \sigma') \in \Phi$. □

Based on this theorem, we can prove soundness and completeness of our equational logic program wrt. the semantics of $\mathcal{A}_{NCC}$ as given by Definition 4.4. More precisely, we will prove that a v-proposition $l$ after $[a_1, \ldots, a_m]$ is entailed iff no model can be found—according to (5.14*)—which contradicts this v-proposition. To this end, recall Definition 5.1, where we have introduced the concept of an action tree to encode the model component $\varphi$. In order to test entailment of a v-proposition using the literal $\text{result}(i, [(a_1, X_1), \ldots, (a_m, X_m)], \tau(l) \circ G)$, we have to take into account the underlying labeled basis which has been used to construct clause (5.14). Let $k$ be maximal in $\{0, \ldots, m\}$ such that $[a_1, \ldots, a_k]$ occurs in this tree then we use the $k$ variables $X_1, \ldots, X_\delta$ assigned to the actions in this node plus pairwise different, new variables $X'_1, \ldots, X'_{m-k}$ for the tail $[a_{k+1}, \ldots, a_m]$ of the action sequence under consideration. As before, $X_1, \ldots, X_\beta$ denotes the entire ordered sequence of variables assigned to the underlying basis:

**Theorem 5.2.** Let $D$ be a domain description in $\mathcal{A}_{NCC}$. Furthermore, let $\nu = l$ after $[a_1, \ldots, a_m]$ be a v-proposition. Then $\nu$ is entailed by $D$ iff

$$P^*_D \cup ACI^* \models \exists X, \exists I, \exists X' \left( \text{model}(I, X_1, \ldots, X_\beta) \land \text{result}(I, [(\mu(a_1), X_a), \ldots, (\mu(a_k), X_\delta), (\mu(a_{k+1}), X'_1), \ldots, (\mu(a_m), X'_{m-k})], \tau(I) \circ G) \right)$$

where $X = X_1, \ldots, X_\beta, X'_1, \ldots, X'_{m-k}$.

**Proof.** Let $\Phi$ be a transition relation for $D$. From Lemma 5.4, (5.14*), (5.13*) and repeated application of Theorem 5.1 it follows that $\text{model}(i, h_1, \ldots, h_\beta)$ is entailed iff the structure $((\sigma_0, \Phi, \varphi))$ satisfies every v-proposition in $D$, that is, if it is model for $D$—where $i$ represents state $\sigma_0$ and $h_1, \ldots, h_\beta$ correspond to $\varphi$ in the following sense: For each $j \in \{1, \ldots, \beta\}$, if, in the underlying labeled basis, $X_j$ has been assigned to the edge ending in the node $[a'_1, \ldots, a'_j]$ then $h_j = \gamma_p(\varphi([a'_1, \ldots, a'_j]))$. The claim then follows from (5.13*) and the fact that $\nu$ is entailed iff there is no model $((\sigma_0, \Phi, \varphi))$ for $D$ in which $l$ holds in $\varphi([a_1, \ldots, a_m])$.

---

18 Notice again that if $a$ causes both $l$ and $l'$ then the corresponding fluent name belongs to $Bf(a, \sigma)$, that is, either value may be true in $\sigma'$. 
5.4. SLDENF-Resolution

Our equational logic programs $\mathcal{F}_{NCC}$ are based on a special equational theory, viz. AC1, and they also contain negation in the body of some program clauses. In the preceding subsection, we have taken the completion of these programs as an adequate semantics if negative literals are to be treated by negation-as-failure. An adequate computation mechanism for programs including equality and (nonmonotonic) negation is SLDENF-resolution, which is based on SLD-resolution (see, e.g. [25]) but with the standard unification procedure replaced by an $E$-unification algorithm and negation-as-failure used to treat negative subgoals. A formal introduction to this resolution principle can be found in [34, 39]. In [34], soundness of SLDENF-resolution wrt. the completion semantics (including the use of unification complete theories) has been proved for arbitrary equational logic programs with negation. That is, if $P$ is a set of normal program clauses, $E$ an equational theory, and $L_1, \ldots, L_n$ a query for which there exists an SLDENF-refutation with computed answer substitution $\theta$, then

$$P^* \cup E^* \models \forall (L_1 \land \cdots \land L_n) \theta$$

Combining this result with Theorem 5.2 proves that SLDENF-resolution can be applied as a sound proof procedure for the entailment relation defined in $\mathcal{A}_{NCC}$.

As regards completeness of SLDENF-resolution, it is common knowledge that completeness cannot be guaranteed even for the special case of programs with negation and the empty equational theory [9, 1]. The classical completeness result for SLDNF-resolution is restricted to so-called hierarchical and allowed programs [9, 1]. In a hierarchical program, any SLDNF-derivation is necessarily finite, and the allowedness criterion prevents so-called floundering: Since by definition of SLD(EN)F-resolution negative subgoals can be selected only if they are ground, a derivation might end up with only non-ground negative subgoals. In such cases, the proof procedure does not come to a conclusion.

In [39], the aforementioned classical result has been lifted to logic programs with equational theories. It has been shown that this is possible only in case the underlying equational theory $E$ meets two restrictions: It should be finitary (that is, for each two terms $s$ and $t$ there exists a finite complete set of $E$-unifiers) to ensure finiteness of derivations in hierarchical programs, and it should also be regular (that is, for each equation $s = t \in E$ the set of variables occurring in $s$ equals the set of variables occurring in $t$) to avoid the problem of floundering in allowed programs. See [39] for a more detailed discussion and formal proof.

The equational theory used in this paper, AC1, is known to be both finitary [35] and, obviously, regular. Nonetheless the result presented in [39] cannot be applied since the program developed in Section 5.1 is neither hierarchical nor allowed. We therefore have to perform a more detailed and specific analysis.

Although the programs $\mathcal{F}_{NCC}$ are not hierarchical, it can be shown that all SLDENF-derivations we are interested in to decide entailment wrt. $\mathcal{A}_{NCC}$ are necessarily finite: Since no mutual recursion involving two or more program clauses occurs, the only crucial clauses are direct recursive ones, that is, where the predicate in the head also occurs in the body.

There are three clauses of this kind, shown in (5.10) and (5.13), respectively. As regards the two definitions of overruled in (5.10), it is easy to see that each recursive call increases the size (viz. the number of subterms) of the third argument, $A$. Moreover, the body of the first (resp. second) clause can only be
satisfied if there exists a strict e-proposition (resp. a selected alternative e-proposition) for an action which includes \( A \). Since there is only a limited number of such propositions in a concrete domain, the recursive calls eventually stop, provided a fair selection rule is used.

Analogously, the number of recursive calls of result, cf. (5.13), is limited by the size of the second argument, provided the latter is (partially) instantiated. This is indeed the case in both clause (5.14) and the query used to decide entailment of an additional v-proposition (cf. Theorem 5.2).

While it is easy to show finiteness of derivations, we cannot in general prove non-floundering. As a matter of fact, whenever we try to decide entailment of a v-proposition \( l \) after \([a_1, \ldots, a_m]\) by creating the clause

\[
\text{satisfiable } \Leftarrow \text{model}(I, \bar{X}),
\]

\[
\text{result}(I, [(\mu(a_1), X_1), \ldots, (\mu(a_m), X_m)], \tau(I) \circ G).
\]

(see Theorem 5.2) and using the query \( \neg \text{satisfiable} \) then the derivation flounders after clause (5.14) has been applied— to solve the subgoal \text{model}(I, \bar{X})—since \( \neg \text{inconsistent}(I) \) cannot be selected as \( I \) is a variable. Analogously, whenever clause (5.13) has been applied such that a subgoal of the form \text{action}(I, A, H) \) occurs then this can only be resolved using clause (5.6). This, however, requires \( I \) and \( H \) be fully instantiated if the derivation is not to flounder.\(^{19}\) Moreover, whenever a suitable collection \( S \) of alternative e-propositions is to be selected via clause (5.7) then this additionally requires \( S \) to be instantiated.

There are two ways of solving this problem. First, one could define an extension of the SLDENF-resolution principle that supports a proper treatment of non-ground, negative subgoals, such as the concept of constructive negation, which is a well-known technique to avoid floundering in case of non-equational logic programs \([8, 29]\). This, however, requires a new formal definition of an extended calculus, and then soundness and completeness have to be proved again.

Here we follow a simpler and more straightforward way. It is possible to rewrite some program clauses such that the crucial variables become instantiated early enough during the derivation to prevent floundering. To this end, we add a clause that provides all possible collections of size \(|D|\) consisting of fluent literals, where \( D \) denotes the underlying set of fluent names of domain \( D \):

\[\text{sterm}(I) \Leftarrow I = \text{AC}_{1} H_{1} \circ \ldots \circ H_{|D|},\]

\[\text{complement}(H_{1} \circ J_{1}), \ldots, \text{complement}(H_{|D|} \circ J_{|D|}).\]

where \( H_{1}, J_{1}, \ldots, H_{|D|}, J_{|D|} \) are pairwise distinct variables.

The new predicate \( \text{sterm} \) can then be used to instantiate the crucial arguments in advance. The following clause is a modification of (5.13):

\[
\text{result}(I, [], I) ,
\]

\[
\text{result}(I, [(A, H)|P], G) \Leftarrow \text{sterm}(H),
\]

\[\text{action}(I, A, H), \text{result}(H, P, G).\]

\(^{19}\) The variable \( A \) always becomes instantiated when deciding entailment since action sequences in v-propositions are fixed.
and this clause can be used instead of (5.14):

\[ \text{model}(I, X_1, \ldots, X_n) \]
\[ \leftarrow \text{sterm}(I), \]
\[ \neg \text{inconsistent}(I), \]
\[ \text{result}(I, \left[ \left( \mu(a_{i1}), X_{a_i}, \ldots, \mu(a_{im_i}), X_{b_i} \right] \circ \tau(l_i) \circ G_1 \right), \]
\[ \vdots \]
\[ \text{result}(I, \left[ \left( \mu(a_{n1}), X_{a_n}, \ldots, \mu(a_{nm_n}), X_{b_n} \right] \circ \tau(l_n) \circ G_n \right). \]

Finally, to avoid floundering after clause (5.7) has been applied, by the following clause we provide all suitable instances for a variable which represents a selection, that is, which contains a collection of triples each of which represents an alternative e-proposition:

\[ \text{eterm}(\emptyset). \]
\[ \text{eterm}((A, E, C) \circ S) \leftarrow \text{ alteprop}(A, E, C), \]
\[ \text{eterm}(S), \]
\[ S \neq AC_1(A, E, C) \circ R. \]

Clause (5.7) is then modified as follows:

\[ \text{selection}(A, S, I) \leftarrow \text{eterm}(S), \]
\[ \neg \text{overrepresented}(A, S, I), \]
\[ \neg \text{underrepresented}(A, S, I). \]

To summarize, employing the modified equational logic program avoids the problem of floundering derivations. Moreover, all derivations which occur when deciding entailment of a v-proposition via the query \( \leftarrow \text{satisfiable} \) in conjunction with the corresponding clause (5.22) are guaranteed to terminate. Hence SLDENF-resolution can now be applied as a sound and complete calculus for the equational logic program encoding domains specified in \( NCC \). In addition, finiteness of derivations shows that we have obtained a decision procedure for entailment in our high-level action semantics.

6. SUMMARY

We have presented formalisms for intentionally specifying actions so as to have nondeterministic effects, and for extracting as much as possible consistent and reasonable information from contradictory representations of dynamic systems by interpreting them so as to be implicitly nondeterministic. Our resulting language \( NCC \) allows the representation of nondeterministic concurrent actions in dynamic systems and the resolution of conflicts.

We have furthermore developed a sound and complete encoding of domain specifications in \( NCC \) in terms of equational logic programming and, in so doing, have provided an instrument for automated reasoning about these domains. Generally, the translation of high-level languages like \( NCC \) into different ap-
proaches designed for reasoning about dynamic systems, actions, and change allows to compare the expressiveness and limitations of these approaches in a precise and uniform way. As argued in, e.g. [12, 32, 33, 36, 37], doing this is in favorable contrast to the traditional way of justifying new approaches with reference to a few standard examples such as the blocksworld or the Yale Shooting scenario and its enhancements. To this end, translations of $\mathcal{A}$ and some of its extensions, for instance, into a number of existing action calculi have recently been used for the purpose of comparison and to study their range of applicability (see, e.g. [12, 11, 20, 5, 23, 10, 22]). The extension defined in this paper, $\mathcal{A}_{\text{NCC}}$, may therefore be employed as a formal, high-level action semantics for action calculi which successfully deal with domains involving explicit and implicit indeterminism as well as concurrency.

Our formalisms are based on the Action Description Language $\mathcal{A}$. Two recent extensions of $\mathcal{A}$, namely, first-order-fluents [11] and indirect effects [23], are not subsumed by our approach. On the other hand, the fluent calculus, which we have used here, has already been extended to successfully cope both with non-propositional fluents and with the ramification problem [41]. Hence we have good reasons to assume that the logic program presented in this paper can be extended to form an adequate encoding of a high-level action semantics including $\mathcal{A}_{\text{NCC}}$ and indirect effects.

APPENDIX A: THE COMPLETED EQUATIONAL LOGIC PROGRAM

Let $D$ be a domain description in $\mathcal{A}_{\text{NCC}}$, and let $(P_D, AC1)$ be the corresponding equational logic program. The completion $P^*_D$ of $P_D$ consists of the following first-order formulas.

Let $F_D$ be the underlying set of fluent names then

$$\forall X \left( \text{complement}(X) \leftrightarrow \bigvee_{f \in F_D} X = f \circ f \right) \quad (5.3^*)$$

completes $\text{FLUENT}_D$.

Let $\{a_i \text{ causes } e_i \text{ if } C_i \} \ 1 \leq i \leq k$ be the set of all strict e-propositions in $E_D$ ($k \geq 0$) then

$$\forall A, E, C \left( \text{eprop}(A, E, C) \leftrightarrow \bigvee_{i=1}^k \left( A = \mu(a_i) \land \right. \left. E = \tau(e_i) \land C = \tau(C_i) \right) \right) \quad (5.4^*)$$

completes $\text{EPROP}_D$. 

The authors want to thank two anonymous reviewers for their exceptionally detailed comments and helpful suggestions. The first author acknowledges support from the German Research Community (DFG) within project MPS under grant no. Ho 1294/3-3.
Let \( \{a_i \} \) alternatively causes \( E_i \) if \( C_i |1 \leq i \leq k \) be the set of all alternative e-propositions in \( E_D \) \((k \geq 0)\) then

\[
\forall A, E, C \left( \text{alteprop}(A, E, C) \leftrightarrow \bigvee_{i=1}^{k} (A = \mu(a_i) \land E = \tau(E_i) \land C = \tau(C_i)) \right)
\]

(5.5*)

completes \( \text{ALTEPROP}_D \).

The completion of clause (5.6) and (5.7) is as follows:

\[
\forall I, A, H(\text{action}(I, A, H) \leftrightarrow \exists S(\text{selection}(A, S, I) \land
\neg \text{impossible}(H, I, A, S) \land
\neg \text{unfounded}(H, I, A, S) \land
\neg \text{inconsistent}(H)))
\]

(5.6*)

\[
\forall A, S, I(\text{selection}(A, S, I) \leftrightarrow \neg \text{overpresented}(A, S, I) \land
\neg \text{underrepresented}(A, S, I)).
\]

(5.7*)

The three clauses shown in (5.8) are completed by

\[
\forall A, I, S(\text{overrepresented}(A, S, I) \leftrightarrow
\exists R, X(\ X \neq \emptyset \land
S = X \circ R \land
\neg \text{underrepresented}(A, S, I)))
\]

(5.8*)

\[
\forall S, X, Y(\text{underrepresented}(X, S, Y) \leftrightarrow
\exists A, B, C, E, J(\ X = A \circ B \land
Y = C \circ J \land
\text{alteprop}(A, E, C) \land
\neg \text{represented}(A, S)))
\]

\[
\forall A, S(\text{represented}(A, S) \leftrightarrow \exists C, E, R. S = (A, E, C) \circ R).
\]

The completion of \( \text{impossible} \) is

\[
\forall A, I, S, X(\text{impossible}(X, I, A, S) \leftrightarrow
\exists F, G, H(\ X = F \circ H \land
\text{overruled}(F, I, \emptyset, A, S) \land
\text{complement}(F \circ G) \land
\neg \text{overruled}(G, I, \emptyset, A, S)))
\]

(5.9*)
and the completion of the two clauses for overruled is the formula

\[ \forall A, S, X, Y, Z (\text{overruled}(X, Y, A, Z, S) \iff \exists B, C, D, F, G, H, J (X = F \circ H \land Y = C \circ J \land Z = A \circ B \circ D \land \text{eprop}(A \circ B, G, C) \land B \neq \emptyset \land \text{complement}(F \circ G) \land \neg \text{overruled}(G, C \circ J, A \circ B, A \circ B \circ D, S)) \] (5.10*)

Completing (5.11) yields

\[ \forall A, S, X, Y (\text{unfounded}(X, Y, A, S) \iff \exists F, G, H, I (X = F \circ H \land Y = G \circ I \land \text{complement}(F \circ G) \land \neg \text{overruled}(G, G \circ I, \emptyset, A, S)). \] (5.11*)

The four clauses used to express consistency are completed as follows:

\[ \forall I (\text{inconsistent}(I) \iff \exists G, H (I = G \circ G \circ H \land G \neq \emptyset) \] (5.12*)

\[ \exists F, G (I = F \circ G \land \text{complement}(F)) \]

\[ \exists F, G (\text{complement}(F \circ G) \land F \neq \emptyset \land G \neq \emptyset \land \neg \text{holds}(F, I) \land \neg \text{holds}(G, I)) \]

\[ \forall F, I (\text{holds}(F, I) \iff \exists H, I = H \circ F). \]
Clause (5.13) is completed by

\[ \forall I, G, L (result(I, L, G) \leftrightarrow (L = [ ] \land I = G)) \land \exists A, H, P (L = [(A, H)]P) \land action(I, A, H) \land result(H, P, G)) \]  \hspace{1cm} (5.13*)

Finally, let \( \{l_i \text{ after } [a_{i1}, \ldots, a_{im_i}] \mid 1 \leq i \leq n \} \) be the set of all \( v \)-propositions in \( V_D \) \( (n \geq 0) \) then

\[ \forall I, X_1, \ldots, X_\beta (model(I, X_1, \ldots, X_\beta) \leftrightarrow \neg inconsistent(I) \land \bigwedge_{i=1}^{n} result(I, [(\mu(a_{i1}), X_{a_{i1}}), \ldots, (\mu(a_{im_i}), X_{a_{im_i}})], \tau(i)) \circ G_i) \]  \hspace{1cm} (5.14*)

completes (5.14).

REFERENCES


