Verifiable Decisions in Autonomous Concurrent Systems*

Lenz Belzner

LMU Munich, PST Chair, Germany
belzner@pst.ifi.lmu.de

Abstract. Being able to take decisions at runtime is a crucial ability for any system that is designed to act autonomously in uncertain or even unknown environments. This autonomy necessitates to formally check system properties at design time to ensure avoidance of problems or even harm caused by the system at runtime. This paper is about the formal specification of concurrent systems that are capable of reasoning about the consequences of their actions, enabling them to coordinate and decide on what to do autonomously. A non-deterministic procedural action programming language is defined to constrain system behaviour at design time. Rewriting logic is employed to construct and evaluate possible traces of programs in a decision-theoretic manner, allowing agents to perform goal-based actions autonomously at runtime as well as providing possibilities to model-check system properties at design time.

1 Introduction

Concurrent multiagent systems designed to act autonomously in highly dynamic, non-deterministic environments expose a number of requirements that have to be addressed by modelling and specification approaches. In fact, their design has to enable a system to reason about possibly uncertain outcomes of its actions in order to decide what behaviour to perform at runtime. This kind of cognitive ability requires specification of knowledge about actions' preconditions and effects. Algorithms have to be provided to enable a system to evaluate and decide on particular behavioural alternatives. Also, in the presence of multiple agents that share the same environment, the issues of concurrency and synchronization have to be accounted for by any specification approach.

Along with the specification of knowledge about actions and uncertainty, it should be possible to constrain the behaviour of an autonomous system at design time. To this end, non-deterministic action programming languages [1] allow for specification of behavioural constraints. In fact, non-deterministic programs allow to specify plan sketches that can be seen as behavioural alternatives provided to a system. These alternatives can then be evaluated in the context of a situation that a system finds itself in at runtime.

For systems that act autonomously, formal verification of properties at design time, e.g. by model checking, is a valuable source of information for design

* This work has been partially funded by the EU project ASCENS, 257414.

E. Kühn and R. Pugliese (Eds.): COORDINATION 2014, LNCS 8459, pp. 17–32, 2014.

choices and error detection. In this paper, the formalization of knowledge and behavioural constraints for ensembles is realized in rewriting logic [2,3,4]. Domain knowledge is represented as a relational Markov Decision Process [5,6]; a non-deterministic action programming language is defined to allow for specification of behavioural constraints. The rewriting logic concepts of matching and rewriting are employed to provide to a system the ability to reason about the consequences of its behaviour. A number of problems that are typical for cognitive systems are addressed in this paper:

- *Specification of knowledge about actions.* Besides providing to a system information about pre- and postconditions of actions, it is necessary that the system knows what properties of the environment remain unchanged. In static logics like first-order logic, this is a non-trivial issue known as the frame problem [7].
- *Legality & projection.* If a system is provided with the ability to choose from various behavioural alternatives, this may lead to action choices that cannot be executed at runtime due to precondition violations. To decide whether a certain behavioural trace is legal (i.e. executable w.r.t. action preconditions), a cognitive agent has to be capable of deciding whether certain properties of the environment hold as consequences of its actions; an ability that is known as projection [8].
- *Trace evaluation & system property verification.* Non-deterministic action programs are employed to constrain system behaviour, giving raise to various execution traces, thus enabling a system to evaluate behavioural alternatives w.r.t. its goals. Also, the consequences arising from the execution of traces can be used to verify properties of the given action program at design time.
- *Concurrency.* When multiple agents interact dynamically at runtime, the issues of concurrency, synchronization and coordination arise, especially in the presence of sharing. Reasoning about concurrent execution should be possible with specified knowledge about a domain; also, the legality and projection problems have to be dealt with in the presence of concurrency.

This paper extends previous work on action programming in rewriting logic [9] by decision-theoretic concepts, and discusses in more detail the integration of concurrency and coordination. Also, an approach towards probabilistic model-checking for action programs is discussed. The paper is outlined as follows. Section 2 introduces rewriting logic and action programming. Section 3 discusses the specification of knowledge and behaviour for autonomous concurrent systems in terms of a rewrite theory, and defines how the concepts of matching and rewriting are employed to interpret and reason about the specification. Section 4 discusses how to perform model checking of action programs for typical desirable system properties. A short example illustrating the approach is given in section 5. In section 6, related work is compared with the presented specification approach. Finally, section 7 summarizes the results and gives a brief outlook on further research venues.

2 Preliminaries

This section gives a brief introduction to rewriting logic (section 2.1) and action programming (section 2.2).

2.1 Rewriting Logic

The core concept of rewriting logic are rewrite theories $(\Sigma, A \cup E, R, \phi)$ [2,3,4]. Σ is a signature defining sorts, sort-orders and operations to build terms from sorts in the signature. A are axiomatic definitions for operations in Σ like associativity, commutativity, idempotency or identity, and E specifies a set of equations for terms in Σ. $A \cup E$ defines equivalence classes for terms, representing the static structure of the rewrite theory. R is a set of possibly conditional rewrite rules that specify the theory's dynamics. Finally, ϕ is a function from operations to natural numbers that defines which arguments of an operation are frozen, meaning that no rewriting will take place on this argument.

Term rewriting with rewrite theories is performed by matching terms with rules. In rewriting logic, matching is performed *modulo axioms with extensions* (called *Ax-matching*). For example, consider an operation $\circ$ that is associative and commutative. Then, the term $a \circ b \circ c$ would Ax-match the term $c \circ a$, because $a \circ b \circ c =_{\mathrm{A}} c \circ a \circ b$ due to axioms for $\circ$, and obviously $c \circ a$ is a subterm of the given term. If terms contain variables, matching applies if there is a substitution that renders a subterm of a given subject term equal to a term to be matched; thus, a more concrete term matches a more general one.

Rewrite rules are of the form $l \to r$ **if** *Conditions*, where l and r are terms in Σ that have an equal least sort, and *Conditions* can either be boolean expressions, matching conditions (true if a term t Ax-matches a term t', denoted $t := t'$) and rewriting conditions (true if a term t rewrites to a term t' according to rules in R, denoted $t \to t'$). Rewriting of a subject term t is performed if a non-variable subterm of t matches a rule's left-hand side, and the conditions of the rule hold for the particular matching substitution. The matched subterm is then replaced by the rules right-hand side. Rewriting of a term to its normal form, where no more rewrite rules can be applied, is denoted by $t \to_! t'$.

As Ax-matching a term with rules may result in different matched subterms and substitutions, rewriting can naturally express non-determinism. Also, as multiple rules may be applied to different subterms in parallel, rewriting logic is considered an intuitive way to deal with concurrency [4].

As long as some admissibility requirements are satisfied (e.g. if there are no fresh variables in a rule's right-hand side), the process of rewriting can be performed computationally. This fact has led to the development of the term-rewriting language Maude [2,3]. Rewrite theories can straightforwardly be implemented as Maude modules. The language provides a built-in breadth-first search command to compute all possible rewrites of a term, where search depth and the number of desired solutions are parametrized. This approach can for example be used to perform automated state-space search on rewrite theories in order to model-check a given specification (see section 4).

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" [1].

Action programming allows to specify non-deterministic programs that can be considered as a plan sketch, roughly outlining desired behaviour of a system without committing to a certain trace of actions. In other words, an action program defines a constrained space of behavioural alternatives. To allow a system to reason about these alternatives at runtime, a domain specification is provided that contains definitions of *fluents* (i.e. state properties) that are subject to change (e.g. due to action execution) as well as *actions* in terms of their preconditions and effects, which may also be non-deterministic. This knowledge can then be employed by the system to decide what to do at runtime.

For example, consider fluents x, y and z, and actions α and β. Informally, assume $x, y \rightarrow_\alpha z$, $x \rightarrow_\beta y$ and $y \rightarrow_\beta z$, denoting an action's precondition and (non-deterministic) effects. Then, given the action program α # β where # is a non-deterministic choice operator and a current situation where x and y hold, an agent can derive that executing α will lead to a situation where z holds, and that execution of β could result in a state where either y holds (as $x, y \rightarrow_\beta y, y$) or where x and z hold (according to the latter specification of β).

Influential action programming formalisms are the situation calculus with its language Golog [8] and the fluent calculus and its language Flux [10]. The situation calculus employs a fluent-wise regressive specification of action effects known as *successor state axioms*, where the frame problem is explicitly handled by also stating under what conditions some fluent does not change. States are not explicitly managed, but rather the notion of a *situation* is used to test whether a fluent holds after execution of an action program. On the other hand, the fluent calculus uses a progressive, action-wise specification of domain dynamics by defining modifications of an associative-commutative representation of a state that arise due to action execution.

3 Decisions in Autonomous Concurrent Systems

As mentioned in section 2.2, to provide to a system the ability to reason in terms of action programming, specification of domain dynamics as well as a language to define behavioural alternatives are necessary. In section 3.1, the encoding of domain knowledge in terms of a rewrite theory encoding a relational MDP is discussed. Section 3.2 defines a non-deterministic, procedural action programming language that employs the specified rewrite theory to construct trace consequences through Ax-matching and rewriting (see section 2.1), and introduces the notion of a trace's value, thus enabling systems to reason about action preferences. Section 3.3 deals with the explicit treatment of projection, and provides definitions for branching and loops. Section 3.4 discusses action programming with concurrency in the presence of sharing, allowing for agent synchronization and coordination.

3.1 RMDPs as Rewrite Theories

For specification of non-deterministic domain dynamics, the proposed approach employs the notion of a relational MDP. A relational MDP is a tuple (S, A, T, R), where S is a set of relational states, A a set of actions, $T : S \times A \times S \to [0; 1]$ is a transition function specifying the probability that execution of a particular action in a state results in another one, and $R : S \to \mathbb{R}$ is a reward function. To encode such a relational MDP as a rewrite theory, a set of fluents F is defined (see section 2.2), from which the set of states Σ is built by means of the following syntax. Both $\wedge$ and $\vee$ are associative and commutative with identity element ϵ_σ. The equivalence classes of terms in Σ can be considered the set of states S of a relational MDP.

$$\Sigma := \epsilon_\sigma \mid f \in F \mid \neg\Sigma \mid \Sigma \wedge \Sigma \mid \Sigma \vee \Sigma$$

States annotated with probability are denoted by $\sigma_p := (\sigma \wedge \mathit{prob}(p)) \in \Sigma$, with a fluent $\mathit{prob} : [0; 1] \to F$ encoding the probability of a system being in state σ. The set of states annotated with a probability fluent is denoted by Σ_P.

The transition function T is encoded in terms of rewrite rules. If there are n non-deterministic transitions in T that define the effects of executing action α in state σ each leading to a state σ^i with probability $p^i, i \in \{1, ..., n\}$, a rule is defined for each of them in the form $\Sigma_P \times A \to \Sigma_P$ as follows (with $\sum_{i=1}^{n} p_i = 1$).

$$\sigma_{p^i} \times \alpha \to \sigma^i_{p*T(\sigma,\alpha,\sigma^i)} \ . \tag{1}$$

The states σ and the σ^i can be considered as pre- and postconditions of action α. Note that this notation of transitions provides a solution to the frame problem [7], as subject terms are matched with extension when applying rules. Thus, there is no necessity to explicitly specify state properties that remain unchanged by application of a rule.

Example 1. (Action rules) A rule for defining an action *move* that moves a robot R at position P (denoted by a fluent $\mathit{pos} : \mathit{Robot} \times \mathit{Position} \to F$) to a desired position P' with a probability of 0.9 and has no effect otherwise is specified by the following rules.

$$\mathrm{pos}(R, P)_p \times \mathrm{move}(R, P') \to \mathrm{pos}(R, P')_{p*0.9} \ .$$
$$\mathrm{pos}(R, P)_p \times \mathrm{move}(R, P') \to \mathrm{pos}(R, P)_{p*0.1} \ .$$

Finally, a MDP's reward function R can simply be encoded in terms of an operation together with corresponding equations. For example, considering a variable S of sort state[1], the equations $R(\mathrm{pos}(r, p) \wedge S) = 1, R(S) = 0\ [\mathit{otherwise}]$ specify a reward of 1 for states where robot r is at position p and zero reward for all other states.

[1] Throughout the paper, variables are denoted by uppercase letters.

3.2 Progressive Action Programming

This section describes how to constrain system behaviour in terms of an action program, and how a system can construct and evaluate behavioural alternatives according to these constraints when given a rewrite theory as domain specification as outlined in section 3.1. To this end, a procedural non-deterministic action programming language is defined with syntax Π.

$$\Pi := \epsilon_\pi \mid \alpha \in A \mid \Pi\,;\Pi \mid \Pi\#\Pi$$

Here, ϵ_π denotes the empty program, α is an action from the domain specification, ; is an associative sequential operator, and # is an associative-commutative non-deterministic choice operator, offering behavioural alternatives to a system executing the program. Both operators have ϵ_π as identity element.

To allow for numerical evaluation of states, these are annotated with a value (i.e. gathered reward), denoted by $\sigma_{p\times v} := (\sigma_p \wedge \mathit{value}(v)) \in \Sigma$ with a fluent $\mathit{value} : \mathbb{R} \to F$ encoding the value v gathered in a state σ. The corresponding set of states annotated with probability and value is denoted by $\Sigma_{P\times V}$.

Program interpretation and trace evaluation is then performed by rewriting *configurations* $\Sigma_{P\times V} \times \Pi \times \Pi \times \mathbb{N}$ that consist of a current state, an action program to be evaluated, an action trace leading to the current state and a planning horizon. The set of configurations is denoted as Ξ. For ease of notation, functions $\sigma_\xi, p_\xi, v_\xi, \pi_\xi, \tau_\xi, h_\xi$ are defined, taking as parameter a configuration $\xi := \sigma_{p\times v} \times \pi \times \tau \times h$, returning its respective values, e.g. $\sigma_\xi(\xi) = \sigma$.

To compute possible consequences of executing for h steps an action program π in a particular state σ, an initial configuration $\xi_{\text{init}} := \sigma_{1.0\times 0.0} \times \pi \times \epsilon_\pi \times h$ is rewritten according to the following rules, resulting in a set of configurations $\{\xi \mid \xi_{\text{init}} \to_! \xi \in \Xi\}$.[2]

$$\sigma_{p\times v} \vee \left(\bigvee_i \sigma^i_{p_i\times v_i}\right) \to \sigma_{p\times v}\ . \tag{2}$$

$$\sigma_{p\times v} \times \alpha \times \tau \times h \to \sigma'_{p'\times v'} \times \epsilon_\pi \times (\tau\,;\alpha) \times (h-1)$$
$$\quad \textbf{if } h > 0 \wedge \sigma_p \times \alpha \to \sigma'_{p'} \wedge v' := v + R(\sigma')\ . \tag{3}$$

$$\sigma_{p\times v} \times (\pi_1\,;\pi_2) \times \tau \times h \to \sigma'_{p'\times v'} \times \pi_2 \times \tau' \times h'$$
$$\quad \textbf{if } \sigma_{p\times v} \times \pi_1 \times \tau \times h \to \sigma'_{p'\times v'} \times \epsilon_\pi \times \tau' \times h'\ . \tag{4}$$

$$\sigma_{p\times v} \times (\pi_1 \# \pi_2) \times \tau \times h \to \sigma_{p\times v} \times \pi_1 \times \tau \times h\ . \tag{5}$$

As state disjunction is associative and commutative, rule (2) rewrites a state in disjunctive normal form to all its conjunctive subterms, leading to subsequent rewriting of configurations only containing conjunctive states. Note that disjunctive normal form for states can easily be assured by specifying according equations for state-terms.

[2] Maude's built-in search operator performs a breadth-first search; a depth-first strategy can be defined using Maude's internal strategies [3] or a dedicated strategy language [11], e.g. by assigning rule application preferences.

Rule (3) rewrites a configuration containing a single action α as program if the planning horizon has not yet been reached and the action's preconditions are satisfied by the current state. Satisfaction of preconditions is tested by rewriting the current state σ_p with the given action according to the transition rules for actions in the domain. If $\sigma_p \times \alpha$ can be rewritten, the preconditions are satisfied, and any rewards of the reached state are added to v[3]. Otherwise rewriting terminates, thus solving the legality problem by Ax-matching left-hand sides of rewrite rules encoding the transition function of a relational MDP with $\sigma_p \times \alpha$.[4] Note that Ax-matching is highly efficient; for *ac*-terms typically met in practice, its complexity does not exceed $O(\log n)$, where n is term size [12]. Ax-matching also results in a non-deterministic argument pick if a parameter of the action that is to be processed is a variable.

Rewriting of sequential programs determines all configurations that can result by executing the sequence's head, which are then further rewritten according to the sequence's tail (rule (4)). Choice programs are rewritten according to rule (5), that will rewrite a configuration according to all possible program choices as the choice operator is associative and commutative.

In contrast to configurations for which rewriting terminated due to precondition violation, rewriting also terminates for a configuration ξ exposing $h_\xi(\xi) = 0$ (due to rule (3)) or $\pi_\xi(\xi) = \epsilon_\pi$ (because there is no rule for interpreting ϵ_π). Thus, these properties can be used to define the set of *legally terminating configurations* $\Xi_\downarrow$ arising from rewriting an initial configuration $\xi_{\text{init}} \in \Xi$ according to program traces where no action precondition is violated.

$$\Xi_\downarrow(\xi_{\text{init}}) := \{\xi \,|\, \xi_{\text{init}} \to^! \xi \wedge (\pi_\xi(\xi) = \epsilon_\pi \vee h_\xi(\xi) = 0)\}$$

The set of *legally terminating program traces* $T_\downarrow$ for an initial configuration $\xi_{\text{init}} \in \Xi$ is defined as follows.

$$T_\downarrow(\xi_{\text{init}}) := \{\tau \,|\, \xi \in \Xi_\downarrow(\xi_{\text{init}}) \wedge \tau_\xi(\xi) = \tau)\}$$

Similarly, the set of *configurations yielding a particular, legally terminating trace* $\Xi_{\tau_\downarrow}$ is defined.

$$\Xi_{\tau_\downarrow}(\xi_{\text{init}}) := \{\xi \,|\, \xi \in \Xi_\downarrow(\xi_{\text{init}}) \wedge \tau_\xi(\xi) = \tau\}$$

The *expected value of a legally terminating trace* V_e is the sum of expected values (which is the product of probability and value) of its non-deterministic outcomes.

$$V_e\,(\tau, \xi_{\text{init}}) := \sum_{\xi \in \Xi_{\tau_\downarrow}(\xi_{\text{init}})} (p_\xi(\xi) * v_\xi(\xi)) \tag{6}$$

[3] When rewards are specified action-wise, i.e. $R : \Sigma \times A \times \Sigma$, the computation of v in the condition of rule (3) changes to $v := R(\sigma, \alpha, \sigma')$.

[4] To allow for Ax-matching of states and actions technically, conjunction of fluents and concurrency of actions (see section 3.4) are implemented in terms of a single associative-commutative operation, e.g. $\circ$: STATE $\to$ STATE, with sorts FLUENT and ACTION being subsorts of sort STATE. I.e. $\sigma \wedge \sigma' = \sigma \circ \sigma', \sigma \times \alpha = \sigma \circ \alpha, \alpha \parallel \alpha' = \alpha \circ \alpha'$.

The *legal termination probability* $P_{\downarrow}$ of a particular trace is the sum of all probabilities of configurations that yield the trace.

$$P_{\downarrow}(\tau, \xi_{\text{init}}) := \sum_{\xi \in \Xi_{\tau_{\downarrow}}(\xi_{\text{init}})} p_{\xi}(\xi) \tag{7}$$

Summarizing, given a domain specification and an initial configuration (i.e. a current state[5], an action program and a planning horizon), rules (2) to (5) together with the definitions for V_e and $P_{\downarrow}$ enable a system to perform the following tasks: (*i*) Computation of all program traces and their consequences. (*ii*) Determination of expected values for traces. (*iii*) Determination of legal termination probability of a particular trace.

Example 2. (Expected trace values and legal termination probability) Consider the rule for action move from example 1. Let S be a variable of sort Σ and $\text{r}(\text{pos}(R, sa) \wedge S) = 1.0$ (0 otherwise). Consider $\xi_{\text{init}} = \text{pos}(r, pos)_{1.0 \times 0.0} \times \text{move}(r, sa) \times \epsilon_{\pi} \times 1$. Then, expected value and legal termination probability of a possible trace of the program $\text{move}(r, sa)$ that tries to move a robot r to the safety area sa compute as follows when executing it in a state where r is at position pos (with $pos \neq sa$).

$$\begin{aligned}
&\xi_{\text{init}} \rightarrow \text{pos}(r, sa)_{0.9 \times 1.0} \times \epsilon_{\pi} \times \text{move}(r, sa) \times 0 \ . \\
&\xi_{\text{init}} \rightarrow \text{pos}(r, pos)_{0.1 \times 0.0} \times \epsilon_{\pi} \times \text{move}(r, sa) \times 0 \ . \\
&V_e(\text{move}(r, sa), \xi_{\text{init}}) = 0.9 * 1.0 + 0.1 * 0.0 = 0.9 \ . \\
&P_{\downarrow}(\text{move}(r, sa), \xi_{\text{init}}) = 0.9 + 0.1 = 1.0 \ .
\end{aligned}$$

3.3 Projection, Branching and Loops

The projection problem is to decide whether a certain property holds in a state or not (e.g. after a number of actions have been executed). As discussed in section 3.2, when interpreting action execution a state term is updated to yield the information about the resulting state. Thus, the projection problem reduces to deciding whether a property holds in a given state term. This is easily solved in rewriting logic: It suffices to check whether a given condition term in Σ Ax-matches a particular state term for a substitution of variables θ. To this end, *state subsumption* (denoted by $\sqsupseteq$) is defined as follows.

$$\begin{aligned}
&\sigma_? \sqsupseteq \sigma \rightarrow \theta \ \textbf{if} \ \sigma_?/\theta := \sigma \ . \\
&\sigma_? \sqsupseteq \sigma \Leftrightarrow \exists \theta : \sigma_? \sqsupseteq \sigma \rightarrow \theta \ . \\
&\sigma_? \not\sqsupseteq \sigma \Leftrightarrow \nexists \theta : \sigma_? \sqsupseteq \sigma \rightarrow \theta \ .
\end{aligned}$$

A state $\sigma_? \in \Sigma$ thus subsumes another state $\sigma \in \Sigma$ if its corresponding state term is more general than the state term of σ, which is checked by Ax-matching the two terms. With this definition of state subsumption, operations

[5] Note that partial observable domains can be accounted for by rewriting an initial configuration $\bigvee_i (\sigma^i_{p^i \times v^i}) \times \pi \times \epsilon_{\pi} \times h$ with $\sum_i p^i = 1$, where the σ^i represent different possible states an agent considers, each with probability p^i.

are introduced for action programs that check whether a condition holds (or does not hold, respectively) at a particular stage of execution. The syntax for action programs is extended as follows.

$$\begin{aligned} \Pi_? := \Pi \mid & ?(\Sigma)\{\Pi_?\} \mid \neg?(\Sigma)\{\Pi_?\} \\ & \mid \textbf{if } \Sigma \textbf{ then } \Pi_? \textbf{ else } \Pi_? \textbf{ end}_{\text{if}} \\ & \mid \textbf{while } \Sigma \textbf{ do } \Pi_? \textbf{ end}_{\text{while}} \end{aligned}$$

Interpretation of $?(\sigma_?)\{\pi\}$ succeeds if there is a substitution θ for which a condition $\sigma_?$ subsumes the state of the current configuration to be rewritten. If so, the variable substitutions for which the condition hold are applied to the argument action program π (rule (8)). Similarly, rewriting $\neg?(\sigma_?)\{\pi\}$ succeeds if there is no such substitution (rule (9)).

$$\sigma_{p\times v}\times?(\sigma_?)\{\pi\} \times \tau \times h \to \sigma_{p\times v} \times \pi/\theta \times \tau \times h \textbf{ if } \sigma_? \sqsupseteq \sigma \to \theta \ . \tag{8}$$

$$\sigma_{p\times v} \times \neg?(\sigma_?)\{\pi\} \times \tau \times h \to \sigma_{p\times v} \times \pi \times \tau \times h \textbf{ if } \sigma_? \not\sqsupseteq \sigma \ . \tag{9}$$

With these operations, conditional branching can be defined as a macro.

$$\textbf{if } \sigma_? \textbf{ then } \pi_1 \textbf{ else } \pi_2 \textbf{ end}_{\text{if}} = ?(\sigma_?)\{\pi_1\}\#\neg?(\sigma_?)\{\pi_2\} \ . \tag{10}$$

Testing whether a condition $\sigma_?$ holds in (i.e. subsumes) a particular state σ also allows for interpretation of loops. If $\sigma_?$ subsumes σ for a substitution θ, the loop body will be executed with θ applied, followed by the loop itself; otherwise, the loop reduces to the empty program.

$$\begin{aligned} &\sigma_{p\times v} \times \textbf{while } \sigma_? \textbf{ do } \pi \textbf{ end}_{\text{while}} \times \tau \times h \to \\ &\quad \sigma_{p\times v} \times \pi/\theta \textbf{;while } \sigma_? \textbf{ do } \pi \textbf{ end}_{\text{while}} \times \tau \times h \textbf{ if } \sigma_? \sqsupseteq \sigma \to \theta \ . \end{aligned} \tag{11}$$

$$\begin{aligned} &\sigma_{p\times v} \times \textbf{while } \sigma_? \textbf{ do } \pi \textbf{ end}_{\text{while}} \times \tau \times h \to \\ &\quad \sigma_{p\times v} \times \pi_\epsilon \times \tau \times h \textbf{ if } \sigma_? \not\sqsupseteq \sigma \ . \end{aligned} \tag{12}$$

3.4 Concurrency

To allow for specification of action programs for agents that act concurrently, the set of actions is extended to $A_{\shuffle} := \epsilon_\alpha \mid A \mid A_{\shuffle} \parallel A_{\shuffle}$ where $\parallel$ is associative-commutative with identity ϵ_α and denotes parallel execution of actions. Programs are allowed to be executed in parallel as well: $\Pi_{\shuffle} := \Pi_?(A/A_{\shuffle}) \mid \Pi_{\shuffle} \parallel \Pi_{\shuffle}$, where the syntax of programs in $\Pi_?$ is extended to allow for parallel actions.

As action rules are applied sequentially to the current state term when rewriting, it is necessary to *lock* changing subterms of the state term to model true concurrency, disallowing interleaving application of actions that would result in race conditions. To this end, action dynamics as in rule (1) (see section 3.1) are *automatically* compiled to rules of form (13), explicitly denoting which fluents of the precondition σ are changed by action execution and which ones are left unchanged. Changed fluents are locked by an operation $\phi : \Sigma \to \Sigma, \phi(\sigma) \wedge \phi(\sigma') =$

$\phi(\sigma \wedge \sigma')$, keeping them from being further matched with parallel actions' preconditions.

$$\sigma_p \times \alpha_{\shuffle} \rightarrow \sigma^i_{p*T(\sigma,\alpha_{\shuffle},\sigma^i)} \text{ where } \sigma^i = \phi(\sigma^i_{\text{changed}}) \wedge \sigma^i_{\text{unchanged}} \ . \tag{13}$$

Example 3. (Action Rules for Concurrent Domains) Consider the specification of actions *grab* and *drop* in a domain with concurrency. Note how state properties changed by the actions are locked.

$$(\text{pos}(R,P) \wedge \text{pos}(O,P))_p \times \text{grab}(R,O) \rightarrow (\text{pos}(R,P) \wedge \phi(\text{on}(R,O)))_{p*0.9} \ .$$
$$(\text{pos}(R,P) \wedge \text{pos}(O,P))_p \times \text{grab}(R,O) \rightarrow (\text{pos}(R,P) \wedge \text{pos}(O,P))_{p*0.1} \ .$$
$$(\text{pos}(R,P) \wedge \text{on}(R,O))_p \times \text{drop}(R,O) \rightarrow (\text{pos}(R,P) \wedge \phi(\text{pos}(O,P)))_{p*1.0} \ .$$

For instance, $\text{grab}(r1,o) \parallel \text{drop}(r2,o)$ should always be considered illegal, as at least one precondition is violated in any case before applying any of the actions; if interleaving rewriting was performed without locking state changes, the application of *drop* would in turn render *grab* executable (and vice versa), which is something that should not occur when the actions are considered truly concurrent. I.e., locking of changed properties resembles a precondition check before *any* action is applied and a check of postconditions after application of *all* actions. Note that, if *interleaving concurrency* is to be modelled, locking of changing state properties can simply be dropped. Interleaving is then achieved by the non-deterministic application of rewrite rules for action dynamics from the domain specification.

Concurrent action programs are normalized to sequential and choice terms as shown in equations (14) and (15) (for $\alpha_i \in A_{\shuffle}, \pi_i \in \Pi_{\shuffle}$). Due to this normalization, interpretation of sequences and choices (rules (4) and (5), see section 3.2) can be performed without rule modification by rewriting configurations with concurrent action programs.

$$(\pi_1 \# \pi_2) \parallel \pi_3 = (\pi_1 \parallel \pi_3) \# (\pi_2 \parallel \pi_3) \ . \tag{14}$$
$$(\alpha_1 ; \pi_1) \parallel (\alpha_2 ; \pi_2) = (\alpha_1 \parallel \alpha_2) ; (\pi_1 \parallel \pi_2) \ . \tag{15}$$

Rewriting (i.e. interpretation) of a parallel action (rule (3) for domains without concurrency, see section 3.2) resulting from transformation of a concurrent action program according to equation (15) is changed to account for resolution of locked state properties arising from rewriting according to rules of form (13) that specify action dynamics for concurrent domains. [6]

$$\begin{aligned} &\sigma_{p \times v} \times \alpha_{\shuffle} \times \tau \times h \rightarrow \\ &\quad (\sigma_1 \wedge \sigma_2)_{p' \times v'} \times \epsilon_\pi \times (\tau ; \alpha_{\shuffle}) \times (h-1) \\ &\quad \textbf{if } h > 0 \wedge \sigma_p \times \alpha_{\shuffle} \rightarrow (\phi(\sigma_1) \wedge \sigma_2)_{p'} \wedge v' := v + R(\sigma_1 \wedge \sigma_2) \ . \end{aligned} \tag{16}$$

[6] For action-wise rewards, it is $v' := v + R(\sigma, \alpha_{\shuffle}, \sigma_1 \wedge \sigma_2)$. Then, for parallel actions reward is additive: $R(\sigma, \alpha \parallel \alpha', \sigma') = R(\sigma, \alpha, \sigma') + R(\sigma, \alpha', \sigma')$.

Example 4. (Concurrency) Consider the *move* action from example 1. The example shows rewriting a state according to a concurrent action that tries to move two robots to a position p_3, illustrating the evaluation of the rewrite condition in rule (16). Rewrite results in the following set of states matching the conditions right-hand side (i.e. all actions have been rewritten). Note that depending on the outcome of actions, different state properties are locked.

$$\begin{aligned}&(\mathrm{pos}(r_1,p_1)\wedge\mathrm{pos}(r_2,p_2))_p\times\mathrm{move}(r_1,p_3)\parallel\mathrm{move}(r_2,p_3)\rightarrow\\&\phi(\mathrm{pos}(r_1,p_3)\wedge\mathrm{pos}(r_2,p_3))_{p*0.9*0.9}\vee(\phi(\mathrm{pos}(r_1,p_3))\wedge\mathrm{pos}(r_2,p_2))_{p*0.9*0.1}\\&\vee(\mathrm{pos}(r_1,p_1)\wedge\phi(\mathrm{pos}(r_2,p_3)))_{p*0.1*0.9}\vee(\mathrm{pos}(r_1,p_1)\wedge\mathrm{pos}(r_2,p_2))_{p*0.1*0.1}\ .\end{aligned}$$

If particular actions require *synchronization* of agents, this can be specified by defining a rewrite rule according to rule (13) with an action that is explicitly parallel. A rule specified this way will only rewrite to a legal configuration if concurrent processes manage to synchronize their actions accordingly.

Example 5. (Synchronization) Consider an action t (for *transport*) that moves a victim to a target position when two agents perform it synchronously. A single agent trying to transport a victim has no effect and should be interpreted as a precondition violation. Thus, this action requires synchronization of agents' actions. This fact is specified as follows (considering a deterministic outcome for the sake of simplicity).

$$\begin{aligned}(\mathrm{pos}(R,P)\wedge\mathrm{pos}(R',P)\wedge\mathrm{pos}(V,P))_p\times\mathrm{t}(R,V,P')\parallel\mathrm{t}(R',V,P')\rightarrow\\\phi(\mathrm{pos}(R,P')\wedge\mathrm{pos}(R',P')\wedge\mathrm{pos}(V,P'))_{p*1.0}\ .\end{aligned}$$

If there is need for agents to synchronize, it may be valuable for agents to *coordinate* their actions by waiting for others to collaborate, eventually rendering a desired action executable. To this end, if an agent should execute an action α as soon as its precondition $\sigma_{\mathrm{pre}_\alpha}$ is satisfied, waiting for satisfaction of preconditions (e.g. collaborating agents) can be specified in terms of a macro **waitFor**. Note that $\sigma_{\mathrm{pre}_\alpha}$ can be determined from the lefthand side of the domain specification rule that specifies the dynamics of α.

$$\mathbf{waitFor}(\alpha)=\mathbf{while}\ \neg\sigma_{\mathrm{pre}_\alpha}\ \mathbf{do}\ \mathrm{noop}\ \mathbf{end}_{\mathrm{while}}\,;\alpha\ . \tag{17}$$

4 Probabilistic Model Checking of Action Programs

This section shows how to perform symbolic probabilistic model checking for action programs, whose interpretation results in configurations containing action traces determined through application of Ax-matching and rewriting as shown in section 3. To this end, this section discusses the relation between typical system properties to be model-checked and rewriting of configurations. System properties are expressed in terms of PCTL formulas. For an in-depth discussion

of model-checking in general, and model-checking with PCTL in particular, see e.g. [13]. For the relation of model-checking and classical planning see e.g. [14].

A typical property that can be checked for an action program is whether it contains a legal trace that will reach a particular goal state satisfying[7] a property σ_{goal} within a given number of steps h with at least a desired probability $p_{\min}$. This property can be expressed in terms of a PCTL formula, where $\mathbb{P}_{\geq p_{\min}}$ denotes the requirement that its argument is satisfied at least with probability $p_{\min}$, and the *eventually* operator $\Diamond^{\leq h}$ denotes that its argument state is eventually reached within the given horizon h.

$$\mathbb{P}_{\geq p_{\min}}(\Diamond^{\leq h}\sigma_{\text{goal}}) \tag{18}$$

To check whether a given configuration ξ_{init} satisfies this property, a positive reward is specified for states that satisfy σ_{goal}, i.e. $R(\sigma_{\text{goal}}) > 0$; all other states expose a reward of zero. Then, (legally terminating) traces eventually satisfying σ_{goal} expose a positive reward.

$$\Xi^{+}_{\tau_{\downarrow}}(\xi_{\text{init}}) = \{\xi \,\big|\, \xi \in \Xi_{\tau_{\downarrow}}(\xi_{\text{init}}) \wedge v_{\xi}(\xi) > 0\} \tag{19}$$

To determine the probability of a trace eventually satisfying σ_{goal}, the probabilities of terminated configurations with equal traces (arising due to nondeterminism) are summed up. If there is no such trace, the probability is zero.

$$P^{+}_{\downarrow}(\tau, \xi_{\text{init}}) = \sum_{\xi \in \Xi^{+}_{\tau_{\downarrow}}(\xi_{\text{init}})} p_{\xi}(\xi) \tag{20}$$

Property (18) is satisfied if there exists a legally terminating trace with an expected value greater than zero and a probability of eventually reaching a state satisfying σ_{goal} greater than $p_{\min}$.[8]

$$\mathbb{P}_{\geq p_{\min}}(\Diamond^{\leq h}\sigma_{\text{goal}}) \Leftrightarrow \exists \tau \in T_{\downarrow}(\xi_{\text{init}}) : V_e(\tau, \xi_{\text{init}}) > 0 \wedge P^{+}_{\downarrow}(\tau, \xi_{\text{init}}) \geq p_{\min} \tag{21}$$

In general, a PCTL formula of the form $\mathbb{P}_J(\Diamond^{\leq h}\sigma)$, where J is an interval in $[0; 1]$ can be checked for any initial configuration $\xi_{\text{init}} \in \Xi$ by specifying a reward greater than zero for states subsumed by σ, and zero reward for all other states; then, it is checked by rewriting whether probabilities of legal traces whose expected value is bigger than zero are in the interval J.

$$\mathbb{P}_J(\Diamond^{\leq h}\sigma_{\text{goal}}) \Leftrightarrow \exists \tau \in T_{\downarrow}(\xi_{\text{init}}) : V_e(\tau, \xi_{\text{init}}) > 0 \wedge P^{+}_{\downarrow}(\tau, \xi_{\text{init}}) \in J \tag{22}$$

In particular situations, a system may be required to maintain a particular state property σ_{maintain}. In these cases, it may be valuable to ensure a minimal probability $p_{\min}$ that the desired property will sustain with program execution. The PCTL formula $\mathbb{P}_{\geq p_{\min}}(\Box^{\leq h}\sigma_{\text{maintain}})$ expresses this system property, where the *always*

[7] A state σ satisfies a property $\sigma_?$ if $\sigma_? \sqsupseteq \sigma$ (see section 3.3).

[8] Depending on the property to be checked, existential quantification is to be replaced by universal quantification.

operator $\Box^{\leq h}$ denotes that its argument state is always satisfied within the given horizon h. This property can be reformulated in terms of an achieve goal that requires an upper bound on the maximal probability that the desired property will be violated by program execution: $\mathbb{P}_{<1-p_{\min}}(\Diamond^{\leq h}\neg\sigma_{\text{maintain}})$. This property can then be checked according to (21). Due to the duality of the always and eventually operators, transformation of maintain to achieve goals can be generalized for probability intervals [13].

$$\mathbb{P}_{]p,p'[}(\Box^{\leq h}\sigma) = \mathbb{P}_{[1-p',1-p[}(\Diamond^{\leq h}\neg\sigma) \tag{23}$$

5 Example

As a short informal example, consider a rescue scenario. There are two robots r_1 and r_2 at positions p_1 and p_2, respectively. There are also two victims v_1 and v_2, the former at position p_1, the latter at a position p_3. I.e. $\sigma^{\text{init}} := \text{pos}(r_1, p_1) \wedge \text{pos}(r_2, p_2) \wedge \text{pos}(v_1, p_1) \wedge \text{pos}(v_2, p_3)$ (see figure 1). The robots are supposed to transport victims to the safety area; to this end, a reward of one is given if there is any victim at the safety area sa, i.e. $R(\text{pos}(V, sa)) = 1$. Consider the following part of an action program that specifies a behavioural policy for the robots, where actions *move* and t are defined as in examples 1 and 5. It states that a robot R either moves to the position P of a victim V if there is any, or, if the robot is already at a position of a victim, it waits until it can transport the victim to the safety area in collaboration with another robot.[9]

$$\begin{aligned} \mathit{policy}(R) := \mathbf{while}\ \sigma_\epsilon\ \mathbf{do}& \\ &?(\text{pos}(V, P))\{\text{move}(R, P)\}\# \\ &?(\text{pos}(V, P) \wedge \text{pos}(R, P))\{\mathbf{waitFor}(\text{t}(R, V, sa))\} \\ \mathbf{end}_{\text{while}}& \end{aligned}$$

The robots can interpret the action program $\pi := \mathit{policy}(r_1) \parallel \mathit{policy}(r_2)$ for a σ^{init}, for example for a planning horizon of 2 (because, for example, their energy resources suffice only for this horizon). Interpretation of the program leads to various traces due to the choice operator and due to the non-deterministic effect of *move*. Rewriting of an initial configuration $\xi_{\text{init}} := \sigma^{\text{init}}_{1.0\times 0.0} \times \pi \times \pi_\epsilon \times 2$ will result in a number of configurations, two of which are considered in more detail.

$$\begin{aligned} \xi_{\text{init}} \rightarrow_! &\ (\text{pos}(r_1, sa) \wedge \text{pos}(r_2, sa) \wedge \text{pos}(v_1, sa))_{0.9\times 1.0} \times \pi \times \tau_1 \times 0 \\ &\vee (\text{pos}(r_1, sa) \wedge \text{pos}(r_2, sa) \wedge \text{pos}(v_2, sa))_{0.81\times 1.0} \times \pi \times \tau_2 \times 0 \\ &\vee \ldots \\ \text{where } \tau_1 &= \text{noop} \parallel \text{move}(r_2, p_1); t(r_1, v_1, sa) \parallel t(r_2, v_1, sa) \\ \tau_2 &= \text{move}(r_1, p_3) \parallel \text{move}(r_2, p_3); t(r_1, v_2, sa) \parallel t(r_2, v_2, sa) \end{aligned}$$

[9] The loop condition σ_ϵ will always subsume any state, as it is the identity element for states and subsumption is performed through Ax-matching.

τ_1 is arising from r_1 waiting and r_2 moving successfully to p_1 before they transport v_1 to the safety area, τ_2 represents the trace where both robots successfully move to p_3 and subsequently transport v_2 to the safety area. Figure 1 informally illustrates σ_{init} and the two traces τ_1 and τ_2.

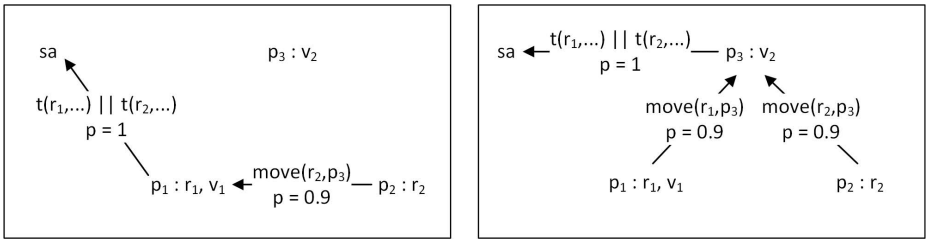

Fig. 1. Two traces τ_1 and τ_2 of executing $policy(r_1) \parallel policy(r_2)$ in state σ_{init}

For the robots to decide which choice is more valuable according to reward specification, the traces' expected values can be determined: $V_e(\tau_1, \xi_{\text{init}}) = 0.9$ and $V_e(\tau_2, \xi_{\text{init}}) = 0.81$. Therefore, deciding on τ_1 is the preferable choice of actions in the state σ_{init}.

Expected values and legal termination probabilities can also be used to prove that executing π in σ_{init} satisfies the PCTL property $\mathbb{P}_{\geq 0.9}(\Diamond^{\leq 2}\text{pos}(V, sa))$, as $\tau_1 \in T_{\downarrow}(\xi_{\text{init}})$ and $V_e(\tau_1, \xi_{\text{init}}) > 0 \wedge P^{+}_{\downarrow}(\tau_1, \xi_{\text{init}}) = 0.9$ (see section 4).

6 Related Work

As mentioned in section 2.2, prominent action programming formalisms have been developed: the situation calculus with its language GOLOG [8] and the fluent calculus with its language FLUX [10]. As already outlined, the situation calculus uses regressive fluent-wise specification of dynamics, which contrasts strongly with modern software design where dynamics are typically progressively defined operation-wise. Regarding this issue, the fluent calculus is more close to modern software design paradigms, as is the approach presented in this paper. While there is a decision-theoretic variant of GOLOG called DT-GOLOG employing the situation calculus [15], and there is a decision-theoretic extension to the fluent calculus (the *probabilistic fluent calculus* [16]), there is no procedural language exploiting decision-theoretic evaluation for a progressive action calculus, as FLUX is purely declarative. The relationship of classical planning and model checking has been investigated thoroughly [14], but, to the best of the author's knowledge, symbolic PCTL model checking approaches have not been investigated in the context of action programming in particular.

Both GOLOG and FLUX have been implemented in the PROLOG language, where specification of domain theories and action programs is done in terms of a logic program. While this is a reasonable approach for the reasoning tasks

tackled by these formalisms, PROLOG does not explicitly provide support for formal, algebraic software engineering like MAUDE [2,3]. Thus, MAUDE's features like modularization, explicit sort-hierarchies and polymorphism as well as meta-language operations and user-definable syntax provide additional properties to an action language specified in rewriting logic, as rewrite theories can be implemented in terms of MAUDE modules straightforwardly. Also, model checking as shown in section 4 is quite naturally supported by a MAUDE implementation due to its built-in *search* operator.

7 Conclusion and Further Work

7.1 Conclusion

This paper reports on efforts to integrate various aspects of action programming, algebraic software engineering and model checking to ease the specification of concurrent autonomous systems which expose formally verifiable behaviour and are able to evaluate action alternatives in a decision-theoretic manner, allowing them to autonomously take decisions at runtime that are sensible w.r.t. specified system goals. Domain knowledge (i.e. a relational MDP) is specified as a rewrite theory; system behaviour in terms of a procedural action program. Decision-theoretic evaluation and model-checking of action programs are performed by matching and rewriting modulo the given rewrite theory. Solutions have been provided for typical problems of symbolically reasoning systems like the frame, projection and legality problems. The approach supports true concurrency in the presence of sharing, and provides facilities to specify synchronization and coordination of actions. Symbolic PCTL model checking of typical system properties has been discussed. Action programming in rewriting logic without support for decision-theoretic concepts has been implemented in the MAUDE language [9].[10]

7.2 Further Work

Some directions for further work remain. It would be interesting to provide support for situations with incomplete knowledge, where properties of the environment have to be actively sensed. To this end, sensing actions have to be induced to plans, possibly considering sensing costs and expected rewards.

While explicit specification of concurrency has been discussed in this paper, evaluation of concurrent programs is performed centralized. In large-scale multi-agent domains it is valuable to decentralize the decision making process by localizing and distributing knowledge. In order to coordinate behaviour, sophisticated communication and interaction models (e.g. SCEL [17]) should be integrated with reasoning abilities of agents.

Finally, the relation of action programming and model checking discussed in section 4 only treated a particular class of PCTL formulae. Generalizing this approach as done for classical planning [14] could allow for specification of more complex goals and model checking of more complex system properties.

[10] `http://www.pst.ifi.lmu.de/~belzner/action-programming/`

References

1. Thielscher, M.: Action Programming Languages. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers (2008)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. 285(2), 187–243 (2002)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
5. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming, 1st edn. John Wiley & Sons, Inc., New York (1994)
6. Boutilier, C., Reiter, R., Price, B.: Symbolic dynamic programming for first-order mdps. In: Nebel, B. (ed.) IJCAI, pp. 690–700. Morgan Kaufmann (2001)
7. Mccarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence 4, 463–502 (1969)
8. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, illustrated edn. The MIT Press, Massachusetts (2001)
9. Belzner, L.: Action programming in rewriting logic. TPLP 13(4-5-online-suppl.) (2013)
10. Thielscher, M.: Flux: A logic programming method for reasoning agents. TPLP 5(4-5), 533–565 (2005)
11. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. Electronic Notes in Theoretical Computer Science 174(11), 3–25 (2007)
12. Eker, S.: Associative-commutative rewriting on large terms. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 14–29. Springer, Heidelberg (2003)
13. Baier, C., Katoen, J.P., et al.: Principles of model checking, vol. 26202649. MIT Press, Cambridge (2008)
14. Giunchiglia, F., Traverso, P.: Planning as model checking. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS (LNAI), vol. 1809, pp. 1–20. Springer, Heidelberg (2000)
15. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S., et al.: Decision-theoretic, high-level agent programming in the situation calculus. In: AAAI/IAAI, pp. 355–362 (2000)
16. Hölldobler, S., Skvortsova, O.: A logic-based approach to dynamic programming. In: Proceedings of the Workshop on Learning and Planning in Markov Processes–Advances and Challenges at the Nineteenth National Conference on Artificial Intelligence (AAAI 2004), pp. 31–36 (2004)
17. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A language-based approach to autonomic computing. In: Beckert, B., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 25–48. Springer, Heidelberg (2012)