Tools for Representing and Reasoning about Biological Models in Action Language $\mathcal{C}$

Steve Dworschak and **Torsten Grote** and **Arne König** and **Torsten Schaub** and **Philippe Veber**
Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract

We elaborate upon the usage of action language $\mathcal{C}$ for representing and reasoning about biological models. First, we provide a simple extension of $\mathcal{C}$ allowing for variables and show its usefulness in modeling biochemical reactions according to the well-known model of BIOCHAM. Second, we show how the biological action description language $\mathcal{C}_{TAID}$ can be mapped onto $\mathcal{C}$. Finally, we describe a toolbox for using action languages, including among them, a compiler mapping $\mathcal{C}$ and $\mathcal{C}_{TAID}$ to logic programs under answer sets semantics along with a web-service integrating different front- and back-ends for addressing dynamical systems by means of action description languages via answer set programming. This is accompanied by an empirical evaluation with existing systems for processing action description languages.

Introduction

We elaborate upon *action languages* (Gelfond and Lifschitz 1998) for qualitative modeling of biological networks. Action languages are formal models used for reasoning about the effects of actions, while being close to natural language. Central to this approach to formalizing actions is the concept of a transition system, which constitutes its semantic underpinning. The first action language for representing and reasoning about biological networks was introduced in (Tran and Baral 2004; Baral et al. 2004; Tran 2006) and further extended in (Dworschak et al. 2008) leading to action language $\mathcal{C}_{TAID}$.

In what follows, we extend the overall approach in several ways while centering it on the classical action language $\mathcal{C}$ (Giunchiglia and Lifschitz 1998). To begin with, we provide a simple extension of $\mathcal{C}$ allowing for variables and show its usefulness in modeling biochemical reactions according to the well-known model of BIOCHAM. Similar to the approach taken in the dlv^k system based on action language $\mathcal{K}$ (Eiter et al. 2003a), we delegate the treatment of variables to Answer Set Programming (ASP; (Baral 2003)) in order to be able to use ASP grounders for variable instantiation. Second, we provide a translation mapping the biologically motivated action description language $\mathcal{C}_{TAID}$ onto $\mathcal{C}$ and give a result fixing the formal correspondence. This allows us to further develop $\mathcal{C}_{TAID}$ within a broader and well-established framework, avoiding further dedicated implementations. Moreover, it provides $\mathcal{C}_{TAID}$ with access to further implementations of $\mathcal{C}$, like $CCalc$ (Giunchiglia et al. 2004) or $CPlan$ (Castellini, Giunchiglia, and Tacchella 2003) (even though they cannot harness existing ASP grounders for variable treatment). Finally, we describe a toolbox for using action languages, including among them, a (pipe-based) compiler mapping $\mathcal{C}$ and $\mathcal{C}_{TAID}$ to logic programs under answer sets semantics along with a web-service integrating different front- and back-ends for addressing dynamical systems by means of action description languages via answer set programming. Our tools are designed for an easy and flexible integration with existing open source tools via pipes, in particular, ASP grounders and solvers, as well as further front- and back-ends. This is accompanied by an empirical evaluation with existing systems for processing action description languages.

Background

Answer Set Programming. Our language is built from a set $\mathcal{F}$ of *function* symbols (including the natural numbers), a set $\mathcal{V}$ of *variable* symbols, and a set $\mathcal{P}$ of *predicate* symbols. The set $\mathcal{T}$ of *terms* is the smallest set containing $\mathcal{V}$ and all expressions of the form $f(t_1,\dots,t_n)$, where $f \in \mathcal{F}$ and $t_i \in \mathcal{T}$ for $1 \leq i \leq n$. The set $\mathcal{A}$ of *atoms* contains expressions of the form $p(t_1,\dots,t_n)$, where $p \in \mathcal{P}$ and $t_i \in \mathcal{T}$ for $1 \leq i \leq n$. A *literal* is an atom a or its negation $\neg a$; both can be preceded by default negation, denoted as $not\ a$ and $not\ \neg a$, respectively. For $a \in \mathcal{A}$, we let $\overline{a} = \neg a$ and $\overline{\neg a} = a$. A *logic program* over $\mathcal{A}$ is a set of *rules* of the form

$$a \leftarrow b_1,\dots,b_m, not\ c_{m+1},\dots, not\ c_n \tag{1}$$

where a, b_i, c_j are literals over $\mathcal{A}$ for $0 < i \leq m < j \leq n$. For a rule r as in (1), let $head(r) = a$, $body(r)^+ = \{b_1,\dots,b_m\}$, and $body(r)^- = \{c_{m+1},\dots,c_n\}$. Given an expression $e \in \mathcal{T} \cup \mathcal{A}$, let $var(e)$ denote the set of all variables occurring in e; analogously, $var(r)$ gives all variables in rule r. The *ground instantiation* of a program P is defined as $grd(P) = \{r\theta \mid r \in P, \theta : var(r) \rightarrow \mathcal{U}\}$, where $\mathcal{U} = \{t \in \mathcal{T} \mid var(t) = \emptyset\}$; analogously, $grd(\mathcal{A}) = \{a \in \mathcal{A} \mid var(a) = \emptyset\}$ is the set of all ground atoms. A set $X \subseteq grd(\mathcal{A}) \cup \overline{grd(\mathcal{A})}$ is a (consistent) *answer set* of a program P over $\mathcal{A}$, if X is the $\subseteq$-smallest model of

$$\{head(r) \leftarrow body(r)^+ \mid r \in grd(P), body(r)^- \cap X = \emptyset\}\ .$$

Action Language $\mathcal{C}$. Action languages use *fluents* to describe the states of a system and *actions* influence the values of fluents. In $\mathcal{C}$ (and $\mathcal{C}_{TAID}$), *static laws* describe properties between fluents that need to be satisfied in every state of the system. *Dynamic laws* describe the effects of actions, that is, how the system evolves when actions are executed.

More formally, we consider *action language $\mathcal{C}$* (Gelfond and Lifschitz 1998) over a Boolean *action signature* $\langle B, F, A\rangle$, where B is the set $\{f, t\}$ of truth values, F is a set of *fluent names*, and A is a set of *action names*. In $\mathcal{C}$, an *action description* $D_{\mathcal{C}}$ over a signature $\langle B, F, A\rangle$ consists of *static laws*, such as

$$(\mathbf{caused}\ \varphi\ \mathbf{if}\ \psi) \tag{2}$$

and *dynamic laws* of the form

$$(\mathbf{caused}\ \varphi\ \mathbf{if}\ \psi\ \mathbf{after}\ \omega), \tag{3}$$

where φ and ψ are propositional combinations of fluent names and ω is a propositional combination of fluent and action names. Every action description $D_{\mathcal{C}}$ induces a unique transition system $\mathcal{T}_{\mathcal{C}}(D_{\mathcal{C}}) = \langle S, V, R\rangle$, where S is a set of *states*, V is a function determining fluents values in state s, and R is a relation containing all possible transitions between states. A *trajectory* $s_0, A_1, s_1, \ldots, s_{n-1}, A_n, s_n$ in a transition system $\langle S, V, R\rangle$ is a sequence of sets of actions $A_i \subseteq A$ and states $s_i \in S$ where $(s_{i-1}, A_i, s_i) \in R$ for $0 \leq i \leq n$. Intuitively, a trajectory represents one possible history (or simply path) within a transition system. In (Gelfond and Lifschitz 1998), several syntactic extensions are defined. For instance, the rule (ω **may cause** φ **if** ψ) is a shorthand for (**caused** φ **if** φ **after** $\psi \wedge \omega$). Similarly, (**inertial** φ) is a shorthand for (**caused** φ **if** φ **after** φ). We refer to (Gelfond and Lifschitz 1998) for more detailed definitions.

Besides an action description language, both $\mathcal{C}$ and $\mathcal{C}_{TAID}$ define a *query language*. We implemented $\mathcal{R}$ (Gelfond and Lifschitz 1998) as the query language for $\mathcal{C}$ and the query mechanisms described in (Dworschak et al. 2008) for $\mathcal{C}_{TAID}$. In this paper, we focus only on the transition systems and our toolbox realizing the different encodings, so we omit a detailed description of query languages and the different reasoning modes. In the biological setting, queries combined with reasoning modes like planning and explanation are used to answer biological questions. For example, one is able to determine whether certain states can be reached in molecular networks, queries about existence of paths can be answered and it is possible to do high-level experiment planning.

Encoding Action Language $\mathcal{C}$

For implementing action language $\mathcal{C}$, we build upon the translation to ASP described in (Lifschitz and Turner 1999). Let $D_{\mathcal{C}}$ be an action description over signature $\langle B, F, A\rangle$. We require $D_{\mathcal{C}}$ to be *definite*, that is, the heads φ of laws (**caused** φ **if** ψ) and (**caused** φ **if** ψ **after** ω) are fluent literals (or the constant $\bot$). Furthermore, ψ is a conjunction of fluent literals and ω is a conjunction of fluent and/or action literals. In what follows, we denote φ by f, ψ by $g_1 \wedge \ldots \wedge g_m$ and ω by $l_1 \wedge \ldots \wedge l_n$.

We define a logic program $lp_n(D_{\mathcal{C}})$ whose answer sets correspond to trajectories of length n in the transition system induced by $D_{\mathcal{C}}$. $lp_n(D_{\mathcal{C}})$ contains atoms $a(t)$ and $f(t)$ for each $a \in A$, $f \in F$ and $t = 0, \ldots, n$. For each static law (**caused** f **if** $g_1 \wedge \ldots \wedge g_m$) in $D_{\mathcal{C}}$, $lp_n(D_{\mathcal{C}})$ contains for each $t = 0, \ldots, n$ a rule

$$f(t) \leftarrow \mathit{not}\ \overline{g_1(t)}, \ldots, \mathit{not}\ \overline{g_m(t)}\ .$$

Analogously, each dynamic law (**caused** f **if** $g_1 \wedge \ldots \wedge g_m$ **after** $l_{m+1} \wedge \ldots \wedge l_n$) in $D_{\mathcal{C}}$, adds to $lp_n(D_{\mathcal{C}})$ for each $t = 0, \ldots, n-1$ a rule

$$f(t{+}1) \leftarrow \mathit{not}\ \overline{g_1(t+1)}, \ldots, \mathit{not}\ \overline{g_m(t+1)}, l_{m+1}(t), \ldots, l_n(t)\ .$$

Furthermore, $lp_n(D_{\mathcal{C}})$ contains

$$\begin{array}{rcl@{\qquad}rcl} \neg a(t) & \leftarrow & \mathit{not}\ a(t), & \neg e(0) & \leftarrow & \mathit{not}\ e(0), \\ a(t) & \leftarrow & \mathit{not}\ \neg a(t), & e(0) & \leftarrow & \mathit{not}\ \neg e(0) \end{array}$$

for each $a \in A$, $t = 0, \ldots, n$ and each $e \in F$.

Our implementation of the encoding allows to use variables when writing rules in $\mathcal{C}$. This is done by delegating the grounding of variables to the grounding process of the underlying logic program. To this end, we start by extending the syntax of $\mathcal{C}$ by a trailing keyword **where** followed by domain predicates for binding the variables occurring in the actual causal laws. To be precise, the causal laws in (2) and (3) are extended as follows:

$$(\mathbf{caused}\ \varphi\ \mathbf{if}\ \psi\ \mathbf{where}\ \delta) \tag{4}$$

$$(\mathbf{caused}\ \varphi\ \mathbf{if}\ \psi\ \mathbf{after}\ \omega\ \mathbf{where}\ \delta) \tag{5}$$

where φ, ψ, and ω are as defined in (2) and (3), except for containing variables, and δ is a combination of non-fluent and non-action atoms such that $var(\varphi) \cup var(\psi) \cup var(\omega) \subseteq var(\delta)$. Intuitively, δ captures static domain information used for binding the variables in φ, ψ, ω. The concept of a definite action description generalizes in the obvious way, restricting δ to conjunctions of non-fluent and non-action atoms. Now, given such a definite action description $D_{\mathcal{C}}$, the variable-tolerating extension of $lp_n(D_{\mathcal{C}})$ is obtained from $lp_n(D_{\mathcal{C}})$ by extending the body of each resulting logic programming rule by $d_1, \ldots, d_o$ whenever the causal law contains the condition **where** $d_1 \wedge \cdots \wedge d_o$.

Let us illustrate the practical impact of this pragmatic extension by modeling the Biochemical Abstract Machine (BIOCHAM; (Fages, Sollman, and Chabrier-Rivier 2004; Chabrier-Rivier et al. 2004)), used to build biochemical systems. The biological background is indeed very easy. A modeled scenario consists of different chemical reactions that specify relations between different compounds. *Reactants* are compounds that need to be present that a reaction can take place and *products* are compounds that will be present after a reaction took place. One can model this scenario using $\mathcal{C}$ with the following rules. At first, our syntax requires to specify a preamble where actions and fluents are defined:

```
<action> occurs(R)  <where> reaction(R).
<fluent> present(P) <where> compound(P).
```

Strings enclosed in <> are keywords, variables start with uppercase letters and lines end with a dot. That is, for every term `t` belonging to the extension of the predicate `reaction`, we introduce the actions `occurs(t)`. For every term `t` belonging to the extension of the predicate `compound`, we introduce the fluents `present(t)`.

We now can define the dynamics of the system:

```
<caused> present(P)
  <after> occurs(R)
  <where> reaction(R), compound(P),
          product(P,R).

<caused> <false>
  <after> occurs(R),-present(P)
  <where> reaction(R),compound(P),
          reactant(P,R).

occurs(R) <may cause> -present(P)
  <where> reaction(R),compound(P),
          reactant(P,R).

<inertial> present(P) <where> compound(P).
<inertial> -present(P) <where> compound(P).
```

The first rule states that a compound `P` is present after a reaction `R` occurred producing `P`. The second rule is a constraint enforcing all compounds `P` to be present if a reaction occurs where `P` is a reactant of. Note that negation is denoted as `-` and `<false>` as well as `<true>` are keywords for the two Boolean constants. The third rule models a certain non-determinism: The semantics of BIOCHAM defines that after a reaction occurs, it remains unclear whether the reactants are still present or not. The reason is that the semantics abstract from concentrations of compounds. That is, we consider two cases: In one transition we assume that the compound `P` was fully consumed, modeled as `-present(P)`. The other transition is that `P` remains to be present. The last two rules state that compounds that are not affected by reactions do not change their value [1].

Let us briefly detail how variables are passed through the encoding proposed in (Lifschitz and Turner 1999). For this, consider the first rule of the BIOCHAM example:

```
<caused> present(P)
  <after> occurs(R)
  <where> reaction(R), compound(P),
          product(P,R).
```

It is translated to the following logic rule:

```
present_fluent(P,T+1)
:- occurs_action(R,T),
   reaction(R),compound(P),product(P,R),
   time(T).
```

Apart from the time-parameter `T`, we attach variable `P` to the fluent `present` and `R` to the action `occurs`. The domain information given in the `<where>` statement is then passed as grounding information to the logic program rule.

[1]These rules represents the frame axiom: Compounds that are not consumed remain present, absent compounds that where not produced remain absent.

The last pending issue is to specify the domains:

```
compound(a). compound(b).
reaction(r1).
reactant(a,r1). product(b,r1).
```

The database is represented as a logic program. It can be seen as static knowledge attached to the modeled dynamic behavior of the system. In most cases, the database only contains facts. In the example, we are now able to reason about a scenario with two compounds and one reaction. An encoding of a simple version of the biological textbook example of the *Mitogen-activated protein kinase (MAPK)*[2] including 23 products and 30 reactions, yields a problem instance containing 147 facts. One of the advantages using variables is that the system can be easily enhanced by extending the database, that is, without touching the specification of the dynamics.

Mapping $\mathcal{C}_{TAID}$ to $\mathcal{C}$

As with $\mathcal{C}$, an *action description* in $\mathcal{C}_{TAID}$ is given relative to an action signature $\langle B, F, A\rangle$. The major conceptual difference between $\mathcal{C}_{TAID}$ and $\mathcal{C}$ is that the latter implicitly treats actions to be exogenous. That is, all actions might occur at every time-point as long as their effects do not lead to a contradiction. For biological purposes, this behavior is inappropriate. Unlike this, $\mathcal{C}_{TAID}$ allows for specifying explicit conditions when actions are executed or not. For example, using $\mathcal{C}_{TAID}$'s *triggering rule*, we can describe properties when (re)actions must be executed immediately. Furthermore, $\mathcal{C}_{TAID}$ offers the following constructs: *Inhibition rules* express when actions must not be executed and *allowance rules* express that actions might occur, but are not forced to. A *default* expresses that a fluent takes a certain value unless it is known otherwise. *No-concurrency constraints* allow to control the parallel execution of actions. In a more formal way, an action description in $\mathcal{C}_{TAID}$ contains expressions of the following form:

$$\begin{array}{l}(a \textbf{ causes } \varphi \textbf{ if } \psi)\\ (\varphi \textbf{ if } \psi)\\ (\varphi \textbf{ triggers } a)\\ (\varphi \textbf{ allows } a)\\ (\varphi \textbf{ inhibits } a)\\ (\textbf{noconcurrency } \omega)\\ (\textbf{default } f),\end{array}$$

where a is an action and ω is either an action or a conjunction of action literals, φ and ψ are conjunctions of fluent literals and f is a fluent literal. We refer to (Dworschak et al. 2008) for a more detailed description of $\mathcal{C}_{TAID}$.

We now describe our translation of $\mathcal{C}_{TAID}$ into $\mathcal{C}$. To this end, we need to extend the action signature to accommodate some control information. To be precise, we add the fluents $ih(a)$, $tr(a)$, $ex(a)$, $al(a)$ for each action name a. Intuitively, these fluents signal properties reflecting the behavior of inhibition, triggering, and allowance rules. With

[2]`http://en.wikipedia.org/wiki/MAPK`

them, we can define the mapping of rules in $\mathcal{C}_{TAID}$ to rules in $\mathcal{C}$ as follows.

Definition 1 *Let $D_{\mathcal{C}_{TAID}}$ be an action description in $\mathcal{C}_{TAID}$ over action signature $\langle B, F, A\rangle$. The corresponding action description $D_{\mathcal{C}}$ in $\mathcal{C}$ over action signature*

$$\langle B, F \cup \bigcup_{a\in A}\{ih(a), tr(a), ex(a), al(a)\}, A\rangle \qquad (6)$$

is defined as follows:

1. *For each action name $a \in A$, action description $D_{\mathcal{C}}$ contains the static laws*

$$(\textbf{caused } \neg ih(a) \textbf{ if } \neg ih(a)),$$
$$(\textbf{caused } \neg tr(a) \textbf{ if } \neg tr(a)),$$
$$(\textbf{caused } \neg ex(a) \textbf{ if } \neg ex(a)), \text{and}$$
$$(\textbf{caused } \neg al(a) \textbf{ if } \neg al(a)).$$

2. *For each dynamic law $(a$ **causes** φ **if** $\psi)$ in $D_{\mathcal{C}_{TAID}}$, where $\varphi = f_1 \wedge \ldots \wedge f_m$, $D_{\mathcal{C}}$ contains the laws*

$$(\textbf{caused } f_i \textbf{ if } \top \textbf{ after } \psi \wedge a)$$

for each f_i where $1 \leq i \leq m$.

3. *For each static law $(\varphi$ **if** $\psi)$ in $D_{\mathcal{C}_{TAID}}$, where $\varphi = f_1 \wedge \ldots \wedge f_m$, $D_{\mathcal{C}}$ contains the laws*

$$(\textbf{caused } f_i \textbf{ if } \psi)$$

for each f_i where $1 \leq i \leq m$.

4. *For each allowance rule $(\varphi$ **allows** $a)$ in $D_{\mathcal{C}_{TAID}}$, $D_{\mathcal{C}}$ contains*

$$(\textbf{caused } al(a) \textbf{ if } \varphi) \text{ and}$$
$$(\textbf{caused } \bot \textbf{ if } \top \textbf{ after } \neg al(a) \wedge a).$$

5. *For each triggering rule $(\varphi$ **triggers** $a)$ in $D_{\mathcal{C}_{TAID}}$, $D_{\mathcal{C}}$ contains*

$$(\textbf{caused } tr(a) \textbf{ if } \varphi),$$
$$(\textbf{caused } ex(a) \textbf{ if } \top \textbf{ after } a),$$
$$(\textbf{caused } \bot \textbf{ if } \neg ex(a) \textbf{ after } tr(a) \wedge \neg ih(a)) \text{ and}$$
$$(\textbf{caused } \bot \textbf{ if } ex(a) \textbf{ after } \neg tr(a)).$$

6. *For each inhibition rule $(\varphi$ **inhibits** $a)$ in $D_{\mathcal{C}_{TAID}}$, $D_{\mathcal{C}}$ contains*

$$(\textbf{caused } ih(a) \textbf{ if } \varphi) \text{ and}$$
$$(\textbf{caused } \bot \textbf{ if } \top \textbf{ after } ih(a) \wedge a).$$

7. *For each constraint $($**noconcurrency** $\omega)$ in $D_{\mathcal{C}_{TAID}}$, $D_{\mathcal{C}}$ contains*

$$(\textbf{caused } \bot \textbf{ if } \top \textbf{ after } \omega).$$

8. *For each default rule $($**default** $f)$ in $D_{\mathcal{C}_{TAID}}$, $D_{\mathcal{C}}$ contains*

$$(\textbf{caused } f \textbf{ if } f).$$

9. *For each $f \in F$, such that $($**default** $f) \notin D_{\mathcal{C}_{TAID}}$, $D_{\mathcal{C}}$ contains*

$$(\textbf{caused } f \textbf{ if } f \textbf{ after } f) \text{ and}$$
$$(\textbf{caused } \neg f \textbf{ if } \neg f \textbf{ after } \neg f).$$

The symbols $\top$ and $\bot$ denote the Boolean constants for t and f in B.

The rules in 1. state that $ih(a)$, $tr(a)$, $ex(a)$, and $al(a)$ are set to be *false* by default. As described in 4.–6., they are only set *true* when certain properties hold. There is a direct correspondence between static and dynamic rules in $\mathcal{C}_{TAID}$ and $\mathcal{C}$ (cf. 2. and 3.) except the fact that conjunctions in heads are split in order to get a definite action description. An allowance rule is expressed using a static rule setting $al(a)$ and a dynamic rule that can be viewed as a constraint eliminating transitions where action a occurred while $al(a)$ was *false* (cf. 4.). Rules given in 5. express triggering rules: whenever a trigger is applicable, $tr(a)$ is set and every execution of an action a causes $ex(a)$ to be true. The second dynamic rule eliminates transitions where the conditions for a triggering rule were satisfied but a was not executed, that is, $ex(a)$ is *false*. Since $\mathcal{C}_{TAID}$ gives inhibition rules priority over triggering rules, the constraint is only applicable if $\neg ih(a)$ is satisfied. The third dynamic rule eliminates transitions where a is executed without having an applicable trigger. Inhibition rules are mapped in the same way as allowance rules (cf. 6.). No-concurrency constraints and defaults in $\mathcal{C}_{TAID}$ have a direct correspondence to rules in $\mathcal{C}$ (cf. 7. and 8.). Given that fluents are implicitly inertial[3] in $\mathcal{C}_{TAID}$ but not in $\mathcal{C}$, for each fluent there is a dynamic rule in $D_{\mathcal{C}}$ that expresses inertial behavior (cf. 9.).

We can show the following result:

Theorem 1 *Let $D_{\mathcal{C}_{TAID}}$ be an action description in $\mathcal{C}_{TAID}$ over action signature $\langle B, F, A\rangle$ and let $D_{\mathcal{C}}$ be the corresponding action description in $\mathcal{C}$ over the action signature in (6) generated from $D_{\mathcal{C}_{TAID}}$ using the mapping in Definition 1.*

Then, each trajectory in the transition system $\mathcal{T}_{\mathcal{C}}(D_{\mathcal{C}})$ (as defined in (Gelfond and Lifschitz 1998)), corresponds to a unique trajectory in the transition system induced by $D_{\mathcal{C}_{TAID}}$ (as defined in (Dworschak et al. 2008)) and vice versa.

Problem descriptions in $\mathcal{C}_{TAID}$ can now be dealt with the general-purpose language $\mathcal{C}$. That is, we do not need a rather complicated (re)definition of semantics in order to describe transition systems having a biological background using $\mathcal{C}_{TAID}$. It can now be seen as another layer of interface on top of the action description language $\mathcal{C}$.

In the following sections we describe how the different encodings can be used in our toolchain and how they perform compared to other implementations.

The `BioPlan` System

Our approach to representing and reasoning about biological models is as follows: at first, the biological model needs to be specified in the action description language of $\mathcal{C}$ or $\mathcal{C}_{TAID}$. This description is compiled into a logic program as described above and subsequently dealt with using an ASP system, usually composed of a grounder and a solver. ~~To a turn~~ ☐nce the logic program is solved, the answers of the ☐ DELETEO

[3]That is, fluents that are neither defaults nor affected directly or indirectly by dynamic rules do not change their value in a transition.

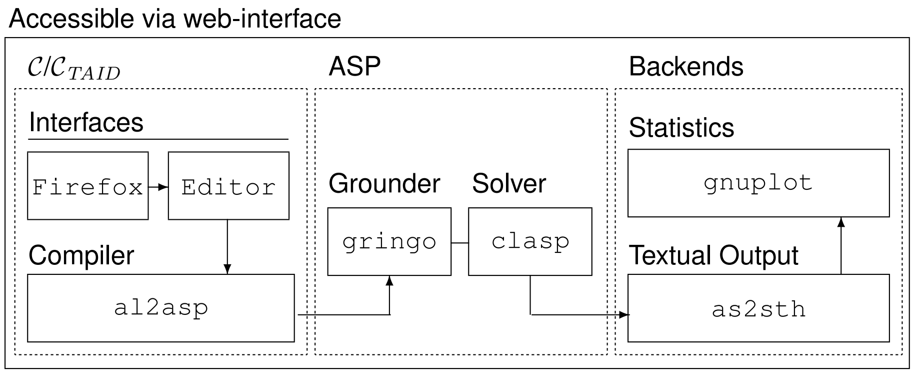


Figure 1: Overview of our system architecture

solver need to be put back in correspondence to the original problem specification. Finally, the obtained data needs to be interpreted in a biologically meaningful way by a human expert. An overview of our system is given in Figure 1.

Interfaces

To begin with, we have a closer look at the interfaces to our system. Our system is able to handle the discussed action descriptions in $\mathcal{C}$ and $\mathcal{C}_{TAID}$. For action descriptions in $\mathcal{C}$, one has to write down the rules in an editor, as shown in the BIOCHAM example. This is of course also possible using $\mathcal{C}_{TAID}$. Since $\mathcal{C}_{TAID}$ has a much more biological orientation than $\mathcal{C}$, we offer another interface for $\mathcal{C}_{TAID}$ that is more intuitive for users having a purely biological background: A graphical interface that was built as a Firefox browser extension. It allows for building rules as a graph whose nodes (fluents and actions) and edges (causal relationships) correspond to the underlying expressions of $\mathcal{C}_{TAID}$. An example is shown in Figure 2. Since this paper has more a technical orientation, we are not detailing a biological example using $\mathcal{C}_{TAID}$.

Compiler

Once the description is done, it is passed to our compiler `al2asp`. As mentioned before, this program is able to handle the described languages and their different encodings that need to be given via command line options:

`al2asp -l c`	direct $\mathcal{C}$ to ASP encoding
`al2asp -l c_taid`	direct $\mathcal{C}_{TAID}$ to ASP encoding
`al2asp -l c_taid2c`	$\mathcal{C}_{TAID}$ to $\mathcal{C}$ encoding

While the two first commands yield a logic program[4], the last one outputs rules in $\mathcal{C}$.[5]

`al2asp` is implemented in `C++` and freely available at (BioASP Tools). Notably, `al2asp` relies on scanner and parser generators `flex` and `bison++`, making it easily amenable to language extensions.

[4]The direct $\mathcal{C}_{TAID}$ to ASP encoding implements a slightly modified encoding according to the one given in (Dworschak et al. 2008) that is not discussed in this paper.

[5]One can just reuse the tool to complete the encoding: `al2asp -l c_taid2c <file.desc> | al2asp -l c`.

An `al2asp` generated logic program containing variables appears incomplete. The additional logic program providing the binding information must be concatenated to the output of `al2asp` in order to get the resulting logic program that can be grounded. This ground program expresses the transition system described by the original description in $\mathcal{C}$.

ASP Tools

Reconsidering Figure 1, the resulting logic program is dealt with by an ASP system, consisting of a grounder and a solver component. As discussed, the logical representation of an action description may contain object variables that are passed on to the grounder. Our grounder, `gringo` (Gebser, Schaub, and Thiele 2007)[6], systematically replaces all variables by ground terms, while aiming at producing a compact propositional program. The resulting program is then passed to the ASP solver, `clasp` (Gebser et al. 2007b; 2007a)[7], which computes the stable models (see (Baral 2003) for details) of the program. Each such model represents a valid trajectory in the transition system induced by the original action description.

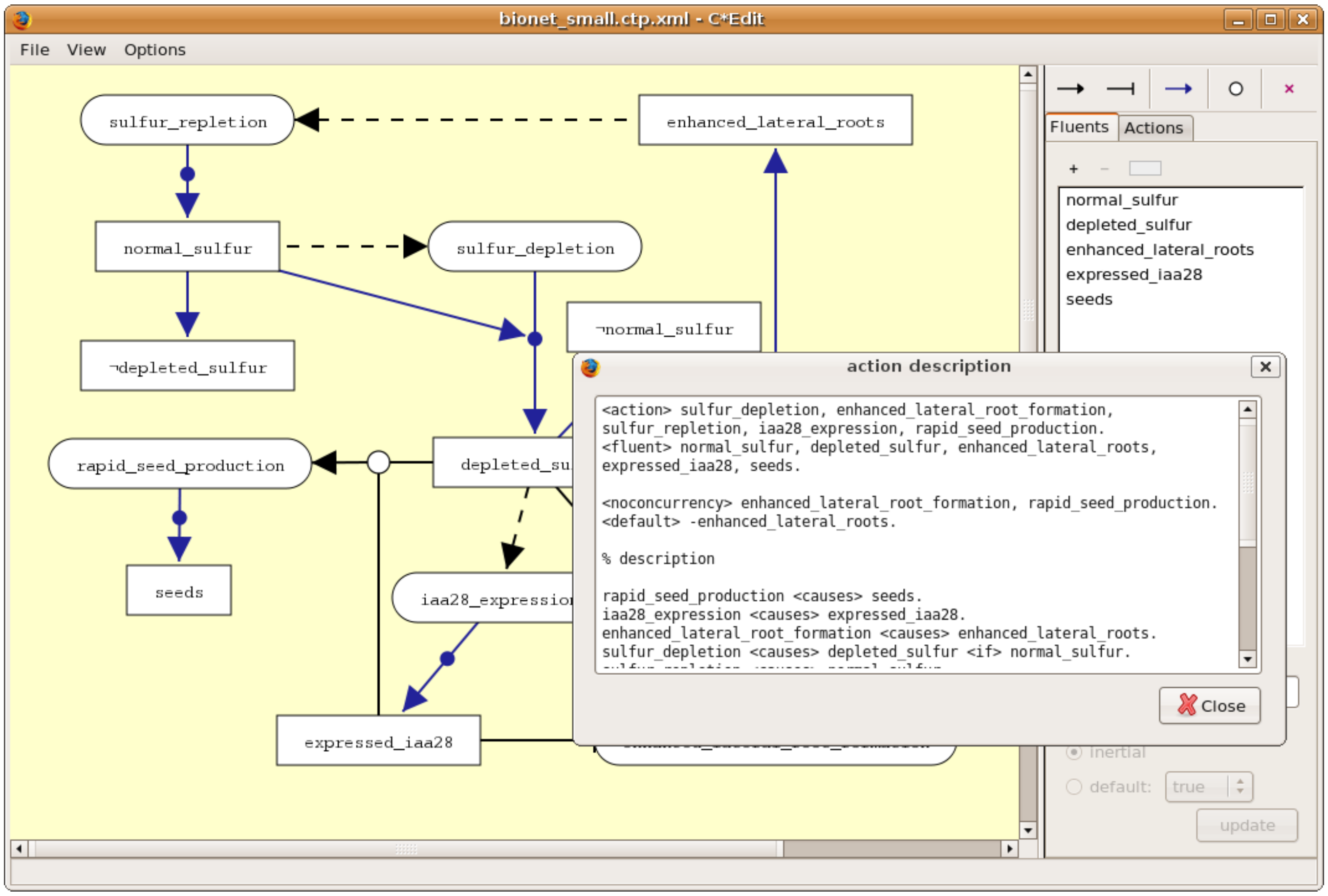
Backends

The action description for the BIOCHAM system combined with the underlying domain induces the transition system given in Figure 3.

Given that fluent and action names are changed in the logical encoding (ie. an additional time parameter appears as additional argument) as well as the obtained solutions appear in an unsorted way, the output of an ASP system must be transformed in a more readable and problem-oriented format. To this end, we offer different possibilities to present the output using the program `as2sth`: One possibility is a textual representation of the trajectories that gives a detailed overview of actions and states involved in a given solution. To illustrate this, recall our BIOCHAM example and consider the answer sets representing all 8 trajectories of length 1. Our interface displays them as follows.

[6]`http://www.cs.uni-potsdam.de/gringo`

[7]`http://www.cs.uni-potsdam.de/clasp`



```
<action> sulfur_depletion, enhanced_lateral_root_formation,
sulfur_repletion, iaa28_expression, rapid_seed_production.
<fluent> normal_sulfur, depleted_sulfur, enhanced_lateral_roots,
expressed_iaa28, seeds.

<noconcurrency> enhanced_lateral_root_formation, rapid_seed_production.
<default> -enhanced_lateral_roots.

% description

rapid_seed_production <causes> seeds.
iaa28_expression <causes> expressed_iaa28.
enhanced_lateral_root_formation <causes> enhanced_lateral_roots.
sulfur_depletion <causes> depleted_sulfur <if> normal_sulfur.
```

Figure 2: Screenshot of our graphical user interface for $\mathcal{C}_{TAID}$. Problem descriptions can be modeled in a graphical way by combining nodes with different arrows that correspond to rules in $\mathcal{C}_{TAID}$. The textual representation is generated by the program in order to process it with our compiler or to directly send the description to our web-based service.

```
## ANSWER 1 ################
0  A  +  occurs(r1)
0  F  +  present(a)
0  F  -  present(b)
1  F  -  present(a)
1  F  +  present(b)
## ANSWER 2 ################
...
## ANSWER 8 ################
0  F  +  present(a)
0  F  -  present(b)
1  F  +  present(a)
1  F  -  present(b)
## SUMMARY #################
models: 8
```

The first column denotes the timestep, the second one the type of the logic literal (action or fluent), the third one the value of the literal (true or false) and the last one the original name as used in the action description.

This method becomes inapplicable when the number of solutions increases, which is the case in most of the biological applications. To this end, another possibility is to generate `csv` output that can be processed with external programs like database systems, statistical tools, etc.

A third possibility is to use our built in `gnuplot` interface: We currently provide some statistical post-processing counting fluent values and actions at each time step in all trajectories. For example, let us assume that a fluent f appears to be true at a certain timestep t in all trajectories. When presenting all occurrences of fluents (or actions) in a graphical way, one can easily see that fluent f is essential for having solutions.[8] Although our simple BIOCHAM example focuses on the transition system (having no queries at all), the graphical representation can already be useful to get an idea. Reconsider the transition system given in Figure 3. Figure 4 is the graphical representation of all 128 trajectories having a length of 6. It is easy to see that there is a direct correspondence between the presence of compound `a` and `b`. While `a` tends to decrease, `b` tends to increase.[9] It is nearly

[8]This can also be seen as a cautious reasoning mode.

[9]Indeed, this outcome seems to be trivial since this is exactly the relation between the two compounds that was modeled before. But on larger scale examples it is possible to identify relations that

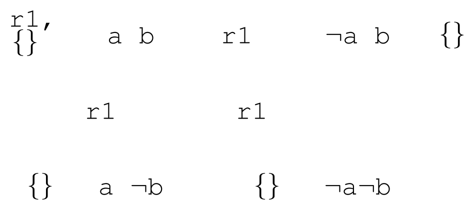


Figure 3: Transition system of the BIOCHAM example. `a` and `b` are shorthands for fluents `present(a)` and `present(b)`, `r1` is a shorthand for action `occurs(r1)`. {} denotes the empty action, that is, no action is executed in a transition labeled like this. Note that the loop at node {a, b} describes two transitions.

Figure 4: Graphical representation of all trajectories of the BIOCHAM example having length 6. Y-Axis denotes the percentage of true propositions (resp. the presence of compounds), and X-Axis denotes the timesteps. The two different bars represent the compounds `a` and `b`. For example, consider the second bar at timestep 1: it denotes that `present(b)` is true at timestep 1 in 50% out of all answer sets.

impossible to gain such information by only looking at the calculated trajectories.

Toolchain Access

The whole reasoning tool is accessible in two ways. The first possibility is to download the tools described in Figure 1 from (BioASP Tools) and to run them on a local machine. We are building up a graphical tool wrapping the underlying command-line execution of the described tools. By now, given that the tools are available on a Linux machine, a user may start the different programs via pipelining by hand. For example, if we have a our BIOCHAM description in $\mathcal{C}$ given in a file named `biocham.alc`, the domain specification given in a file named `biocham.stat` and want to display the chart as given in Figure 4 using gnuplot, you invoke on your local system the following commands:

```
$UNIX> al2asp -l c biocham.alc | \
       cat - biocham.stat | \
       gringo -c n=5 | clasp 0 | \
       as2sth --csv | \
       asplot present(a) present(b) \
       && gnuplot plot.plt
```

The second possibility is more user-friendly. To this end, we built up a web-based interface at (BioASP Tools), where the described tools are fully encapsulated as a server application. In this way, one can use the whole reasoning system without installing local applications. The mentioned Firefox-Plugin to describe $\mathcal{C}_{TAID}$ problems in a graphical way is able to access the web interface directly by sending the underlying action description to the web server. We added several examples on our web interface, where one can see how descriptions and queries to the system look like and how a user is able to access the different backends.

were not given explicitly in the action description.

Benchmarks

In this section, our core tools (`al2asp`, `gringo` and `clasp`) will be empirically compared to the systems *CCalc* (Giunchiglia et al. 2004) and dlv^k (Eiter et al. 2003a) since all of them use input languages based on $\mathcal{C}$. Unfortunately, the system *CPlan* (Castellini, Giunchiglia, and Tacchella 2003) is no longer maintained and the authors provided a windows executable only which was not usable in our benchmark setting.

The benchmarks were carried out on an Intel Core2Duo 6400 with 2.13GHz and 2 GB RAM running a 32-bit version of Ubuntu GNU/Linux. For our tests, we used `al2asp` v0.4, `gringo` v1.0.0 and `clasp` v1.0.5 with default settings. *CCalc* was used in version 2.0 and among the provided SAT solvers *grasp* was used. Although *grasp* does not provide the current state of the art SAT solving techniques, it was the only solver in our tests that produced all solutions. Regarding dlv^k, we used release 2007-10-11 with default settings.

Concerning pure planning problems, one is often interested in finding only the first solution. This issue is different in our approach, in most of the biological applications there is a need to consider all solutions. For example, recall Figure 4 where we need to process all answer sets in order to do statistical analysis. Biological queries to the system often lead to a large number of answers that need to be processed by biologists afterwards. Due to biologist's additional knowledge, some of the answers might make no sense in the real biological background and sometimes they want to figure out subsets satisfying certain constraints they did not know before. [2]To this end, we consider both cases when comparing the different systems, finding one, and finding all solutions. [2] SD: rewrote thi

Unfortunately, our current biological applications get solved too fast to make systems comparable. Being not generic[10], a comparison of different systems using our biological problems is not yet feasible. We use crafted artificial problems instead to compare performance of systems.

The first problem is the well known *blocks world* which consists of a table and several blocks. Given an initial state of piled up blocks, the planning system's task is to find out how to rearrange the blocks such that they are piled up in a predefined order. We used the dlv^kencoding and problem instances from (Eiter et al. 2003b). Due to advances in computer hardware, these old instances are solved too fast to get reasonable runtimes. That is why we came up with five additional instances (p6 - p10, see Table 1) which are still demanding for the systems running on today's hardware.

Our second benchmark suite *lights out*[11] is very similar to

[10]Unlike most artificial problems, we do not have parameters controlling the size of problem instances.

[11]Idea taken from General Gameplaying Competition 2008.

No.	Instance	length	`bioplan` - one	$CCalc$ - one	dlv^k - one	`bioplan` - all	$CCalc$ - all	dlv^k - all
1	p01	05	0.31	0.51	0.06	0.32	0.51	0.06
2	p02	06	0.22	0.35	0.05	0.22	0.42	0.05
3	p03	08	1.19	1.21	1.21	1.21	8.23	4.57
4	p04	09	3.89	3.88	1.17	4.05	5.74	15.19
5	p05	11	5.04	5.39	2.92	4.92	11.24	22.98
6	p06	13	4.21	3.88	21.15	4.78	408.23	—
7	p07	14	8.76	7.08	42.04	11.81	364.22	—
8	p08	16	36.88	14.28	—	133.49	—	—
9	p09	16	39.39	129.15	—	41.75	—	—
10	p10	17	66.68	—	—	85.83	—	—
Average Time (Sum Timeouts)			17.49 (0)	19.47 (3)	10.54 (9)	20.61 (0)	122.87 (9)	9.58 (15)
Average Penalized Time			17.49	77.52	187.38	20.61	266.01	304.79

Table 1: Blocks world experiments computing one and all solutions

No.	Instance	length	`bioplan`	$CCalc$	dlv^k
1	l1nc	10	0.10	0.14	17.81
2	l2nc	15	0.20	0.19	—
3	l3nc	20	2.12	0.26	—
4	l4nc	25	—	0.39	—
5	l1c	1	8.63	—	2.43
6	l2c	1	17.39	—	5.24
7	l3c	1	26.41	—	8.09
8	l4c	1	35.43	—	10.56
Average Time (Sum Timeouts)			12.90 (3)	0.24 (12)	8.82 (9)
Average Penalized Time			86.29	300.12	230.51

Table 2: Lights out experiments computing one solution

the *bomb in the toilet* problem: All of a variable number of light bulbs has to be switched off. In every state, every light can either be switched on or off. The problem comes in two flavors, either with concurrent execution of actions allowed or with concurrency disabled. The optimal[12] plan length in the latter case is equal to the number of light bulbs. It is easy to see that this problem leads to $n!$ many optimal plans regarding n bulbs that only differ in the sequence of switching off bulbs. Due to this behaviour, we omit computing all solutions as in the blocks world setting.

The results of the *blocks world* benchmarks are listed in Table 1 and the *lights out* results are in Table 2. For every problem instance, we measured the time in seconds of three separate solving processes and computed the average which is shown in each systems column. A dash indicates that a system was unable to compute a solution in less than 600 seconds. The column labeled *length* denotes the length of the shortest possible plan(s) for the problem instance which is passed as a parameter to the different systems. The last row in the tables lists penalized average times. In contrast to normal average times, the penalized ones take timeouts into account. Although the system might have taken much longer to find a solution, the penalized average is computed as if the system found a solution after 600 seconds.

Results show that compared to the other systems our system performs quite well and appears to be robust. In the *blocks world* example, it was the only system that could enumerate all solutions in reasonable time. As mentioned, this issue is especially valuable because our biological applications often need all solutions to be computed. But also when only one solution has to be found, our system outperformed both $CCalc$ and dlv^k. $CCalc$'s performance was comparable to ours until the problems became too hard in benchmark number 9.

Regarding the *lights out* problem, $CCalc$ performs surprisingly well when concurrent execution of actions is not allowed. It computes a solution almost instantly, while dlv^k has difficulties even in the smallest instance. Although being quite fast with a few light bulbs, the runtime of our system rises rapidly as soon as more than twenty bulbs are involved. When allowing concurrency in this example, dlv^k is the fastest system. $CCalc$ seems to have great problems with the huge[13] number of light bulbs which was used in the problem instances and is unable to find a solution in any instance. Our system performs quite well in this benchmark, though not as well as dlv^k. In general, the benchmarks show that our system is more than competitive compared to other planning systems.

Discussion

Although we motivate (and apply) our approach in a biological setting, many features are readily applicable to representing and reasoning about dynamical systems in general. Centering our approach on $\mathcal{C}$ has several benefits. First, $\mathcal{C}$ is a rich and well-studied formalism. Second, it constitutes a mainstream implementation line for action languages. To this end, we provided a translation of the biologically motivated action language $\mathcal{C}_{TAID}$ to $\mathcal{C}$ and devise several tools for dealing with action descriptions in $\mathcal{C}$ (and $\mathcal{C}_{TAID}$). Among them, we implemented the compiler `al2asp` allowing for translating action descriptions in $\mathcal{C}$ (and $\mathcal{C}_{TAID}$) to logic programs under answer sets semantics. This approach is similar to the one taken by dlv^k for processing action language $\mathcal{K}$. Both approaches exploit the

[12]Optimal means that there is at least one solution at bound t, but no solution can at bound $t-1$.

[13]25.000 bulbs in instance *l1c* up until 100.000 bulbs in instance *l4c*.

grounding and solving capacities of ASP, offering uniform (and thus instance independent) problem encodings and easy variable handling. Our approach is supported by a variety of pragmatic yet indispensable tools for addressing real world applications. Compared to other planning systems, we are able to compete with, and sometimes even outperform current systems. Finally, our tools (as well as their source code) and the benchmark problems are freely available at (BioASP Tools).

References

Baral, C.; Chancellor, K.; Tran, N.; Tran, N.; Joy, A.; and Berens, M. 2004. A knowledge based approach for representing and reasoning about signaling networks. In *Proceedings of the Twelfth International Conference on Intelligent Systems for Molecular Biology/Third European Conference on Computational Biology (ISMB'04/ECCB'04)*, 15–22.

Baral, C.; Brewka, G.; and Schlipf, J., eds. 2007. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

BioASP Tools. `http://www.cs.uni-potsdam.de/wv/bioasp`.

Castellini, C.; Giunchiglia, E.; and Tacchella, A. 2003. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence* 147(1-2):85–117.

Chabrier-Rivier, N.; Chiaverini, M.; Danos, V.; Fages, F.; and Schächter, V. 2004. Modeling and querying biomolecular interaction networks. *Theor. Comput. Sci.* 325(1):25–44.

Dworschak, S.; Grell, S.; Nikiforova, V.; Schaub, T.; and Selbig, J. 2008. Modeling biological networks by action languages via answer set programming. *Constraints* 13(1-2):21–65.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003a. A logic programming approach to knowledge-state planning. *Artificial Intelligence* 144(1-2):157–211.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003b. A logic programming approach to knowledge-state planning, ii: The dlv$^{\text{k}}$ system. *Artificial Intelligence* 144(1-2):157–211.

Fages, F.; Sollman, S.; and Chabrier-Rivier, N. 2004. Modelling and querying interaction networks in the biochemical abstract machine biocham. *Journal of Biological Physics and Chemistry* 4:64–73.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007a. clasp: A conflict-driven answer set solver. In Baral et al. (2007), 260–265.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007b. Conflict-driven answer set solving. In Veloso, M., ed., *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, 386–392. AAAI Press/The MIT Press. Available at http://www.ijcai.org/papers07/contents.php.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. GrinGo: A new grounder for answer set programming. In Baral et al. (2007), 266–271.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence* 3(6):193–210.

Giunchiglia, E., and Lifschitz, V. 1998. An action language based on causal explanation: Preliminary report. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 623–630.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence* 153(1-2):49–104.

Lifschitz, V., and Turner, H. 1999. Representing transition systems by logic programs. In Gelfond, M.; Leone, N.; and Pfeifer, G., eds., *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, volume 1730 of *Lecture Notes in Artificial Intelligence*, 92–106. Springer-Verlag.

Tran, N., and Baral, C. 2004. Reasoning about triggered actions in AnsProlog and its application to molecular interactions in cells. In Dubois, D.; Welty, C.; and Williams, M., eds., *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, 554–564. AAAI Press.

Tran, N. 2006. *Reasoning and hypothesing about signaling networks*. Ph.D. Dissertation, Arizona State University.