On Supervising Agents in Situation-Determined ConGolog

Giuseppe De Giacomo
Sapienza Univ. Roma
Rome, Italy
degiacomo@dis.uniroma1.it

Yves Lespérance
York University
Toronto, ON, Canada
lesperan@cse.yorku.ca

Christian Muise
University of Toronto
Toronto, ON, Canada
cjmuise@cs.toronto.edu

ABSTRACT

We investigate agent supervision, a form of customization, which constrains the actions of an agent so as to enforce certain desired behavioral specifications. This is done in a setting based on the Situation Calculus and a variant of the ConGolog programming language which allows for nondeterminism, but requires the remainder of a program after the execution of an action to be determined by the resulting situation. Such programs can be fully characterized by the set of action sequences that they generate. Hence operations like intersection and difference become natural. The main results of the paper are a characterization of the maximally permissive supervisor that minimally constrains the agent so as to enforce the desired behavioral constraints when some agent actions are uncontrollable, and a sound and complete technique to execute the agent as constrained by such a supervisor.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms

Languages, Theory, Verification

Keywords

Agent Reasoning::Knowledge representation, Agent theories, Models and Architectures::Logic-based approaches and methods, Agent Reasoning::Reasoning (single and multi-agent)

1. INTRODUCTION

There has been much work on *process customization*, where a generic process for performing a task or achieving a goal is customized to satisfy a client's constraints or preferences [9, 11, 16]. Process customization gained special momentum in the context of web service composition [17]. For example in [12], a generic process provides a wide range of alternative ways to perform a task. During customization, alternatives that violate the constraints are eliminated. Some parameters in the remaining alternatives may be restricted or instantiated so as to ensure that any execution of the customized process will satisfy the client's constraints. A related approach to service composition synthesizes an orchestrator that controls the execution of a set of available services to ensure that they realize a desired service [15, 1].

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), June, 4–8, 2012, Valencia, Spain.

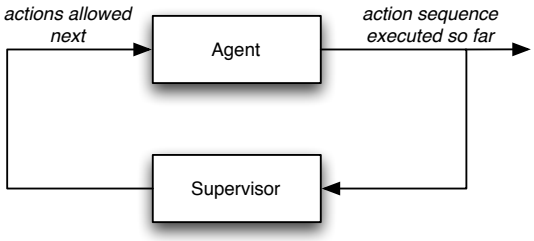

Figure 1: Supervised execution loop.

In this paper, we develop a framework for a similar type of process refinement that we call *supervised execution* (see Figure 1). We assume that we have a nondeterministic process that specifies the possible behaviors of an agent, and a second process that specifies the possible behaviors that a supervisor wants to allow (or alternatively, of the behaviors that it wants to rule out). For example, we could have an agent process representing a child and its possible behaviors, and a second process representing a babysitter that specifies the behaviors by the child that can be allowed. If the supervisor can control all the actions of the supervised agent, then it is straightforward to specify the behaviors that may result as a kind of synchronized concurrent execution of the agent and supervisor processes. A more interesting case arises when some agent actions are *uncontrollable*. For example, it may be impossible to prevent the child from getting muddy once he/she is allowed outside. In such circumstances, the supervisor may have to block some agent actions, not because they are undesirable in themselves (e.g., going outside), but because if they are allowed, the supervisor cannot prevent the agent from possibly performing some undesirable actions later on (e.g., getting muddy).

We follow previous work [12, 9] in assuming that processes are specified in a high level agent programming language defined in the Situation Calculus [14].[1] In fact, we define and use a restricted version of the ConGolog agent programming language [3] that we call Situation-Determined ConGolog (SDConGolog). In this version, following [4] all transitions involve performing an action (i.e., there are no transitions that merely perform a test). Moreover, nondeterminism is restricted so that the remaining program is a function of the action performed, i.e., given a program δ in a situation

[1]Clearly, there are applications where a declarative formalism is preferable, e.g., linear temporal logic (LTL), regular expressions over actions, or some type of business rules. However, there has been previous work on compiling such declarative specification languages into ConGolog, for instance [9], which handles an extended version of LTL interpreted over a finite horizon.

s, executing an action a (allowed by the program and the situation) results in a unique successor situation $s' = do(a, s)$ and a unique remaining program $\delta' = next(\delta, s, a)$, where $next$ is a function (formally defined later in this paper) that takes as arguments only δ, s, and a. This means that a run of such a program starting in a given situation can be taken to be simply a sequence of actions, as all the intermediate programs one goes through during the execution are functionally determined by the starting program and situation and the actions performed. Thus we can see a program in a situation as specifying a language formed by all the sequences of actions that are runs of the program in the situation. This allows us to define language theoretic notions such as union of languages, intersection, and difference/complementation in terms of operations on the corresponding programs, which has applications in many areas (e.g., programming by demonstration and programming by instruction [8], and plan recognition [6, 10]). We note that in [4], it is (implicitly) shown that any ConGolog program can be made situation-determined by recording nondeterministic choices as additional actions. Working with situation-determined programs also facilitates the formalization of supervision/customization. In particular it allows us to build on on the well-known Wonham and Ramadge framework for supervisory control of discrete event systems [13, 19, 2, 18].

Based on such a framework, we provide a general characterization of the *maximally permissive supervisor* that minimally constrains the actions of the agent specified in SDConGolog so as to enforce the desired behavioral specifications, showing its existence and uniqueness; secondly, we define a special program construct for *supervised execution* that takes the agent program and supervisor program, and executes them to obtain only runs allowed by the maximally permissive supervisor, showing its soundness and completeness.

The rest of the paper proceeds as follows. In the next section, we briefly review the Situation Calculus and the ConGolog agent programming language. In Section 3, we define SDConGolog, discuss its properties, and introduce some useful programming constructs and terminology. Then in Section 4, we develop our account of agent supervision, and define the maximally permissive supervisor and supervised execution. Finally in Section 5, we conclude the paper also discussing implementation.

2. PRELIMINARIES

The *situation calculus* is a logical language specifically designed for representing and reasoning about dynamically changing worlds [14]. All changes to the world are the result of *actions*, which are terms in the logic. We denote action variables by lower case letters a, action types by capital letters A, and action terms by α, possibly with subscripts. A possible world history is represented by a term called a *situation*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol do, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Predicates and functions whose value varies from situation to situation are called *fluents*, and are denoted by symbols taking a situation term as their last argument (e.g., $Holding(x, s)$). Within the language, one can formulate action theories that describe how the world changes as the result of actions [14].

To represent and reason about complex actions or processes obtained by suitably executing atomic actions, various so-called *high-level programming languages* have been defined. Here we concentrate on (a fragment of) ConGolog that includes the following constructs:

α	atomic action
$\varphi?$	test for a condition
$\delta_1; \delta_2$	sequence
if φ **then** δ_1 **else** δ_2	conditional
while φ **do** δ	while loop
$\delta_1 \vert \delta_2$	nondeterministic branch
$\pi x.\delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
$\delta_1 \Vert \delta_2$	concurrency

In the above, α is an action term, possibly with parameters, and φ is situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed. As usual, we denote by $\varphi[s]$ the situation calculus formula obtained from φ by restoring the situation argument s into all fluents in φ. Program $\delta_1 | \delta_2$ allows for the nondeterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* nondeterministic choice of a legal binding for variable x (observe that such a choice is, in general, unbounded). δ^* performs δ zero or more times. Program $\delta_1 \Vert \delta_2$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2.

Formally, the semantics of ConGolog is specified in terms of single-step transitions, using the following two predicates [3]: *(i)* $Trans(\delta, s, \delta', s')$, which holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed; and *(ii)* $Final(\delta, s)$, which holds if program δ may legally terminate in situation s. The definitions of $Trans$ and $Final$ we use are as in [4]; these are in fact the usual ones [3], except that the test construct $\varphi?$ does not yield any transition, but is final when satisfied. Thus, it is a *synchronous* version of the original test construct (it does not allow interleaving). As a consequence, in the version of ConGolog that we use, every transition involves the execution of an action (tests do not make transitions), i.e.,

$$\Sigma \cup \mathcal{C} \models \mathit{Trans}(\delta, s, \delta', s') \supset \exists a.s' = do(a, s).$$

Here and in the remainder, we use Σ to denote the foundational axioms of the situation calculus from [14] and $\mathcal{C}$ to denote the axioms defining the ConGolog programming language.

3. SD-PROGRAMS

We focus on a restricted class of ConGolog programs to describe processes, namely "situation-determined programs"; we call this class SDConGolog. A program δ is *situation-determined* in a situation s if for every sequence of transitions, the remaining program is determined by the resulting situation, i.e.,

$$SituationDetermined(\delta, s) \doteq \forall s', \delta', \delta''.\\ \mathit{Trans}^*(\delta, s, \delta', s') \wedge \mathit{Trans}^*(\delta, s, \delta'', s') \supset \delta' = \delta'',$$

where *Trans** denotes the reflexive transitive closure of *Trans*. Thus, a (possibly partial) execution of a situation-determined program is uniquely determined by the sequence of actions it has produced. This is a key point. In general, the possible executions of a ConGolog program are characterized by sequences of configurations formed by the remaining program and the current situation. In contrast, the *execution of a SDConGolog program in a situation can be characterized in terms of a set of sequences (or language) of actions*. Such sequences correspond to situations reached from the situation where the program started.

For example, assuming for simplicity that all actions are executable in every situation, the ConGolog program $(a; b) \mid (a; c)$ is not situation-determined in situation S_0 as it can make a transition to a configuration $(b, do(a, S_0))$, where the situation is $do(a, S_0)$ and the remaining program is b, and it can also make a transition to a configuration $(c, do(a, S_0))$, where the situation is also $do(a, S_0)$

and the remaining program is instead c. It is impossible to determine what the remaining program is given only a situation, e.g. $do(a, S_0)$, reached along an execution. In contrast, the program $a; (b \mid c)$ is situation-determined in situation S_0. There is a unique remaining program $(b \mid c)$ in situation $do(a, S_0)$ (and similarly for the other reachable situations).

When we restrict our attention to situation-determined programs, we can use a simpler semantic specification for the language; instead of *Trans* we can use a $next$ (partial) function, where $next(\delta, a, s)$ returns the program that remains after δ does a transition involving action a in situation s (if δ is situation determined, such a remaining program must be unique). We will axiomatize the $next$ function so that it satisfies the following properties:

$$next(\delta, a, s) = \delta' \wedge \delta' \neq \bot \supset \mathit{Trans}(\delta, s, \delta', do(a, s)) \text{ (N1)}$$

$$\exists!\delta'.\mathit{Trans}(\delta, s, \delta', do(a, s)) \supset \forall\delta'.(\mathit{Trans}(\delta, s, \delta', do(a, s)) \supset next(\delta, a, s) = \delta') \text{ (N2)}$$

$$\neg\exists!\delta'.\mathit{Trans}(\delta, s, \delta', do(a, s)) \supset next(\delta, a, s) = \bot \text{ (N3)}$$

Here $\exists!x.\phi(x)$ means that there exists a unique x such that $\phi(x)$; this is defined in the usual way. $\bot$ is a special value that stands for "undefined". The function $next(\delta, a, s)$ is only defined when there is a unique remaining program after program δ does a transition involving the action a; if there is such a unique remaining program, then $next(\delta, a, s)$ denotes it.

We define the function $next$ inductively on the structure of programs using the following axioms.

Atomic action:

$$next(\alpha, a, s) = \begin{cases} nil & \text{if } Poss(a, s) \text{ and } \alpha = a \\ \bot & \text{otherwise} \end{cases}$$

Sequence: $next(\delta_1; \delta_2, a, s) =$

$$\begin{cases} next(\delta_1, a, s); \delta_2 & \text{if } next(\delta_1, a, s) \neq \bot \text{ and } (\neg Final(\delta_1, s) \text{ or } next(\delta_2, a, s) = \bot) \\ next(\delta_2, a, s) & \text{if } \mathit{Final}(\delta_1, s) \text{ and } next(\delta_1, a, s) = \bot \\ \bot & \text{otherwise} \end{cases}$$

Conditional:

$$next(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2, a, s) = \begin{cases} next(\delta_1, a, s) & \text{if } \varphi[s] \\ next(\delta_2, a, s) & \text{if } \neg\varphi[s] \end{cases}$$

Loop:

$$next(\textbf{while } \varphi \textbf{ do } \delta, a, s) = \begin{cases} next(\delta, a, s); \textbf{while } \varphi \textbf{ do } \delta & \text{if } \varphi[s] \text{ and } next(\delta, a, s) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Nondeterministic branch:

$$next(\delta_1 | \delta_2, a, s) = \begin{cases} next(\delta_1, a, s) & \text{if } next(\delta_2, a, s) = \bot \text{ or } next(\delta_2, a, s) = next(\delta_1, a, s) \\ next(\delta_2, a, s) & \text{if } next(\delta_1, a, s) = \bot \\ \bot & \text{otherwise} \end{cases}$$

Nondeterministic choice of argument:[2]

$$next(\pi x.\delta, a, s) = \begin{cases} next(\delta^x_d, a, s) & \text{if } \exists!d.next(\delta^x_d, a, s) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

[2]Notice that d in δ^x_d depends on a and s. In particular d may be instantiated by the action a in s. Read on for an example.

Nondeterministic iteration:

$$next(\delta^*, a, s) = \begin{cases} next(\delta, a, s); \delta^* & \text{if } next(\delta, a, s) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Interleaving concurrency: $next(\delta_1 \| \delta_2, a, s) =$

$$\begin{cases} next(\delta_1, a, s) \| \delta_2 & \text{if } next(\delta_1, a, s) \neq \bot \text{ and } next(\delta_2, a, s) = \bot \\ \delta_1 \| next(\delta_2, a, s) & \text{if } next(\delta_2, a, s) \neq \bot \text{ and } next(\delta_1, a, s) = \bot \\ \bot & \text{otherwise} \end{cases}$$

Test, empty program, undefined:

$$next(\varphi?, a, s) = \bot \quad next(nil, a, s) = \bot \quad next(\bot, a, s) = \bot$$

The undefined program is never *Final*: $\mathit{Final}(\bot, s) \equiv \texttt{false}$.

Let $\mathcal{C}^n$ be the set of ConGolog axioms extended with the above axioms specifying $next$ and $\mathit{Final}(\bot, s)$. It is easy to show that:

PROPOSITION 1. *N1, N2, and N3 are entailed by* $\Sigma \cup \mathcal{C}^n$.

Note in particular that as per N3, if the remaining program is not uniquely determined, then $next(\delta, a, s)$ is undefined. Notice that for situation-determined programs this will never happen, and if $next(\delta, a, s)$ returns $\bot$ it is because δ cannot make any transition using a in s:

COROLLARY 2.

$$\Sigma \cup \mathcal{C}^n \models \forall\delta, s.\mathit{SituationDetermined}(\delta, s) \supset \forall a\,[(next(\delta, a, s) = \bot) \equiv (\neg\exists\delta'.\mathit{Trans}(\delta, s, \delta', do(a, s)))].$$

Let's look at an example. Imagine an agent specified by δ_{B1} below that can repeatedly pick an available object and repeatedly use it and then discard it, with the proviso that if during use the object breaks, the agent must repair it:

$$\delta_{B1} = [\pi\, x.Available(x)?; [use(x); (nil \mid [break(x); repair(x)])]^*; discard(x)]^*$$

We assume that there is a countably infinite number of available unbroken objects initially, that objects remain available until they are discarded, that available objects can be used if they are unbroken, and that objects are unbroken unless they break and are not repaired (this is straightforwardly axiomatized in the situation calculus). Notice that this program is situation-determined, though very nondeterministic.

Language theoretic operations on programs. We extend the SDConGolog language so as to close it with respect to language theoretic operations, such as *union*, *intersection* and *difference/complementation* of sets of sequences of actions. We can already see the nondeterministic branch construct as a union operator, and intersection and difference can be defined as follows.

Intersection/synchronous concurrency:

$$next(\delta_1\ \&\ \delta_2, a, s) = \begin{cases} next(\delta_1, a, s)\ \&\ next(\delta_2, a, s) & \text{if both are different from } \bot \\ \bot & \text{otherwise} \end{cases}$$

Difference: $next(\delta_1 - \delta_2, a, s) =$

$$\begin{cases} next(\delta_1, a, s) - next(\delta_2, a, s) & \text{if both are different from } \bot \\ next(\delta_1, a, s) & \text{if } next(\delta_2, a, s) = \bot \\ \bot & \text{if } next(\delta_1, a, s) = \bot \end{cases}$$

For these new constructs, *Final* is defined as follows:

$$\mathit{Final}(\delta_1\ \&\ \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s)$$

$$\mathit{Final}(\delta_1 - \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \neg\mathit{Final}(\delta_2, s)$$

We can express the *complement* of a program δ using difference as follows: $(\pi a.a)^* - \delta$.

It is easy to check that Proposition 1 and Corollary 2 also hold for programs involving these new constructs.[3]

As we will see later, synchronous concurrency can be used to constrain/customize a process. Difference can be used to prohibit certain process behaviors: $\delta_1 - \delta_2$ is the process where δ_1 is executed but δ_2 is not. To illustrate, consider an agent specified by program δ_{S1} that repeatedly picks an available object and does anything to it provided it is broken at most once before it is discarded:

$$\begin{array}{l}\delta_{S1} = [\pi\, x.Available(x)?;\\ \quad [\pi\, a.(a{-}(break(x) \mid discard(x)))]^*;\\ \quad (nil \mid (break(x)); [\pi\, a.(a{-}(break(x) \mid discard(x)))]^*);\\ \quad discard(x)]^*\end{array}$$

Sequences of actions generated by programs. We now characterize situation determined programs in terms of sequences of actions (runs) that may be performed from a given starting situation. This allows us to associate to such programs a language formed by such sequences of actions. Notice that in most cases not only the language will be infinite, but even the alphabet on which the language is defined will be infinite, since it is formed by all actions obtained by substituting values for parameters in the action types.

We start by extending the function $next$ to deal with sequences of actions. We assume sequences of actions to be inductively defined on their length as follows: *(i)* ϵ is the sequence of action of length 0; *(ii)* given a sequence of actions $\vec{a}$ of length n, and an action a the sequence $a\vec{a}$ is a sequence of actions of length $n+1$. (Notice that we are considering only finite sequences of actions in this way.)[4] When convenient, we will use the notation $\vec{a}\vec{b}$ to denote the sequence of actions formed by concatenating the two subsequences $\vec{a}$ and $\vec{b}$. As a special case we use also $\vec{a}a$ where a is an action.

The extension of the function $next$ to sequences of actions is a function $next^*(\delta, \vec{a}, s)$ that takes a program δ, a sequence of actions $\vec{a}$, and a situation s, and returns the remaining program δ' after executing δ in s producing the sequence of actions $\vec{a}$, defined by induction on the length of the sequence of actions as follows:

$$\begin{array}{l}next^*(\delta, \epsilon, s) = \delta\\ next^*(\delta, a\vec{a}, s) = next^*(next(\delta, a, s), \vec{a}, do(a, s))\end{array}$$

where ϵ denotes the empty sequence. Note that if along $\vec{a}$ the program becomes $\perp$ then $next^*$ returns $\perp$ as well.

Runs. We define the set $\mathcal{RR}(\delta, s)$ of (partial) *runs* of a program δ in a situation s as the sequences of actions that can be produced by executing δ from s:[5]

$$\mathcal{RR}(\delta, s) = \{\vec{a} \mid next^*(\delta, \vec{a}, s) \neq \perp\}$$

Note that if $\vec{a} \in \mathcal{RR}(\delta, s)$, then all prefixes of $\vec{a}$ are in $\mathcal{RR}(\delta, s)$ as well.

[3]We may extend the definition of $Trans$ to the new constructs $\circ \in \{\&, -\}$ as follows $Trans(\delta_1 \circ \delta_2, s, \delta', s') \equiv \exists a.s = do(a, s) \wedge \delta' = next(\delta_1 \circ \delta_2, a, s) \neq \perp$.

[4]Notice that such sequences of actions have to be axiomatized in second-order logic in a standard way, similarly to situations. Also as an alternative, sequences of actions could also be characterized directly in terms of "difference" between situations. For sake of brevity, we leave out the formalization of sequences of actions here, and just assume that such sequences have been fully characterized.

[5]Here and in what follows, we use set notation for readability; if we wanted to be very formal, we could introduce $\mathcal{RR}$ as a defined predicate, and similarly for $\mathcal{CR}$, etc.

Complete runs. Notice that not all runs in $\mathcal{RR}$ reach eventually a final configuration. We are interested in distinguishing those runs that do. Hence we define the set of complete runs and that of their prefixes, called the good runs. We define the set $\mathcal{CR}(\delta, s)$ of *complete runs* of a program δ in a situation s as the sequences of actions that can be produced by executing δ from s until a *Final* configuration is reached:

$$\mathcal{CR}(\delta, s) = \{\vec{a} \mid Final(next^*(\delta, \vec{a}, s), do(\vec{a}, s))\}$$

As an alternative characterization, we observe that for every sequence of actions $\vec{a}$, we have $\vec{a} \in \mathcal{CR}(\delta, s)$ iff $Do(\delta, s, do(\vec{a}, s))$ using the usual terminology of [14, 3].

Good runs. We define the set $\mathcal{GR}(\delta, s)$ of *good runs* of a program δ in a situation s as the sequences of actions that can be produced by executing δ from s which can be *extended* until a *Final* configuration is reached:

$$\mathcal{GR}(\delta, s) = \{\vec{a} \mid \exists \vec{b}.Final(next^*(\delta, \vec{a}\vec{b}, s), do(\vec{a}\vec{b}, s))\}$$

In other words $\vec{a} \in \mathcal{GR}(\delta, s)$ if $\vec{a}$ is a prefix of a sequence of action $\vec{a'} = \vec{a}\vec{b}$ such that $\vec{a'} \in \mathcal{CR}(\delta, s)$.

$\mathcal{RR}(\delta, s)$, $\mathcal{CR}(\delta, s)$, and $\mathcal{GR}(\delta, s)$ can be considered as three languages (of sequences of actions) generated by the program δ in s. This allows us to apply language theoretic notions to situation-determined programs, and we exploit this possibility for studying supervision.

Before turning to that, let's make a few observations. First, it is easy to see that $\mathcal{CR}(\delta, s) \subseteq \mathcal{GR}(\delta, s) \subseteq \mathcal{RR}(\delta, s)$, i.e., complete runs are good runs, and good runs are indeed runs. Moreover, $\mathcal{CR}(\delta, s) = \mathcal{CR}(\delta', s)$ implies $\mathcal{GR}(\delta, s) = \mathcal{GR}(\delta', s)$, i.e., if two programs in a situation have the same complete runs, then they also have the same good runs; however they may still differ in their sets of non-good runs, since $\mathcal{CR}(\delta, s) = \mathcal{CR}(\delta', s)$ does not imply $\mathcal{RR}(\delta, s) = \mathcal{RR}(\delta', s)$. We say that a program δ in s is *non-blocking* iff $\mathcal{RR}(\delta, s) = \mathcal{GR}(\delta, s)$, i.e., if all runs of the program δ in s can be extended to runs that reach a *Final* configuration.

Search construct. Interestingly in the literature on Situation Calculus based programs, a special construct $\Sigma(\delta)$ was introduced to ensure that the only actions produced by a program δ are those that eventually lead to a final state. This is the so called "search construct" [5].

We can add such a construct to SDConGolog, and use it to generate only good runs. The search construct Σ is characterized in terms of $next$ as follows:

$$next(\Sigma(\delta), a, s) = \begin{cases} \Sigma(next(\delta, a, s)) & \text{if there exists } \vec{a} \text{ s.t.}\\ \quad Final(next^*(\delta, a\vec{a}, s)) & \\ \perp & \text{otherwise}\end{cases}$$

$$Final(\Sigma(\delta), s) \equiv Final(\delta, s).$$

Intuitively, $next(\Sigma(\delta), a, s)$ does lookahead to ensure that action a is in a good run of δ in s, otherwise it returns $\perp$.

Notice that: *(i)* $\mathcal{RR}(\Sigma(\delta), s) = \mathcal{GR}(\Sigma(\delta), s)$, i.e., under the search construct all programs are non-blocking; *(ii)* $\mathcal{RR}(\Sigma(\delta), s) = \mathcal{GR}(\delta, s)$, i.e., $\Sigma(\delta)$ produces exactly the good runs of δ; *(iii)* $\mathcal{CR}(\Sigma(\delta), s) = \mathcal{CR}(\delta, s)$, i.e., $\Sigma(\delta)$ and δ produce exactly the same set of complete runs. Thus $\Sigma(\delta)$ trims the behavior of δ by eliminating all those runs that do not lead to a *Final* configuration.

Note also that if a program is *non-blocking* in s, then $\mathcal{RR}(\Sigma(\delta), s) = \mathcal{RR}(\delta, s)$, in which case there is no point in

using the search construct. Finally, we have that: $\mathcal{CR}(\delta, s) = \mathcal{CR}(\delta', s)$ implies $\mathcal{RR}(\Sigma(\delta), s) = \mathcal{RR}(\Sigma(\delta'), s)$, i.e., if two programs have the same complete runs, then under the search construct they have exactly the same runs.

4. SUPERVISION

Let us assume that we have two agents: an agent B with behavior represented by the program δ_B and a supervisor S with behavior represented by δ_S. While both are represented by programs, the roles of the two agents are quite distinct. The first is an agent B that acts freely within its space of deliberation represented by δ_B. The second, S, is supervising B so that as B acts, it remains within the behavior permitted by S. This role makes the program δ_S act as a specification of allowed behaviors for agent B.[6]

The behavior of B under the supervision of S is constrained so that at any point B can execute an action in its original behavior, only if such an action is also permitted in S's behavior. Using the synchronous concurrency operator, this can be expressed simply as:

$$\delta_B \ \& \ \delta_S.$$

Note that unless $\delta_B \ \& \ \delta_S$ happens to be non-blocking, it may get stuck in dead end configurations. To avoid this, we need to apply the search construct, getting $\Sigma(\delta_B \ \& \ \delta_S)$. In general, the use of the search construct to avoid blocking, is always needed in the development below.

We can use the example programs presented earlier to illustrate. The execution of δ_{B1} under the supervision of δ_{S1} is simply $\delta_{B1} \ \& \ \delta_{S1}$ (assuming all actions are controllable). It is straightforward to show that the resulting behavior is to repeatedly pick an available object and use it as long as one likes, breaking it at most once, and repairing it whenever it breaks, before discarding it. It can be shown that the set of partial/complete runs of $\delta_{B1} \ \& \ \delta_{S1}$ is exactly that of:

$$\begin{array}{l}[\pi\, x.Available(x)?;\\ \quad use(x)^*;\\ \quad [nil \mid (break(x); repair(x); use(x)^*)];\\ \quad discard(x)]^*\end{array}$$

Uncontrollable actions. In the above, we implicitly assumed that all actions of agent B could be controlled by the supervisor S. This is often too strong an assumption, e.g., once we let a child out in a garden after rain, there is nothing we can do to prevent her/him from getting muddy. We now want to deal with such cases.

To do this, we follow the general approach of the well-known Wonham and Ramadge (W&R) framework for supervisory control of discrete event systems [13, 19, 2, 18], suitably extended to deal with rich languages such as those generated by SDConGolog programs.

We start by distinguishing between actions that are *controllable* by the supervisor and actions that are *uncontrollable*. The supervisor can block the execution of the controllable actions, but cannot prevent the supervised agent from executing the uncontrollable ones.

To characterize the uncontrollable actions in the situation calculus, we use a special fluent $A_u(a_u, s)$, which we call an *action filter*, that expresses that action a_u is uncontrollable in situation s. Notice that, unlike in the W&R framework, we allow controllability to be *context dependent* by allowing an arbitrary specification of the fluent $A_u(a_u, s)$ in the situation calculus.

While we would like the supervisor S to constrain agent B so that $\delta_B \ \& \ \delta_S$ is executed, in reality, since S cannot prevent uncontrollable actions, S can only constrain B on the controllable actions. When this is sufficient, we say that the supervison specification δ_S is "controllable". Technically, following again W&R, this can be captured by saying that the *supervision specification* δ_S *is controllable wrt* δ_B *in situation* s iff:

$$\begin{array}{l}\forall \vec{a} a_u.\vec{a} \in \mathcal{GR}(\delta_S, s) \text{ and } A_u(a_u, do(\vec{a}, s)) \text{ implies}\\ \quad \text{if } \vec{a} a_u \in \mathcal{GR}(\delta_B, s) \text{ then } \vec{a} a_u \in \mathcal{GR}(\delta_S, s).\end{array}$$

What this says is that if we postfix a good run $\vec{a}$ for S with an uncontrollable action a_u that is good for B (and so $\vec{a}$ must be good for B as well), then this uncontrollable action a_u must also be good for S. By the way, notice that $\vec{a} a_u \in \mathcal{GR}(\delta_B, s)$ and $\vec{a} a_u \in \mathcal{GR}(\delta_S, s)$ together imply that $\vec{a} a_u \in \mathcal{GR}(\delta_B \ \& \ \delta_S, s)$.

What about if such a property does not hold? We can take two orthogonal approaches: *(i)* simply relax δ_S so that it places no constraints on the uncontrollable actions; *(ii)* require that δ_S be indeed enforced, but also disallow those runs that prevent δ_S from being controllable. We look at both approaches below.

Relaxed supervision. To define relaxed supervision we first need to introduce two operations on programs: *projection* and, based on it, *relaxation*. The *projection operation* takes a program and an *action filter* A_u, and projects all the actions that satisfy the action filter (e.g., are uncontrollable), out of the execution. To do this, projection substitutes each occurrence of an atomic action term α_i by a conditional statement that replaces it with the trivial test `true`? when $A_u(\alpha_i)$ holds in the current situation, that is:

$$\begin{array}{l}pj(\delta, A_u) = \delta^{\alpha_i}_{\textbf{if } A_u(\alpha_i) \textbf{ then } \texttt{true?} \textbf{ else } \alpha_i}\\ \text{for every occurrence of an action term } \alpha_i \text{ in } \delta.\end{array}$$

(Recall that such a test does not perform any transition in our variant of ConGolog.)

The *relaxation operation* on δ wrt $A_u(a, s)$ is as follows:

$$rl(\delta, A_u) = pj(\delta, A_u) \| (\pi a.A_u(a)?; a)^*.$$

In other words, we project out the actions in A_u from δ and run the resulting program concurrently with one that picks (uncontrollable) actions filtered by A_u and executes them. The resulting program no longer constrains the occurrence of actions from A_u in any way. In fact, notice that the remaining program of $(\pi a.A_u(a)?; a)^*$ after the execution of an (uncontrollable) filtered action is $(\pi a.A_u(a)?; a)^*$ itself, and that such a program is always *Final*.

Now we are ready to define *relaxed supervision*. Let us consider a supervisor S with supervision specitfication δ_S for agent B with behavior δ_B. Let the action filter $A_u(a_u, s)$ specify the uncontrollable actions. Then the *relaxed supervision* of δ_S (for $A_u(a_u, s)$) in s is the relaxation of δ_S so as that it allows every uncontrollable action, namely: $rl(\delta_S, A_u)$. So we can characterize the behavior of B under the relaxed supervision of S as:

$$\delta_B \ \& \ rl(\delta_S, A_u).$$

The following properties are immediate consequences of the definitions:

PROPOSITION 3. *The relaxed supervision* $rl(\delta_S, A_u)$ *is controllable wrt* δ_B *in situation* s.

PROPOSITION 4. $\mathcal{CR}(\delta_B \ \& \ \delta_S, s) \subseteq \mathcal{CR}(\delta_B \ \& \ rl(\delta_S, A_u), s)$.

[6] Note that, because of these different roles, one may want to assume that all configurations generated by (δ_S, s) are $Final$, so that we leave B unconstrained on when it may terminate. This amounts to requiring the following property to hold: $\mathcal{CR}(\delta_S, s) = \mathcal{GR}(\delta_S, s) = \mathcal{RR}(\delta_S, s)$. While reasonable, for the technical development below, we do not need to rely on this assumption.

PROPOSITION 5. *If* $\mathcal{CR}(\delta_B \ \& \ rl(\delta_S, A_u), s) \subseteq \mathcal{CR}(\delta_B \ \& \ \delta_S, s)$*, then* δ_S *is controllable wrt* δ_B *in situation* s*.*

Notice that, the first one is what we wanted. But the second one says that $rl(\delta_S, A_u)$ may indeed by more permissive than δ_S: some complete runs that are disallowed in δ_S may be permitted by its relaxation $rl(\delta_S, A_u)$. This is often not acceptable. Proposition 5 says that when the converse of Proposition 4 holds, we have that the original supervision δ_S is indeed controllable wrt δ_B in situation s. Notice however that even if δ_S is controllable wrt δ_B in situation s, it may still be the case that $\mathcal{CR}(\delta_B \ \& \ rl(\delta_S, A_u), s) \subset \mathcal{CR}(\delta_B \ \& \ \delta_S, s)$.

Maximally permissive supervisor. Next we study a more interesting, more conservative approach: we require the supervision specification δ_S to be fulfilled, and for getting controllability we further restrict the specification if needed. This is the classical approach adopted in the W&R framework, and indeed, we are able to show that the key result of the W&R framework is preserved in our generalized setting: there is, in principle, a unique maximally permissive way of restricting the supervision specification so that it still fulfills δ_S while being controllable. We call the resulting supervisor the *maximally permissive supervisor*.

To phrase this result in our setting, however, we need to augment our programming language with a new construct $\mathbf{set}(E)$ that takes an arbitrary set of sequences of actions E and makes it a program. For such a construct $next$ and *Final* are defined as follows:

$$next(\mathbf{set}(E), a, s) = \begin{cases} \mathbf{set}(E') \ \text{ with } E' = \{\vec{a} \mid a\vec{a} \in E\} \\ \qquad \text{if } E' \neq \emptyset \\ \bot \ \text{ if } E' = \emptyset \end{cases}$$

$$\mathit{Final}(\mathbf{set}(E), s) \equiv (\epsilon \in E)$$

Thus $\mathbf{set}(E)$ can be executed to produce any of the sequences of actions in E.

Notice that for every program δ and situation s, we can define $E_\delta = \mathcal{CR}(\delta, s)$ such that $\mathcal{CR}(\mathbf{set}(E_\delta), s) = \mathcal{CR}(\delta, s)$. The converse does not hold in general, i.e., there are programs $\mathbf{set}(E)$ such that for all programs δ, not involving the $\mathbf{set}(\cdot)$ construct, $\mathcal{CR}(\mathbf{set}(E_\delta), s) \neq \mathcal{CR}(\delta, s)$. That is, the syntactic restrictions in SDConGolog may not allow us to represent some possible sets of sequences of actions.[7]

With the $\mathbf{set}(E)$ construct at hand, following [19], we may define the *maximally permissive supervisor* $mps(\delta_B, \delta_S, s)$ of the agent behavior δ_B which fulfills the supervision specification δ_S in situation s, as:

$$mps(\delta_B, \delta_S, s) = \mathbf{set}(\textstyle\bigcup_{E \in \mathcal{E}} E) \ \text{ where}$$
$$\mathcal{E} = \{E \mid E \subseteq \mathcal{CR}(\delta_B \ \& \ \delta_S, s) \\ \text{and } \mathbf{set}(E) \textit{ is controllable wrt } \delta_B \textit{ in } s\}$$

Intuitively mps denotes the maximal set of runs that are effectively allowable by a supervisor that fulfills the specification δ_S, and which can be left to the arbitrary decisions of the agent behaving as δ_B on the uncontrollable actions. A quite interesting result is that, even in the general setting we are presenting, such a maximally permissive supervisor always *exists* and is *unique*. Indeed, we can show:

THEOREM 6. *For the maximally permissive supervisor* $mps(\delta_B, \delta_S, s)$ *the following properties hold:*

1. $mps(\delta_B, \delta_S, s)$ *always exists and is unique;*
2. $mps(\delta_B, \delta_S, s)$ *is controllable wrt* δ_B *in* s*;*
3. *For every possible controllable supervision specification* $\hat{\delta}_S$ *for* δ_B *in* s *such that* $\mathcal{CR}(\delta_B \ \& \ \hat{\delta}_S, s) \subseteq \mathcal{CR}(\delta_B \ \& \ \delta_S, s)$*, we have that* $\mathcal{CR}(\delta_B \ \& \ \hat{\delta}_S, s) \subseteq \mathcal{CR}(\delta_B \ \& \ mps(\delta_B, \delta_S, s), s)$*.*

PROOF. We prove the three claims separately.

Claim 1. It follows directly from the fact $\mathbf{set}(\emptyset)$ satisfies the conditions to be included in $mps(\delta_B, \delta_S, s)$.

Claim 2. It suffices to show that $\forall \vec{a}a_u.\vec{a} \in \mathcal{GR}(\delta_B \ \& \ mps(\delta_B, \delta_S, s), s)$ and $A_u(a_u, do(\vec{a}, s))$ we have that if $\vec{a}a_u \in \mathcal{GR}(\delta_B, s)$ then $\vec{a}a_u \in \mathcal{GR}(mps(\delta_B, \delta_S, s), s)$. Indeed, if $\vec{a} \in \mathcal{GR}(\delta_B \ \& \ mps(\delta_B, \delta_S, s), s)$ then there is an controllable supervision specification $\mathbf{set}(E)$ such that $\vec{a} \in \mathcal{GR}(\delta_B \ \& \ \mathbf{set}(E), \delta_S, s), s)$. $\mathbf{set}(E)$ being controllable wrt δ_B in s, if $\vec{a}a_u \in \mathcal{GR}(\delta_B, s)$ then $\vec{a}a_u \in \mathcal{GR}(\mathbf{set}(E), s)$, but then $\vec{a}a_u \in \mathcal{GR}(mps(\delta_B, \delta_S, s), s)$.

Claim 3. It follows immediately from the definition of $mps(\delta_B, \delta_S, s)$, by noticing that $\mathcal{CR}(\delta_B \ \& \ \hat{\delta}_S, s) = \mathcal{CR}(\delta_B \ \& \ \mathbf{set}(E_{\hat{\delta}_S}), s)$, and observing that $mps(\delta_B, \delta_S, s)$ is essentially the union of such controllable $\mathbf{set}(E_{\hat{\delta}_S})$. □

Returning to our running example, if we assume that the $break$ action is uncontrollable (and the others are controllable), the supervisor $S1$ can only ensure that its constraints are satisfied if it forces $B1$ to discard an object as soon as it is broken and repaired. This is what we get as maximally permissive supervisor $mps(\delta_{B1}, \delta_{S1}, S_0)$, whose set of runs can be shown to be exactly that of:

$$[\pi\, x.Available(x)?;$$
$$use(x)^*;$$
$$[nil \mid (break(x); repair(x))];$$
$$discard(x)]^*$$

By the way, notice that $(\delta_{B1} \ \& \ rl(\delta_{S1}, A_u))$ instead is completely ineffective since it has exactly the runs of δ_{B1}.

Unfortunately, in general, $mps(\delta_B, \delta_S, s)$ requires the use of the program construct $\mathbf{set}(E)$ which is mostly of theoretical interest. For this reason the above characterization remains essentially mathematical. So next, we develop a new construct for execution of programs under maximally permissive supervision, giving up on precomputing the maximally permissive supervision specification a priori and instead directly computing it online while the agent is operating.

Maximally permissive supervised execution. To capture the notion of maximally permissive execution of agent B with behavior δ_B under the supervision of S with behavior δ_S in situation s, we introduce a special version of the synchronous concurrency construct that takes into account the fact the some actions are uncontrollable. Without loss of generality, we assume that δ_B and δ_S both start with a common controllable action (if not, it is trivial to add a dummy action in front of both so as to fullfil the requirement). Then, we characterize the construct through $next$ and *Final* as follows:

[7] Obviously there are certain sets that can be expressed directly in SDConGolog , e.g., when E is finite. However notice that in the general case the object domain may be infinite, and $\mathbf{set}(E)$ may not be representable as a finitary SDConGolog program.

$$next(\delta_B \,\&_{A_u}\, \delta_S, a, s) = \begin{cases} next(\delta_B, a, s) \,\&_{A_u}\, next(\delta_S, a, s) & \text{if} \\ \quad next(\delta_B, a, s) \neq \bot \text{ and } next(\delta_S, a, s) \neq \bot \text{ and} \\ \quad \text{if } \neg A_u(a, s), \text{ then} \\ \quad\quad \text{for all } \vec{a_u} \text{ such that } A_u(\vec{a_u}, do(a, s)) \\ \quad\quad\quad \text{if } next^*(\Sigma(\delta_B), a\vec{a_u}, s) \neq \bot, \\ \quad\quad\quad \text{then } next^*(\Sigma(\delta_S), a\vec{a_u}, s) \neq \bot \\ \bot \quad \text{otherwise} \end{cases}$$

where $A_u(\vec{a_u}, s)$, meaning that action sequence $\vec{a_u}$ is uncontrollable in situation s, is inductively defined on the length of $\vec{a_u}$ as the smallest predicate such that: *(i)* $A_u(\epsilon, s) \equiv \texttt{true}$; *(ii)* $A_u(a_u\vec{a_u}, s) \equiv A_u(a_u, s) \wedge A_u(\vec{a_u}, do(a_u, s))$. Thus, the maximally permissive supervised execution of δ_B for the specification δ_S is allowed to perform action a in situation s if a is allowed by both δ_B and δ_S and moreover, if a is controllable, then for every sequence of uncontrollable actions $\vec{a_u}$, if $\vec{a_u}$ may be performed by δ_B right after a on one of its complete runs, then it must also be allowed by δ_S (on one of its complete runs). Essentially, an action a by the agent must be forbidden if it can be followed by some sequence of uncontrollable actions that violates the specification.

Final for the new construct is as follows:

$$\mathit{Final}(\delta_B \,\&_{A_u}\, \delta_S, s) \equiv \mathit{Final}(\delta_B, s) \wedge \mathit{Final}(\delta_S, s).$$

This new construct captures exactly the maximally permissive supervisor; indeed the theorem below shows the *correctness of maximally permissive supervised execution*:

THEOREM 7.

$$\mathcal{CR}(\delta_B \,\&_{A_u}\, \delta_S, s) = \mathcal{CR}(\delta_B \,\&\, mps(\delta_B, \delta_S, s), s).$$

PROOF. We start by showing: $\mathcal{CR}(\delta_B \,\&_{A_u}\, \delta_S, s) \subseteq \mathcal{CR}(\delta_B \,\&\, mps(\delta_B, \delta_S, s), s)$. It suffices to show that $\delta_B \,\&_{A_u}\, \delta_S$ is controllable for δ_B in s. Indeed, if this is the case, by considering that $\delta_B \,\&\, mps(\delta_B, \delta_S, s)$ is the largest controllable supervisor for δ_B in s, and that $\mathcal{RR}(\delta_B \,\&\, (\delta_B \,\&_{A_u}\, \delta_S), s) = \mathcal{RR}(\delta_B \,\&_{A_u}\, \delta_S, s)$, we get the thesis.

So we have to show that: $\forall \vec{a}a_u.\vec{a} \in \mathcal{GR}(\delta_B \,\&_{A_u}\, \delta_S, s)$ and $A_u(a_u, do(\vec{a}, s))$ we have that if $\vec{a}a_u \in \mathcal{GR}(\delta_B, s)$ then $\vec{a}a_u \in \mathcal{GR}(\delta_B \,\&_{A_u}\, \delta_S, s)$.

Since, wlog we assume that δ_B and δ_S started with a common controllable action, we can write $\vec{a} = \vec{a'}a_c\vec{a_u}$, where $\neg A_u(a_c, do(\vec{a'}, s))$ and $A_u(\vec{a_u}, do(\vec{a'}a_c, s))$ holds. Let $\delta'_B = next^*(\delta_B, \vec{a'}, s)$, $\delta'_S = next^*(\delta_S, \vec{a'}, s)$, and $s' = do(\vec{a'}, s)$. By the fact that $\vec{a'}a_c\vec{a_u} \in \mathcal{GR}(\delta_B \,\&_{A_u}\, \delta_S, s)$ we know that $next(\delta'_B \,\&_{A_u}\, \delta'_S, do(a_c, s')) \neq \bot$. But then, by de definition of $next$, we have that for all $\vec{b_u}$ such that $A_u(\vec{b_u}, s')$ if $\vec{b_u} \in \mathcal{GR}(\delta'_B, do(a_c, s'))$ then $\vec{b_u} \in \mathcal{GR}(\delta'_S, do(a_c, s'))$. In particular this holds for $\vec{b_u} = \vec{a_u}a_u$. Hence we have that if $\vec{a}a_u \in \mathcal{GR}(\delta_B, s)$ then $\vec{a}a_u \in \mathcal{GR}(\delta_S, s)$.

Next we prove: $\mathcal{CR}(\delta_B \,\&\, mps(\delta_B, \delta_S, s), s) \subseteq \mathcal{CR}(\delta_B \,\&_{A_u}\, \delta_S, s)$. Suppose not. Then there exist a complete run $\vec{a}$ such that $\vec{a} \in \mathcal{CR}(\delta_B \,\&\, mps(\delta_B, \delta_S, s), s)$ but $\vec{a} \notin \mathcal{CR}(\delta_B \,\&_{A_u}\, \delta_S, s)$.

As an aside, notice that $\vec{a} \in \mathcal{CR}(\delta, s)$ then $\vec{a} \in \mathcal{GR}(\delta, s)$ and for all prefixes $\vec{a'}$ such that $\vec{a'}\vec{b} = \vec{a}$ we have $\vec{a'} \in \mathcal{GR}(\delta, s)$.

Hence, let $\vec{a'} = \vec{a''}a$ such that $\vec{a'} \in \mathcal{GR}(\delta_B \,\&_{A_u}\, \delta_S, s)$ but $\vec{a''}a \notin \mathcal{GR}(\delta_B \,\&_{A_u}\, \delta_S, s)$, and let $\delta''_B = next^*(\delta''_B, \vec{a''}, s)$, $\delta''_S = next^*(\delta_S, \vec{a''}, s)$, and $s' = do(\vec{a''}, s)$.

Since $\vec{a''}a \notin \mathcal{GR}(\delta_B \,\&_{A_u}\, \delta_S, s)$, it must be the case that $next(\delta''_B \,\&_{A_u}\, \delta''_S, a, s'') = \bot$. But then, considering that both

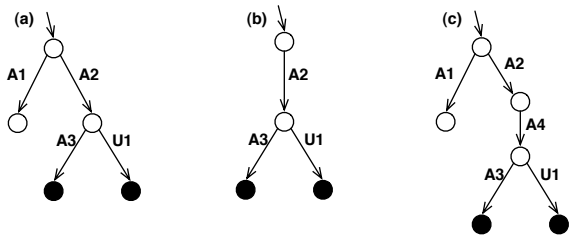

Figure 2: Diagrams of agent behavior specifications δ_{B1} in (a), δ_{B2} in (b), and δ_{B3} in (c).

$next(\delta''_B, a, s'') \neq \bot$ and $next(\delta''_S, a, s'') \neq \bot$, it must be the case that $\neg A_u(a, s'')$ and exists $\vec{b_u}$ such that $A_u(\vec{b_u}, do(a, s''))$, and $a\vec{b_u} \in \mathcal{GR}(\delta''_B, s'')$ but $a\vec{b_u} \notin \mathcal{GR}(\delta''_S, s'')$.

Notice that $\vec{b_u} \neq \epsilon$, since we have that $a \in \mathcal{GR}(\delta''_S, s'')$. So $\vec{b_u} = \vec{c_u}b_u\vec{d_u}$ with $a\vec{c_u} \in \mathcal{GR}(\delta''_S, s'')$ but $a\vec{c_u}b_u \notin \mathcal{GR}(\delta''_S, s'')$.

Now $\vec{a'} \in \mathcal{GR}(\delta_B \,\&\, mps(\delta_B, \delta_S, s), s)$ and since $A_u(\vec{c_u}b_u, do(\vec{a'}, s))$, we have that $\vec{a'}\vec{c_u}b_u \in \mathcal{GR}(\delta_B \,\&\, mps(\delta_B, \delta_S, s), s)$. Since, $mps(\delta_B, \delta_S, s)$ is controllable for δ_B in s, we have that, if $\vec{a'}\vec{c_u}b_u \in \mathcal{GR}(\delta_B, s)$ then $\vec{a'}\vec{c_u}b_u \in \mathcal{GR}(mps(\delta_B, \delta_S, s), s)$. This, by definition of $mps(\delta_B, \delta_S, s)$, implies $\vec{a'}\vec{c_u}b_u \in \mathcal{GR}(\delta_B \,\&\, \delta_S, s)$, and hence, in turn, $\vec{a'}\vec{c_u}b_u \in \mathcal{GR}(\delta_S, s)$. Hence, we can conclude that $a\vec{c_u}b_u \in \mathcal{GR}(\delta''_S, s'')$, getting a contradiction. □

Examples. Let us illustrate what is involved in obtaining a maximally permissive supervisor in the presence of uncontrollable actions. Suppose that we are in situation S_1 with an agent that has the following behavior (see Figure 2a):

$$\delta_{B1} = A_1 \mid (A_2; (A_3 \mid U_1)),$$

where action U_1 is uncontrollable (i.e. $A_u(a, s) \equiv a = U_1$); we also assume that all actions are always executable. Suppose as well that we have the following supervision specification:

$$\delta_S = (\pi a.(a \neq A_1 \wedge a \neq A_3 \wedge a \neq U_1)?; a)^*; (A_1 \mid A_3),$$

i.e., eventually A_1 or A_3 should be performed, and U_1 should never occur. If we let the agent perform action A_2, we get to situation $do(A_2, S_1)$ with the remaining agent behavior $(A_3 \mid U_1)$ and desired behavior δ_S, where we clearly can no longer effectively control the agent to ensure that it continues to behave as desired. Thus in situation S_1, we have to force the agent to perform A_1. We can in fact do this since A_1 and A_2 are controllable. The maximally permissive controllable supervisor $mps(\delta_{B1}, \delta_S, S_1)$ for this agent in S_1 is simply $\mathbf{set}(\{A_1\})$ (since $\mathcal{CR}(\delta, s) = \{A_1, (A_2; A_3)\}$ and $\mathbf{set}(\{(A_2; A_3)\})$ is not controllable wrt δ_{B1} in S_1).

If instead the agent's behavior in S_1 had been (see Figure 2b):

$$\delta_{B2} = A_2; (A_3 \mid U_1),$$

there would have been no way to ensure that the agent behaved as desired other than ruling out all actions, because even if we can control the agent in S_1, we are not be able to ensure that it continues to behave as desired once it gets to $do(A_2, S_1)$. Indeed $mps(\delta_{B2}, \delta_S, S_1) = \mathbf{set}(\emptyset)$.

The point of the example is that the supervisor must look ahead and always steer the agent away from paths where it cannot be prevented from eventually doing undesirable actions. Our definitions of mps and $\&_{A_u}$ ensure this. Indeed, for our example, we have that $next((\delta_{B1} \,\&_{A_u}$

$\delta_S), A_2, S_1) = \perp$ since $next^*(\Sigma(\delta_{B1}), [A_2, U_1], S_1) \neq \perp$ and $next^*(\Sigma(\delta_S), [A_2, U_1], S_1) = \perp$. Thus

$$next((\delta_{B1} \ \&_{A_u} \ \delta_S), a, S_1) \neq \perp \equiv a = A_1.$$

Moreover, $next((\delta_{B1} \ \&_{A_u} \ \delta_S), A_1, S_1) = (nil \ \&_{A_u} \ nil)$ and $Final((nil \ \&_{A_u} \ nil), do(A_1, S_1))$. Thus the only way execute $(\delta_{B1} \ \&_{A_u} \ \delta_S)$ in S_1 is to perform A_1, after which one terminates successfully. For agent behavior δ_{B2} on the other hand, we have

$$\forall a.next((\delta_{B1} \ \&_{A_u} \ \delta_S), a, S_1) = \perp,$$

i.e., all we can do is block.

Note also that in general, one must do lookahead search over the program to find complete executions using $\&_{A_u}$. Consider the following variant of the above example (see Figure 2c):

$$\delta_{B3} = A_1 \mid (A_2; A_4; (A_3 \mid U_1)).$$

In this case, $next((\delta_{B3} \ \&_{A_u} \ \delta_S), A_2, S_1) = ((A_4; (A_3 \mid U_1) \ \&_{A_u} \ \delta_S)$; the resulting program is not final in $do(A_2, S_1)$, yet $next(((A_4; (A_3 \mid U_1) \ \&_{A_u} \ \delta_S), a, do(A_2, S_1)) = \perp$ for all a. However if we do looakead search, we get that

$$next(\Sigma(\delta_{B3} \ \&_{A_u} \ \delta_S), a, S_1) \neq \perp \equiv a = A_1,$$

as well as $next(\Sigma(\delta_{B3} \ \&_{A_u} \ \delta_S), A_1, S_1) = \Sigma(nil \ \&_{A_u} \ nil)$ and $Final(\Sigma(nil \ \&_{A_u} \ nil), do(A_1, S_1))$.

5. CONCLUSION

We have investigated agent supervision in Situation-Determined ConGolog, or SDConGolog, programs. Our account of maximally permissive supervisor builds on the well-established Wonham and Ramadge framework for supervisory control of discrete event systems. However, virtually all work on this framework deals with finite state automata [2, 18], while we handle infinite state systems in the context of the rich agent setting provided by the situation calculus and ConGolog. We used ConGolog as a representative of an unbounded-states process specification language, and it should be possible to adapt our account of supervision to other related languages. We considered a form of supervision that focuses on complete runs, i.e., runs that lead to *Final* configurations. We can ensure that an agent finds such executions by having it do lookahead/search. Also of interest is the case in which agents act boldly without necessarily performing search to get to *Final* configurations. In this case, we need to consider all partial runs, not just good ones. Note that this would actually yield the same result if we engineered the agent behavior such that all of its runs are good runs, i.e. if $\mathcal{RR}(\delta_B, s) = \mathcal{GR}(\delta_B, s)$, i.e., all configurations are final. In fact, one could define a closure construct $cl(\delta)$ that would make all configurations of δ final. Using this, one can apply our specification of the maximally permissive supervisor to this case as well if we replace δ_B & δ_S by $cl(\delta_B$ & $\delta_S)$ in the definition. Observe also, that under the assumption that $\mathcal{RR}(\delta_B, s) = \mathcal{GR}(\delta_B, s)$, in $next(\delta_B \ \&_{A_u} \ \delta_S, a, s)$ we no longer need to do the search $\Sigma(\delta_B)$ and $\Sigma(\delta_S)$ and can directly use δ_B and δ_S.

We conclude by mentioning that if the object domain is finite, then ConGolog programs assume only a finite number of possible configurations. In this case, we can take advantage of the finite state machinery developed for discrete event systems on the basis of [19] (generalizing it to deal with situation-dependent sets of controllable actions), and the recent work on translating ConGolog into finite state machines and back [7], to obtain a program that actually characterizes the maximally permissive supervisor. In this way, we can completely avoid doing search during execution. We leave an exploration of this notable case for future work.

Acknowledgments. We thank Murray Wonham for inspiring discussions on supremal controllable languages in finite state discrete event control, which actually made us look into agent supervision from a different and very fruitful point of view. We also thank the anonymous referees for their comments. We acknowledge the support of EU Project FP7-ICT ACSI (257593).

6. REFERENCES

[1] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4):316–361, 2010.

[2] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008.

[3] G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[4] G. De Giacomo, Y. Lespérance, and A. R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In *KR*, 2010.

[5] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *KR*, pages 453–465, 1998.

[6] R. Demolombe and E. Hamon. What does it mean that an agent is performing a typical procedure? a formal definition in the situation calculus. In *AAMAS*, pages 905–911, 2002.

[7] C. Fritz, J. A. Baier, and S. A. McIlraith. ConGolog, Sin Trans: Compiling ConGolog into basic action theories for planning and beyond. In *KR*, pages 600–610, 2008.

[8] C. Fritz and Y. Gil. Towards the integration of programming by demonstration and programming by instruction using Golog. In *PAIR*, 2010.

[9] C. Fritz and S. McIlraith. Decision-theoretic Golog with qualitative preferences. In *KR*, pages 153–163, 2006.

[10] A. Goultiaeva and Y. Lespérance. Incremental plan recognition in an agent programming framework. In *PAIR*, 2007.

[11] N. Lin, U. Kuter, and E. Sirin. Web service composition with user preferences. In *ESWC*, pages 629–643, 2008.

[12] S. McIlraith and T. Son. Adapting Golog for composition of semantic web services. In *KR*, pages 482–493, 2002.

[13] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. In A. Bensoussan and J. Lions, editors, *Analysis and Optimization of Systems*, volume 63 of *Lecture Notes in Control and Information Sciences*, pages 475–498. Springer Berlin / Heidelberg, 1984.

[14] R. Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[15] S. Sardiña and G. De Giacomo. Composition of ConGolog programs. In *IJCAI*, pages 904–910, 2009.

[16] S. Sohrabi, N. Prokoshyna, and S. A. McIlraith. Web service composition via the customization of Golog programs with user preferences. In *Conceptual Modeling: Foundations and Applications*, pages 319–334. Springer, 2009.

[17] J. Su. Special issue on semantic web services: Composition and analysis. *IEEE Data Eng. Bull.*, 31(3), 2008.

[18] W. Wonham. *Supervisory Control of Discrete-Event Systems*. University of Toronto, 2011 edition, 2011.

[19] W. Wonham and P. Ramadge. On the supremal controllable sub-language of a given language. *SIAM J Contr Optim*, 25(3):637–659, 1987.