Agent Supervision in Situation-Determined ConGolog

Giuseppe De Giacomo
Sapienza – Università di Roma
Rome, Italy
degiacomo@dis.uniroma1.it

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

Christian Muise
University of Toronto
Toronto, Canada
cjmuise@cstoronto.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. The main results 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.

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 [Fritz and McIlraith, 2006; Lin *et al.*, 2008; Sohrabi *et al.*, 2009]. This approach was originally proposed in [McIlraith and Son, 2002] in the context of web service composition [Su, 2008]. The idea is that the generic process provides a wide range of alternative ways to perform the 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. Another 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 [Sardiña and De Giacomo, 2009; Bertoli *et al.*, 2010].

In this paper, we develop a framework for a similar type of process refinement that we call *supervised execution*. 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 performing some undesirable actions later on (e.g. getting muddy).

We follow previous work [McIlraith and Son, 2002; Fritz and McIlraith, 2006] in assuming that processes are specified in a high level agent programming language defined in the Situation Calculus [Reiter, 2001].[1] In fact, we define and use a restricted version of the ConGolog agent programming language [De Giacomo *et al.*, 2000] that we call Situation-Determined ConGolog (SDConGolog). In this version, following [De Giacomo *et al.*, 2010] 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. there is a unique remaining program δ' such that a given program δ can perform a transition $(\delta, s) \rightarrow_a (\delta', do(a, s))$ involving action a in situation s. 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 are functionally determined by the starting program and situation and the actions performed. Thus we can see a program and a starting situation as specifying a language, that of 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, intersection, and difference/complementation in terms of op-

[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 [Fritz and McIlraith, 2006], which handles an extended version of LTL interpreted over a finite horizon.

erations on the corresponding programs, which has applications in many areas (e.g. programming by demonstration and programming by instruction [Fritz and Gil, 2010], and plan recognition [Demolombe and Hamon, 2002]).Working with situation-determined programs also greatly facilitates the formalization of supervision/customization. In [De Giacomo *et al.*, 2010], it is in fact shown that any ConGolog program can be made situation-determined by recording nondeterministic choices made in the situation.

Besides a detailed characterization of SDConGolog,[2] the main contributions of the paper are as follows: first, based on previous work in discrete event control [Wonham and Ramadge, 1987], we provide a characterization of the *maximally permissive supervisor* that minimally constrains the actions of the agent so as to enforce the desired behavioral specifications, showing its existence and uniqueness; secondly, we define a 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 maximal permissive supervisor and supervised execution. Finally in Section 5, we review our contributions and discuss related and future work.

2 Preliminaries

The *situation calculus* is a logical language specifically designed for representing and reasoning about dynamically changing worlds [Reiter, 2001]. All changes to the world are the result of *actions*, which are terms in the language. 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 [Reiter, 2001].

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 $\delta_1 \| \delta_2$ 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 [De Giacomo *et al.*, 2000]: *(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 [De Giacomo *et al.*, 2010]; these are in fact the usual ones [De Giacomo *et al.*, 2000], 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). A consequence of this is that in the version of ConGolog that we use, every transition involves the execution 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 [Reiter, 2001] and $\mathcal{C}$ to denote the axioms defining the ConGolog language.

3 Situation-Determined Programs

As mentioned earlier, we are interested in process customization. For technical reasons, we will focus on a restricted class of ConGolog programs for describing processes, namely "situation-determined programs". 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.,

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

where *Trans** denotes the reflexive transitive closure of *Trans*. Thus, a (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 situation-determined programs can be characterized in terms of sequences of actions only*, those sequences that correspond to the situations reached from where the program started.

[2] In [De Giacomo *et al.*, 2010], situation-determined programs were only dealt with incidentally.

For example, 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:

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

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:

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

Moreover 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 *Properties 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 can extend the SDConGolog language so as to close it with respect to language theoretic operations, such as *union*, *intersection* and *difference/complementation*. 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.

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:

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

Sequences of actions generated by programs. We can extend the function $next$ to the function $next^*(\delta, \vec{a}, s)$ that takes a program δ, a finite sequence of actions $\vec{a}$,[3] 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:

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

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

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:[4]

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

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

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:

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

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:

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

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.

The search construct. We can add to the language a search construct Σ, as in [De Giacomo *et al.*, 1998]:

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

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

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 $\bot$.

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.

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.

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

[3] Notice that such sequences of actions have to be axiomatized in second-order logic, similarly to situations (with UNA and domain closure). As a short cut they could also be characterized directly in terms of "difference" between situations.

[4] 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.

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.

Following [Wonham and Ramadge, 1987], we distinguish between actions that are *uncontrollable* by the supervisor and actions that are *controllable*. 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, differently from the Wonham and Ramadge work, 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 supervisor is "effective". Technically, following again Wonham and Ramadge's ideas, this can be captured by saying that the *supervision by δ_S is effective for δ_B in situation s* iff:

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

What this says is that if we postfix a good run $\vec{a}$ for both B and S with an uncontrollable action a_u that is good for B, 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)* relax δ_S so that it places no constraints on the uncontrollable actions; *(ii)* require that δ_S be indeed enforced, but disallow all those runs that prevent δ_S from being effective. 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 behavior 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 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 effective for δ_B in situation s.*

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

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 effective for 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 not always acceptable. The last one, says that when the converse of Proposition 4 holds, we have that the original supervision δ_S is indeed effective for δ_B in situation s. Notice however that even if δ_S effective for δ_B in situation s, it may still be the case that $\mathcal{CR}(\delta_B \ \& \ rl(\delta_S, A_u), s) \subset \mathcal{CR}(\delta_B \ \& \ \delta_S, s)$.

Maximal permissive supervisor. Next we study a more conservative approach: we require the supervision δ_S to be fulfilled, and for getting effectiveness we restrict it further. Interestingly, we show that there is a single maximal way of restricting the supervisor S so that it both fulfills δ_S and becomes effective. We call the resulting supervisor the *maximal permissive supervisor*.

We start by introducing a new abstract program construct $\mathbf{set}(E)$ taking as argument a possibly infinite set E of sequences of actions, with $next$ and *Final* 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 abstract 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 ConGolog may not allow us to represent some possible sets of sequences of actions.

With the $\mathbf{set}(E)$ construct at hand, following [Wonham and Ramadge, 1987], we may define the *maximal permissive supervisor* $mps(\delta_B, \delta_S, s)$ of B with behavior δ_B by S with behavior δ_S in situation s, as:

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

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 δ_B on the non-controllable 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 maximal 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 an effective supervisor for δ_B in s;*
3. *For every possible effective supervisor $\hat{\delta}_S$ 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 follows directly from the fact $\mathbf{set}(\emptyset)$ satisfies the conditions to be included in $mps(\delta_B, \delta_S, s)$.

Claim 3 also follows immediately from the definition of $mps(\delta_B, \delta_S, s)$, by recalling that $\mathcal{CR}(\delta_B \,\&\, \hat{\delta}_S, s) = \mathcal{CR}(\delta_B \,\&\, \mathbf{set}(E_{\hat{\delta}_S}), s)$.

For *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 effective supervisor $\mathbf{set}(E)$ such that $\vec{a} \in \mathcal{GR}(\delta_B \,\&\, \mathbf{set}(E), \delta_S, s), s)$. $\mathbf{set}(E)$ being effective for 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 $\vec{a}a_u \in \mathcal{GR}(mps(\delta_B, \delta_S, s), s)$. □

We can illustrate using our example programs. 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 maximal permissive supervisor $mps(\delta_{B1}, \delta_{S1}, S_0)$, whose set of partial/complete 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 as δ_{B1}.

Unfortunately, in general, $mps(\delta_B, \delta_S, s)$ requires the use of the abstract program construct $\mathbf{set}(E)$, which can be expressed directly in ConGolog only if E is finite.[5] For this reason the above characterization remains essentially mathematical. So next, we develop a new construct for execution of programs under maximal permissive supervision, which is indeed realizable.

Maximal permissive supervised execution. To capture the notion of maximal 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:

$$next(\delta_B \,\&_{A_u}\, \delta_S, a, s) =$$
$$\begin{cases} \bot \text{ if } \neg A_u(a, s) \text{ and } \exists \vec{a_u}. A_u(\vec{a_u}, do(a, s)) \text{ s.t.} \\ next^*(\Sigma(\delta_B), a\vec{a_u}, s) \neq \bot \text{ and } next^*(\Sigma(\delta_S), a\vec{a_u}, s) = \bot \\ \bot \text{ if } next(\delta_B, a, s) = \bot \text{ or } next(\delta_S, a, s) = \bot \\ next(\delta_B, a, s) \,\&_{A_u}\, next(\delta_S, a, s) \qquad \text{otherwise} \end{cases}$$

Here $A_u(\vec{a_u}, s)$ 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)* $A_u(a_u\vec{a_u}, s) \equiv A_u(a_u, s) \wedge A_u(\vec{a_u}, do(a_u, s))$.

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 maximal permissive supervisor; indeed the theorem below shows the *correctness of maximal 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 effective 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 effective 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.

[5] Note that the object domain may be uncountable in general, hence not even an infinitary ConGolog program could capture $\mathbf{set}(E)$ in general.

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 $next(\delta''_B, a, s'') \neq \bot$ 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 effective 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. □

5 Conclusion

In this paper, we have investigated agent supervision in situation-determined ConGolog programs. Our account of maximal permissive supervisor builds on [Wonham and Ramadge, 1987]. However, Wonham and Ramage's work deals with finite state automata, while we handle infinite state systems in the context of the rich agent framework 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 maximal permissive supervisor to this case as well if we replace $\delta_B \ \& \ \delta_S$ by $cl(\delta_B \ \& \ \delta_S)$ in the definition. Observe also, that under the assumption $\mathcal{RR}(\delta_B, s) = \mathcal{GR}(\delta_B, s)$, 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 that was originally proposed by Wonham and Ramage (generalizing it to deal with situation-dependent sets of controllable actions), and the recent work on translating ConGolog into finite state machines and back [Fritz *et al.*, 2008], 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).

References

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

[De Giacomo *et al.*, 1998] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In *KR*, pages 453–465, 1998.

[De Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[De Giacomo *et al.*, 2010] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In *KR*, 2010.

[Demolombe and Hamon, 2002] Robert Demolombe and Erwan 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.

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

[Fritz and McIlraith, 2006] Christian Fritz and Sheila McIlraith. Decision-theoretic Golog with qualitative preferences. In *KR*, pages 153–163, June 2–5 2006.

[Fritz *et al.*, 2008] Christian Fritz, Jorge A. Baier, and Sheila A. McIlraith. ConGolog, sin trans: Compiling ConGolog into basic action theories for planning and beyond. In *KR*, pages 600–610, 2008.

[Lin *et al.*, 2008] Naiwen Lin, Ugur Kuter, and Evren Sirin. Web service composition with user preferences. In *ESWC*, pages 629–643, 2008.

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

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

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

[Sohrabi *et al.*, 2009] Shirin Sohrabi, Nataliya Prokoshyna, and Sheila 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.

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

[Wonham and Ramadge, 1987] WM Wonham and PJ Ramadge. On the supremal controllable sub-language of a given language. *SIAM J Contr Optim*, 25(3):637659, 1987.