Action Programming In Rewriting Logic

Lenz Belzner

LMU Munich, Chair for Programming and Software Engineering
(*e-mail:* belzner@pst.ifi.lmu.de)

submitted 10 April 2013; accepted 23 May 2013

Abstract

Action programming for autonomous agents is a valuable technique when it comes to construction and evaluation of behavioural alternatives due to changing and highly dynamic environments. This paper discusses the application of rewriting logic to formalize high-level action programming in domains with non-deterministic actions, concurrency and relational knowledge. It is shown that rewriting logic semantics and associative-commutative rewriting can be effectively applied to express non-determinism and concurrency as well as to solve the frame problem in the context of action programming in dynamic domains.

A relational term-structure for the state-action space is defined that allows to represent state properties and the actions affecting them as a single term. Using this unified representation, the interplay between an agent's behaviour and the domain can be modelled by repeatedly choosing new actions according to the agent's action program and reducing the resulting state-action term to a form that cannot be further rewritten in order to determine the effect of the actions on the domain. This approach is formalized in rewriting logic by defining syntax and semantics for state-action space construction and rewriting with regards to a given action program, thus allowing for the deduction of sequential action alternatives for agents in non-deterministic and concurrent domains, and the specification of action effects and domain dynamics in the form of rewrite laws is discussed.

The presented rewrite theory for action programming can be directly translated into an executable interpreter by implementing it in the language MAUDE. Thus, several of MAUDE's properties are exposed to the resulting action programming language and the interpreter, e.g. explicit sorting, sub-sorting and operator polymorphism allowing for structured domain specification and order-sorted unification, its object-layer and meta-programming facilities as well as support for domain specific syntactic representation of fluents and actions due to MAUDE's user-definable syntax.

KEYWORDS: Rewriting Logic, Action Programming, Non-Determinism, Concurrency

1 Introduction

Many modern software systems are designed to act in highly dynamic and non-deterministic environments. In many cases, there may be no possibility for developers to change the behaviour of the system once it has been deployed. Thus, the design of adequate controllers that allow software agents to react properly to most or all of the situations they find themselves in for environments that exhibit manifold dynamics and change becomes a challenging task. While system designers could specify controllers for each possible situation that could arise, this is an error prone task that may be impossible to realize in many cases due to the sheer number of situations the agent should be able to cope with.

This setting results in two main requirements to agent controllers that are designed to work in

such dynamic environments: First, they have to be able to construct and evaluate various behavioural alternatives according to the current state of their environment, and second, in order to do so, they need some internal model of the environment to be able to reason about its properties. Classical planning approaches offer these capabilities, but may not be capable to deal with the sheer amount of different configurations of the environment, especially in domains where an agent needs to react to a new situation as fast as possible.

Action programming (or high-level agent programming) allows a programmer to specify only sketches of a program, that may exhibit non-deterministic choice of subroutines when needed. Such a high-level program is designed to work with a model of the environment that exposes to agents expert knowledge about the preconditions, effects and success probability of their actions, goals in terms of rewards or exogenous events that could occur out of the scope controlled by the agent. An agent's behaviour is only partially specified deterministically in a high-level program, restricting the points in the trace of actions where it is required to make a decision by itself. The specified model of the environment together with the only partially non-deterministic high-level program thus constrain the space of action alternatives, which is a necessary requirement for performing reasonable search of alternatives in environments that are subject to highly dynamic change.

Another requirement that is becoming more and more important with the rise of multi-agent systems is the ability to cope with the concurrent execution of actions. As intended actions of agents may use constrained resources or even be contradictory, autonomous agents should be able to reason about the outcome of their choices w.r.t. actions that could be executed by other agents at the same time. Concurrency is known to be a subtle but important issue, and a high-level programming language should allow to express these subtleties clearly.

A formal, logical specification of a high-level programming language is beneficial when it comes to reasoning capabilities, and the fact that controllers designed in it should specify the behaviour of agents that are to act autonomously renders provability of behavioural properties extremely valuable. *Rewriting logic* is a logical framework that has clear semantics for concurrency, and can be used to express non-determinism in a natural and elegant way. Thus, it lends itself naturally to the formal definition of a high-level programming language that meets the requirements outlined in this section. Also, theories specified in rewriting logic can be directly implemented in the programming language MAUDE, rendering formal specifications executable in a straightforward manner. Rewriting logic and the MAUDE language support features like sorting, sub-sorting, polymorphism, object-orientation and reflection, which are not provided explicitly by existing action-programming languages like GOLOG or FLUX. Thus, the field of high-level programming is likely to benefit from these features by being formalized in rewriting logic.

The rest of this paper is organized as follows: Section 2 discusses rewriting logic and action programming. In section 3, a framework for action programming in rewriting logic is presented, syntax and semantics for an action programming language based on the approach are defined, and specification of non-deterministic, concurrent domains is discussed. Section 4 summarizes the results and outlines directions for further work.

2 Foundations & Related Work

2.1 Rewriting Logic

The core element to specify non-determinism and concurrency in rewriting logic are *rewrite theories* (Meseguer 1992; Clavel et al. 2007). They are defined as a four-tuple $(\Sigma, E \cup A, \phi, R)$, where Σ contains sorts, sub-sorts, kinds (i.e. sorts with error types) and (possibly polymorphic) operators, $E \cup A$ is an equational theory defined by explicitly specified equations for terms in Σ and axioms as associativity, commutativity, idempotency or identity defined for operators in Σ, ϕ is the set of *frozen* operator arguments that will not be rewritten through the application of rewrite laws, and R is a collection of (possibly conditional) rewrite laws. Rewrite theories define a logical theory with deduction rules given by R. They represent a transition system, where states are terms in Σ, and transition rules are specified by the rewrite laws in R.

A powerful concept of rewriting logic is term matching *modulo axioms*. If, for example, an operation $\circ$ is specified as associative and commutative, this renders terms that are constructed by means of $\circ$ effectively into a *multiset*, i.e. a data structure where terminal symbols may occur multiple times and where ordering of symbols does not affect the term's equivalence class. Another matching concept used in rewriting logic is matching *with extension*, that allows to match terms partially. With $\circ$ being associative and commutative, $a \circ b$ would match modulo axioms with extension the term $b \circ c \circ a$, as $a \circ b$ matches with extension the equivalence class of $b \circ c \circ a$ that is defined through the axioms for $\circ$ (and that effectively renders equal the terms $b \circ c \circ a$ and $a \circ b \circ c$). Note that the concept of matching modulo axioms with extension is also used for order-sorted unification in rewriting logic (Meseguer et al. 1989; Hendrix and Meseguer 2012). A more detailed description of these matching concepts is found for example in Meseguer (1992) or Eker (1996).

Rewrite rules that specify the transitions between states (i.e. terms) are of the form

$$label : \ t \ \longrightarrow \ t' \ \textbf{if} \ Conditions$$

where t and t' are terms of the same kind that may contain free sorted variables, t specifying the term (or a part thereof when matching modulo axioms) to be rewritten, and t' is the term (or part thereof) resulting from the application of the rewrite law. The label may also be omitted. Conditions are optional and can have four different forms:

- A pure equational condition constraining a variable value through an equation $u = u'$, u and u' being terms of a sort used in t. Equality is computed according to $E \cup A$ of the rewrite theory.
- A matching condition $u := u'$ that constrains the syntactic structure of a variable in t (again modulo axioms with extension). Note that u may contain free variables not occurring in t but which may occur in t' or other conditions, so that this condition can be used to define variables with a local scope.
- A sort or kind test, denoted $s : Sort$.
- A rewrite condition $u \longrightarrow u'$, that evaluates to true if (multiple) application of rewrite laws from the rewrite theory to the term u result in the term u'. As for matching conditions, new variables not occurring in t may be introduced, this time in the term u'.

Rewriting can take place if a term to be rewritten matches modulo axioms with extension the left-hand term t of a specified rewrite law in R where all conditions evaluate to true. Note that, for term patterns typically met in practice, one step of associative rewriting can be achieved

in constant time (independent of term size), and that complexity for associative-commutative rewriting is $O(log\ n)$ (Eker 2003).

As portions of a term can be rewritten simultaneously, concurrency can be modelled in rewriting logic very naturally. Consider two rewrite laws whose left-hand terms match with extension a term to be rewritten. Both laws can be applied simultaneously, thus concurrently rewriting the term (or, in the context of a transition system, transitions are firing concurrently). Non-determinism is achieved if the same or overlapping portions of a term match left-hand terms of rewrite laws. On the other hand, not both rewrite laws may apply simultaneously. In this case, each of them will be applied to the subject term for its own, resulting in two new terms that can be further rewritten. This can be seen as non-deterministic rewriting of the original term. Another way to specify non-determinism is by specifying a commutative operator, e.g. *or*, and to specify a rewrite law $X\ or\ Y\ \rightarrow\ X$, that yields two possible results of rewriting a term, namely X and Y, as the terms $X\ or\ Y$ and $Y\ or\ X$ belong to the same equivalence class due to the commutativity of *or*. For an in-depth discussion of rewriting logic semantics for non-determinism and concurrency, refer to Meseguer (1992).

In rewriting logic, logical deduction and program execution are considered semantically equal, a fact that is exploited by the programming language MAUDE (Clavel et al. 2002; Clavel et al. 2007). MAUDE allows to specify rewrite theories as outlined in this section directly as a program, and offers various commands to compute results that are achieved through rewriting, thus rendering theoretical results as presented in section 3 executable in a fairly straightforward manner. For example, MAUDE's *search* command searches for all possible ways to rewrite a term, thus incorporating results that arise due to non-deterministic and concurrent rewriting as discussed above. MAUDE has already been used successfully to implement prototypically the agent programming language 3APL (van Riemsdijk et al. 2006).

2.2 Action Programming

Action programming is "the art and science of devising high-level control strategies for autonomous systems which employ a mental model of their environment and which reason about their actions as a means to achieve their goals" (Thielscher 2008). Action programming provides to agents a cognitive model which can be used to construct and evaluate behavioural alternatives in highly dynamic environments.

The core element of action programming is the notion of an action. Typically, actions are defined by means of several concepts:

- A *precondition*, that states which properties of the environment have to hold so that the particular action is executable.
- An *effect*, that states which properties of the environment change due to action execution and how they do so. Actions may have non-deterministic effects, leading to different situations depending on the outcome of their execution.

In order to evaluate the consequences of action execution in a qualitative way, some decision-theoretic values may be specified as well:

- The *probability* that a certain effect occurs due to action execution, if the action has non-deterministic effects.
- A *reward* that an agent obtains if it finds itself in a certain situation or executes a certain action in a particular state.

To allow for specification of preconditions and effects of an action as outlined above, there needs to be a notion of *state*, so that changing properties (called *fluents*) of the environment can be represented. This representation of state is updated by an agent through sensing, and reasoning is performed on the agent's internal, mental model of the environment. Some formalisms that have been successfully applied to represent the environment in action programming are the *situation calculus* (Reiter 2001) and the *fluent calculus* (Thielscher 1998), both using axioms in first-order logic to specify actions and domain dynamics.

Typical action programming languages like GOLOG (and its variants, e.g. INDIGOLOG) or FLUX (Reiter 2001; Giacomo et al. 2009; Thielscher 2005) provide several operations to define agent behaviour, like sequential execution of subroutines (i.e. actions or other programs) and non-deterministic operations like choice of subroutines or arguments. These non-deterministic operations lead to the construction of behavioural alternatives, which are subsequently being evaluated. Consider for example a sequential ;-operator, a non-deterministic choice operator # and the actions a and b respectively. The program $a;a\#b$ would result in two possible alternative action traces: $a;a$ and $a;b$. The logic that is used to describe the effects of these traces should enable the agent to reason about the two alternatives. How this is achieved in rewriting logic will be discussed in section 3.

The two languages mentioned have been implemented as PROLOG interpreters, where reasoning is performed on a theory formulated as a set of Horn clauses. Sorts can be added by specifying corresponding predicates, but unification is usually performed on a purely syntactical level. Although performance gain through order-sorted unification has been proven in theory for the situation-calculus (Gu and Soutchanski 2009), the authors did not yet report about an implementation of their work. Also, there is no syntactic freedom in the specification of domains as PROLOG does not support user-definable syntax, and no explicit object layer has yet been integrated to GOLOG or FLUX. As MAUDE natively provides all these features, it seems a valuable approach to implement action programming in rewriting logic (thus providing a specification for a MAUDE implementation, see section 2.1).

3 Action Programming In Rewriting Logic

The semantics of rewriting logic for concurrency and non-determinism that can be expressed in terms of corresponding rewrite laws can naturally be exploited to implement action programming in dynamic domains. To model the dynamics of actions and change, they will be specified in the form of rewrite laws expressing the preconditions and consequences of actions that can be performed by agents.

The key idea is to represent the state in a term that can be *tensed* by adding actions to it according to program execution, thus creating a representation of the current *state-action space*. In terms of rewriting logic, this means that the term can subsequently be rewritten according to domain dynamics specified in the form of rewrite laws to one (or more, if non-determinism takes place) term(s) that cannot be further reduced, i.e. *relaxing* the state-action space again. Thus, action programming in rewriting logic naturally models a cycle of tension (actions that are to and can be executed are part of the state-action space) and relaxation (no executable actions are in the state-action space). The execution cycle of action programming in rewriting logic is illustrated in figure 1. A formal definition of tensed and relaxed state-action spaces is given in section 3.1.

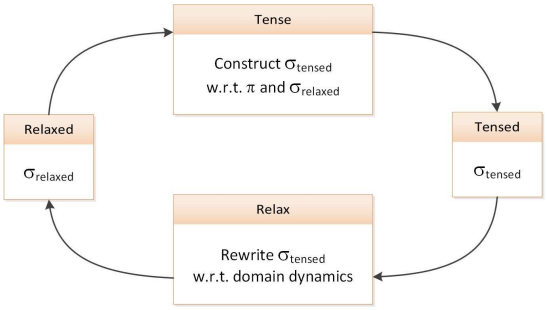

Fig. 1. State-action space tension and relaxation cycle.

$$\Sigma := \varepsilon \,|\, noop \,|\, f(\vec{x}) \,|\, a(\vec{x}) \,|\, \Sigma \wedge \Sigma \,|\, \Sigma \vee \Sigma$$

Fig. 2. State-action space syntax.

3.1 Syntax

The syntactic definition of the *state-action space* Σ is given in figure 2. ε stands for the empty space, *noop* represents a no-operation of an agent. $f(\vec{x})$ are *fluents*, domain specific values of the state that change due to the dynamics of actions, which are represented through the $a(\vec{x})$. The *state space* consists of all formulas in Σ that only contain fluents; *action space* formulas consist of actions only. Both fluents and actions can be parametrized, the parameters are usually of sorts specified for a specific domain; thus, the state-action space can be expressed relationally, resulting in a concise representation that is highly beneficial from a computational perspective. Fluents and actions are terminal symbols, thus allowing for syntactic freedom in their concrete representation. The operations $\wedge$ and $\vee$ denote logical conjunction and disjunction of state-action space terms. Note that terms from Σ can always be transformed to a disjunctive normal form by specifying an according distributivity equation. Note also that there is no explicit form of negation, as this operation gives rise to semantical consequences complicating the process of reasoning about states. Thus, negation is only implicitly supported by assuming all fluents that are not explicitly represented in the state-action term as not being true.

To support associative-commutative matching, rewriting and unification on state-action terms as discussed in section 2, the following axioms are defined for $\wedge$ and $\vee$:

- Both $\wedge$ and $\vee$ are associative and commutative with identity element ε.
- $\wedge$ has a lower precedence than $\vee$.

Finally, formal definitions for tensed and relaxed state-action terms are given.

Definition (tensed and relaxed state-action terms): Let R be a set of rewrite laws. A state-action term σ in Σ is called *tensed* if there is a rewrite law in R that can be applied to σ. Otherwise, it is called *relaxed*.

Programs are terms in Π that will tense the state-action space by injecting actions to it. The syntactical definition of Π is given in figure 3. *nil* denotes the empty program, $a(\vec{x})$ represents

$$\Pi := nil \mid noop \mid a(\vec{x}) \mid \Pi;\Pi \mid \Pi\#\Pi \mid \Pi \parallel_{det} \Pi \mid \Pi \parallel_{ndet} \Pi \mid \Pi \parallel_{true} \Pi \mid \textbf{if } \Sigma \textbf{ then } \Pi \textbf{ else } \Pi \textbf{ end}$$

Fig. 3. Action program syntax.

actions. Note that the $a(\vec{x})$ are the same as for specifying terms in Σ. The operator ; stands for sequential execution, # is non-deterministic choice of programs, $\parallel_{det}$ and $\parallel_{ndet}$ represent deterministic and non-deterministic interleaving of programs, and $\parallel_{true}$ denotes true concurrency without interleaving. Finally, there is a conditional branching operator.

As for states, there are mathematical axioms defined for various operators to simplify specification of their semantics in rewriting logic:

- ; and $\parallel_{det}$ are associative with identity element *nil*.
- #, $\parallel_{ndet}$ and $\parallel_{true}$ are associative and commutative with identity element *nil*.
- Furthermore, # is idempotent, rendering $\pi\#\pi = \pi$ for any program π.

3.2 Semantics

The action programming cycle represented in figure 1 is specified in rewriting logic by two operations *process* and *exec*. *process* transforms a given action program, eventually yielding various results due to non-deterministic choice or conditional branching w.r.t. the current state-action space. *exec* is responsible for the orchestration of tension and relaxation of the state-action space through programs, representing the dynamics of the approach. The semantics of these operations in terms of rewriting logic equations and rewrite laws are given in section 3.2.1 and section 3.2.2. Section 3.2.3 discusses the representation of domain dynamics in terms of rewrite laws. In section 3.2.4, a short example of application is shown.

3.2.1 Action Space Construction

To determine the effects on a state σ of the action alternatives provided by a program π, the current state-action space is constructed w.r.t. π, rendering it tensed (i.e. the term can be rewritten w.r.t. rewrite laws that specify domain dynamics). This tension of the state-action space means that one needs to determine which actions can be executed in the next step of the program π. This computation is specified through an operation *process*, that takes the current program π and the current state-action space σ as parameters, and transforms π to the form $\sigma_\alpha;\pi'$ (w.r.t. σ in the case of conditional branching), where σ_α is a term from Σ containing a conjunction of actions, thus representing the current action space; and π' is the tail of the program to be executed. The action space σ_α will subsequently be injected to the current state, thus rendering it tensed and triggering its rewriting to a relaxed form through rewrite laws which are specifying domain dynamics (see section 3.2.3).

The semantics of *process* for transforming programs from Π are defined in figure 4, excluding branching, which is discussed separately. Note that there is only one rewrite law specified for the processing of a program choice, which is sufficient to capture its non-deterministic semantics as the operator # has been specified as commutative (see section 2). Deterministic and non-deterministic interleaving are specified by analogous rewrite laws; as $\parallel_{ndet}$ has been specified

$$process : \Pi \times \Sigma \rightarrow \Pi$$

$$\begin{array}{lcl}
process(nil,\ \sigma) & = & noop;nil \\
process(a(\vec{x}),\ \sigma) & = & a(\vec{x});nil \\
process(\pi;\pi',\ \sigma) & = & \sigma_\alpha;\pi'';\pi' \\
 & & \textbf{if } process(\pi,\ \sigma) \rightarrow \sigma_\alpha;\pi'' \\
process(\pi \# \pi',\ \sigma) & \rightarrow & process(\pi,\ \sigma) \\
process(\pi \parallel_{det} \pi',\ \sigma) & \rightarrow & \sigma_\alpha;(\pi' \parallel_{det} \pi'') \\
 & & \textbf{if } process(\pi,\ \sigma) \rightarrow \sigma_\alpha;\pi'' \\
process(\pi \parallel_{ndet} \pi',\ \sigma) & \rightarrow & \sigma_\alpha;(\pi' \parallel_{ndet} \pi'') \\
 & & \textbf{if } process(\pi,\ \sigma) \rightarrow \sigma_\alpha;\pi'' \\
process(\pi \parallel_{true} \pi',\ \sigma) & \rightarrow & \sigma_\alpha \wedge \sigma_\alpha';(\pi'' \parallel_{true} \pi''') \\
 & & \textbf{if } process(\pi,\ \sigma) \rightarrow \sigma_\alpha;\pi'' \\
 & & \textbf{and } process(\pi',\ \sigma) \rightarrow \sigma_\alpha';\pi'''
\end{array}$$

Fig. 4. Action space and program tail construction through *process*.

$$\begin{array}{lcl}
process(\textbf{if } \sigma_{cond} \textbf{ then } \pi \textbf{ else } \pi' \textbf{ end},\ \sigma) & \rightarrow & process(\pi[\vec{x}/substitution],\ \sigma) \\
 & & \textbf{if } xmatch(\sigma_{cond},\ \sigma) \rightarrow substitution \\
process(\textbf{if } \sigma_{cond} \textbf{ then } \pi \textbf{ else } \pi' \textbf{ end},\ \sigma) & \rightarrow & process(\pi',\ \sigma) \\
 & & \textbf{if } xmatch(\sigma_{cond},\ \sigma) \rightarrow no\ match
\end{array}$$

Fig. 5. *process* for branches.

commutative, this leads to different semantics though. True concurrency is represented through the construction of an action space term by means of the $\wedge$-operator and is further discussed in section 3.2.3.

Conditions that specify the branching of programs w.r.t. to the current state-action space are themselves terms in Σ. Assuming a closed world and complete knowledge, a condition holds in a state if the condition term *matches with extension modulo axioms* the current state term for some substitution of variables (see section 2.1). Note that state-action terms are specified as multisets (associative, commutative and with identity element, see section 3.1). This matching attempt can be seen as a query (the condition) to a database (the current state-action space). Note also that unification of variables in order to find matching substitutions can be performed order-sorted, as fluents and actions in the state-action space can have sorted variables, and is done modulo a specified rewrite theory, thus also considering domain-specific sorts and sub-sorts (Meseguer et al. 1989; Hendrix and Meseguer 2012). If a matching substitution is found, it is applied to the variables in the true-branch, which will be subsequently processed. Otherwise, the false-branch is to be processed further. Figure 5 shows the semantics of branching.

3.2.2 State-Action Space Tension & Relaxation

The general approach to action programming in rewriting logic by means of tensing and relaxing the state-action space as illustrated in figure 1 is specified by the operation *exec*, that takes the following parameters:

$$exec : \Sigma \times \Pi \times \Pi \rightarrow \Sigma \times \Pi$$

$$\begin{array}{lcl} exec(\sigma,\ nil,\ \tau) & \rightarrow & \sigma \times \tau \\ exec(\sigma \vee \sigma',\ \pi,\ \tau) & \rightarrow & exec(\sigma,\ \pi,\ \tau) \\ & & \textbf{and } \pi \neq nil \\ exec(\sigma,\ \pi,\ \tau) & \rightarrow & exec(\sigma_{relaxed},\ \pi',\ \tau;\sigma_\alpha) \\ & & \textbf{if } process(\pi,\ \sigma) \rightarrow \sigma_\alpha;\pi' \\ & & \textbf{and } \sigma \wedge \sigma_\alpha \rightarrow \sigma_{relaxed} \\ & & \textbf{and } \sigma' \wedge \sigma'' := \sigma \\ & & \textbf{and } \pi \neq nil \end{array}$$

Fig. 6. State-action space tension and relaxation through *exec*.

- A state-action term σ in Σ, representing the current system state.
- A program term π in Π, representing the program to be executed.
- A program term τ in Π, representing the action trace that results in σ.

From these parameters, *exec* deduces all possible final state-action terms that can be reached and the corresponding traces leading there by repeated tension and relaxation of the state-action space w.r.t. the given program and the specification of domain dynamics in form of rewrite laws. This deduction is realized through the following steps for conjunctive state-action terms:

- First, the action space σ_α and the remaining program π' are constructed by applying the *process*-operation to π and σ. Note that σ_α is a conjunctive term in Σ where only actions or *noop* occur as terminal symbols. Also note that rewriting *process* may yield different results due to non-deterministic choice of actions or argument bindings (through branching).
- Then the current state-action space σ is tensed by conjoining it with the action space σ_α, possibly resulting in a term that can be rewritten by rewrite laws from the domain specification. Possible state-action space relaxations $\sigma_{relaxed}$ are constructed by the application of these rewrite laws through evaluation of the condition $\sigma \wedge \sigma_\alpha \rightarrow \sigma_{relaxed}$. Note that the specification of non-deterministic action effects (see section 3.2.3) may lead to various terms for $\sigma_{relaxed}$.
- Finally, the resulting $\sigma_{relaxed}$ and the tail π' of the original program are passed again as parameters to *exec*, thus executing the next program step. Note that the trace τ is updated to a new sequence $\tau;\sigma_\alpha$ containing the action space that was currently used to tense σ.

Deduction terminates when a *nil*-program is passed to *exec*, which will always occur for terminating programs π because of the semantics of *process*. If a disjunctive state-action term σ is passed as argument, *exec* will be executed for all disjuncts of σ. Again, a single rewrite law is sufficient, because the axioms specified for state-action term disjunction allow for associative-commutative rewriting (see section 2). See figure 6 for the specification of the semantics of *exec* in terms of rewrite rules.

3.2.3 Domain Dynamics As Rewrite Laws

A rewrite law specifying an action in terms of precondition and effect (see section 2) has the form $\sigma_{prec \cup affected} \wedge \sigma_\alpha \rightarrow \sigma_{unchanged \cup effect}$ **if** *Conditions*. In particular:

$$T : Truck,\ C : City,\ B : Box,\ P,R : Float$$

$$truckIn(T, C) \wedge boxLoaded(B, T) \wedge reward(R) \wedge prob(P) \wedge unload(T, B) \rightarrow truckIn(T, C) \wedge boxIn(B, C) \wedge reward(R + 1.0) \wedge prob(P * 0.9)$$

$$truckIn(T, C) \wedge boxLoaded(B, T) \wedge reward(R) \wedge prob(P) \wedge unload(T, B) \rightarrow truckIn(T, C) \wedge boxLoaded(B, T) \wedge reward(R) \wedge prob(P * 0.1)$$

Fig. 7. Domain law example from BOXWORLD.

$$T, T' : Truck,\ C, C', C'' : City$$

$$truckIn(T, C) \wedge truckIn(T', C'') \wedge driveTo(T, C') \wedge driveTo(T', C') \rightarrow truckIn(T, C') \wedge truckIn(T', C') \wedge trafficJam(C')$$

Fig. 8. True concurrency example.

- $\sigma_{prec \cup affected}$ is a term in Σ containing the precondition and affected fluents for the action(s) specified in σ_α.
- σ_α is the action (or a conjunction of multiple actions) whose effect is specified by the rewrite law.
- $\sigma_{unchanged \cup effect}$ consists of any fluents specified in $\sigma_{prec \cup affected}$ that remain unchanged or that are changed or added to the state-action space as a consequence of executing the action(s) specified in σ_α.

Non-deterministic action effects can be captured by specifying multiple rewrite laws for a term $\sigma_{prec \cup affected} \wedge \sigma_\alpha$ or by specifying the effect as a disjunctive state-action term.

Probabilities for a certain effect to take place can be deduced by specifying a fluent $prob(P : Float)$ and altering the value of P in $\sigma_{unchanged \cup effect}$ according to transition probability. In the same manner rewards can be defined for transitions to certain states, e.g. by specifying a fluent $reward(R : Float)$ and updating its value in the corresponding rewrite laws. Note that with these representations of probability and reward, a relational Markov decision process can be specified straightforwardly in the presented approach.

An example for a rewrite law specifying an action with non-deterministic effects, transition probabilities and reward accumulation is given in figure 7. It is taken from the famous BOXWORLD domain, and states that a truck can unload a box in a city if the truck has loaded the box and is currently located in the city. While this operation succeeds with a probability of 90% giving a reward of 1, in the remaining 10% of cases it will consume the *unload* action without any effect. Note that this non-determinism could also be expressed in a single rewrite law with a disjunctive right-hand side term.

Note that, as state-action space terms are constructed by associative and commutative operations, the exploitation of matching and rewriting modulo axioms together with the specification of any fluents that remain unchanged leads to a solution of the frame problem (Mccarthy and Hayes 1969; Martí-Oliet and Meseguer 1999), rendering unchanged all parts of the state-action space that are not explicitly specified in the rewrite law. Also this property ensures that all possible partial matches of the state-action space with the rewrite law's left-hand term will be deduced and further computed.

truckIn(T, C) $\wedge$ *driveTo(T, C')* $\rightarrow$ *truckIn(T, C')*
truckIn(T, C) $\wedge$ *boxIn(B, C)* $\wedge$ *load(T, B)* $\rightarrow$ *truckIn(T, C)* $\wedge$ *boxLoaded(B, T)*

Fig. 9. Example domain laws for actions *driveTo* and *load*.

σ = *truckIn(t, rome)* $\wedge$ *boxLoaded(b, truck)* $\wedge$ *prob(1.0)* $\wedge$ *reward(0.0)*
π = *unload(t, b) # load(t, b) # driveTo(t, paris)*;
if *boxLoaded(t, b)* ***then*** *unload(t, b)* ***else*** *driveTo(t, paris)*

Fig. 10. Example state σ and program π.

To specify the effects of actions that are truly concurrent, one can specify the corresponding rewrite law with a conjunction of action terminals, so that only state-action terms constructed by means of truly concurrent programs will match the corresponding law's left-hand side term. An example is given in figure 8; it states that if two trucks drive to the same city concurrently, this will result in a traffic jam.

3.2.4 An example

As an example, consider the rule for action *unload* from figure 7 together with the rules for actions *driveTo* and *load* as shown in figure 9. Note that the latter two actions will always succeed. Consider that a truck *t* is situated in Rome, and it has a box *b* loaded. Also consider a program that allows an agent controlling the truck to either try to unload the box (which is not guaranteed to succeed), try to load it, or to drive to Paris. Next, the agent should check whether the truck still has the box loaded. If so, it will again try to unload, otherwise, it will drive to Paris. This situation can be encoded according to section 3.1 in terms of a state σ and a program π as shown in figure 10. By rewriting the term $exec(\sigma, \pi, nil)$ as specified in section 3.2, all possible resulting states after executing program π together with the trace of actions leading to each particular state can be deduced.

Consider now that the agent is supposed to achieve a certain goal, for example to deposit box *b* in Rome. Action traces resulting from executing π that will lead to states satisfying this goal can be deduced by invoking MAUDE's conditional *search* command, specifying the goal as condition in form of a term that all rewriting results have to match. In the example case, this would result in the traces *unload(t, b);unload(t, b)* if the first *unload* action fails to have an effect and the second one succeeds (see figure 7), and *unload(t, b);driveTo(t, paris)* if the first execution of *unload* succeeds. The probabilities that online execution of these traces will in effect result in states where the goal is reached is deduced in terms of the *prob* fluent, yielding a probability of 0.09 for the first trace and 0.9 for the second one, respectively. From this information, an agent could for example deduce that when first executing *unload*, the probability of reaching a goal state when subsequently executing the rest of π (also checking whether *unload* succeeded) equals to 0.99. Reward information could be exploited in a similar manner as well.

The traces *driveTo(t, paris);unload(t, b)* and *driveTo(t, paris);driveTo(t, paris)*, while being considerable trace alternatives according to π, will not lead to a goal state in any case; con-

sequently, these traces will not be a result of performing a conditional search of rewrites of the original *exec* term. Also, all traces beginning with the *load* action will not be deduced, as the left-hand side of the rewrite law specifying the precondition for executing a *load* action (see figure 9) does not match (modulo axioms with extension) the initial state term σ.

4 Conclusion & Further Work

4.1 Conclusion

This paper presented a rewriting logic framework for action programming in non-deterministic, concurrent domains. Relational representations of state-action space and action programs have been defined in terms of syntax and axiomatization for state-action and program terms and operators that allow for the exploitation of associative-commutative rewriting to express non-determinism and concurrency. Deduction of plans that lead to resulting state-action spaces is achieved by repeatedly tensing and relaxing state-action terms according to given action programs and domain dynamics in the form of rewrite laws that can capture the effects of non-deterministic or disjunctive actions. An approach to deduce probabilities and rewards that account for resulting state-action terms has been discussed, and an illustrating example has been presented.

The concepts discussed in this paper have been implemented in the MAUDE language. Thus, several of MAUDE's beneficial properties were be exposed to the resulting action programming language and its interpreter, e.g. explicit sorting, sub-sorting and operator polymorphism allowing for structured domain specification and order-sorted unification, its object-layer and meta-programming facilities as well as support for domain specific syntactic representation of fluents and actions due to MAUDE's user-definable syntax. The current implementation and sample programs can be downloaded from http://www.pst.ifi.lmu.de/%7Ebelzner/action-programming/.

4.2 Further Work

There remain various directions to explore in future research. The implementation of the language in MAUDE provides direct access to a number of tools, most prominently to MAUDE's implementation of a model checker, that also allows to perform model checking w.r.t. to formulas specified in linear temporal logic. It seems to be valuable to explore the applicability of these tools to programs written in the proposed action programming language.

Another task will be to integrate the presented approach more closely with existing action formalisms like the situation calculus or the fluent calculus, as these provide well-understood solutions to the frame, ramification and qualification problems (Reiter 2001; Thielscher 1998). To this end, the presented approach could be generalized towards agent logic programs (Drescher et al. 2009), decoupling the specification of behaviour in terms of action programs from the underlying action formalism used to reason about changes and effects arising from program execution.

As shortly mentioned in section 3.2.3, relational markov decision processes can be represented in rewriting logic straightforwardly through the presented approach for modelling system dynamics. For this framework, manifold techniques for dynamic programming, such as value and policy iteration, as well as reinforcement learning algorithms have been defined (e.g. Tadepalli et al. 2004). The application of these techniques in rewriting logic action programming seems to be a promising direction for future work.

Finally, as true concurrency is supported by the proposed action programming framework in rewriting logic, it seems natural to investigate planning and action programming for multi-agent domains more extensively. In particular, it would be interesting to integrate results from game theory into the framework to allow for specification and optimization of multi-agent coordination.

References

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND QUESADA, J. F. 2002. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci. 285,* 2, 187–243.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. L., Eds. 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, vol. 4350. Springer.

DRESCHER, C., SCHIFFEL, S., AND THIELSCHER, M. 2009. A declarative agent programming language based on action theories. In *FroCoS*, S. Ghilardi and R. Sebastiani, Eds. Lecture Notes in Computer Science, vol. 5749. Springer, 230–245.

EKER, S. 1996. Fast matching in combinations of regular equational theories. *Electr. Notes Theor. Comput. Sci. 4*, 90–109.

EKER, S. 2003. Associative-commutative rewriting on large terms. In *RTA*, R. Nieuwenhuis, Ed. Lecture Notes in Computer Science, vol. 2706. Springer, 14–29.

GIACOMO, G., LESPRANCE, Y., LEVESQUE, H., AND SARDINA, S. 2009. Indigolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming:*, A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, Eds. Springer US, 31–72.

GU, Y. AND SOUTCHANSKI, M. 2009. Order-sorted reasoning in the situation calculus. In *Proceedings of Commonsense 2009*, G. Lakemeyer, L. Morgenstern, and M.-A. Williams, Eds. 65–72.

HENDRIX, J. AND MESEGUER, J. 2012. Order-sorted equational unification revisited. *Electr. Notes Theor. Comput. Sci. 290*, 37–50.

MARTÍ-OLIET, N. AND MESEGUER, J. 1999. Action and change in rewriting logic. In *Dynamic Worlds: From the Frame Problem to Knowledge Management*, R. Pareschi and B. Fronhöfer, Eds. Applied Logic Series, vol. 12. Kluwer, 1–53.

MCCARTHY, J. AND HAYES, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*. Vol. 4. 463–502.

MESEGUER, J. 1992. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci. 96*, 1 (Apr.), 73–155.

MESEGUER, J., GOGUEN, J. A., AND SMOLKA, G. 1989. Order-sorted unification. *J. Symb. Comput. 8*, 4, 383–413.

REITER, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, illustrated ed. The MIT Press, Massachusetts, MA.

TADEPALLI, P., GIVAN, R., AND DRIESSENS, K. 2004. Relational reinforcement learning: An overview. In *Proceedings of the ICML'04 Workshop on Relational Reinforcement Learning*.

THIELSCHER, M. 1998. Introduction to the fluent calculus. *Electron. Trans. Artif. Intell. 2*, 179–192.

THIELSCHER, M. 2005. Flux: A logic programming method for reasoning agents. *TPLP 5*, 4-5, 533–565.

THIELSCHER, M. 2008. *Action Programming Languages*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

VAN RIEMSDIJK, M. B., DE BOER, F. S., DASTANI, M., AND MEYER, J.-J. C. 2006. Prototyping 3apl in the maude term rewriting language. In *CLIMA*, K. Inoue, K. Satoh, and F. Toni, Eds. Lecture Notes in Computer Science, vol. 4371. Springer, 95–114.