Component-based Answer Set Programming*

Stefania Costantini

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{stefcost}@di.univaq.it

Abstract. In previous work we have discussed the importance of identifying the cycles that occur in a logic program under the answer set semantics, and the connections between cycles, that we have called *handles*. We have proved that answer sets of the overall program are composed of answer sets of suitable sub-programs, corresponding to the cycles. We have defined the *cycle graph* CG, where each vertex of this graph corresponds to one cycle, and each edge corresponds to one *handle.* On the CG, we can check consistency by checking the existence of certain subgraphs. In this paper we will show that cycles and cycle graphs can be generalized to *components* and *component graphs.* By suitably arranging the handles connecting components, larger consistent programs can be built out of smaller one. Components can be understood as *agents*, where one agent communicates with another one if there is a handle which puts them in connection, and the agent makes it *active*.

1 Introduction

An important hot topic is, in our opinion, that of defining software engineering principles for Answer Set Programming, which is an alternative logic programming paradigm [GelLif88] [GelLif91] based on the Answer Set Semantics of [MarTru99] [Lif99], and put into practice by means of effective inference engines [Systems]. Also, it is important to have the possibility of defining significant subprograms as *components* to be possibly distributed over different nodes of a network. Both the connections between components and the ways of exchanging information should be clearly defined.

In previous work [Cos03] we have discussed the importance of identifying the cycles that occur in a logic program under the answer set semantics, and the connections between cycles, that we have called *handles*. We have proved that answer sets of the overall program are composed of answer sets of suitable sub-programs, corresponding to the cycles. We have defined the *cycle graph* CG, where each vertex of this graph corresponds to one cycle, and each edge corresponds to one *handle*, which is a literal containing an atom that, occurring in both cycles, actually determines a connection between them. In fact, the truth value of the handle in the cycle where it appears as the head of a rule, influences the truth value of the atoms of the cycle(s) where it occurs in

* We acknowledge support by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

the body. Cycles can be even, if they consist of an even number of rules, or vice versa they can be odd. Problems for consistency, as it is well-known, originate in the odd cycles. If for every odd cycle we can find a subgraph of the CG with certain properties, then the existence of answer sets is guaranteed.

In this paper we will show that cycles and cycle graphs can be generalized to components and *component graphs.* Precisely, a *component* $\mathcal{C}$ is defined to be a bunch of cycles. It can be developed on its own, or it can be identified on the CG of a larger program. Previous results allow us to define conditions on the *input handles* that keep a component consistent. Symmetrically, the *out-handles* of $\mathcal{C}$ are atoms belonging to its answer sets that correspond to input handles of other components.

We will propose a program development methodology for Answer Set Programming based building larger consistent programs by suitably combining components. On the $CompCG$ in fact, the connection between components are easily identified. In particular, its edges correspond to the handles of each component, whose value may possibly be provided by other components. On the $CompCG$, one can either add new consistent components, or modify existing ones, and can check on the edges if there are problems for consistency, and how to fix them.

In this framework, components may even be understood as independent agents, and providing a suitable value for an handle of a component may be understood as sending a message to the component itself. Consider the following example, representing a fragment of the code of a *controller* component/agent:

$circuit\_ok \leftarrow \textit{not}\ fault$
$fault \leftarrow \textit{not}\ fault, \textit{not}\ test\_ok$
where $test\_ok$ is an incoming handle, coming from a *tester* component/agent. As soon as the *tester* will achieve $test\_ok$, this incoming handle will become true, thus making the *controller* consistent, and able to conclude $circuit\_ok$.

In Section 2 we define cycles, handles, and components. In Section 3 we define the cycle graph CG and discuss how to check consistency on the CG.

In Section 4 we discuss how to generalize the CG to build the $CompCG$, and propose the guidelines for a program development methodology for Answer Set Programming based on defining components and connecting them on the $CompCG$ (a full definition is postponed to the extended version of the paper). We will then argue that components can be understood as independent agents, even located on different nodes of a network. We will illustrate the proposed approach by means of a concrete example. where we will also show that components/agents can be identified on the non-ground version of the program.

2 Preliminary Definitions, Cycles and Handles

In this paper we consider the language $DATALOG^{\neg}$ for deductive databases, which is more restricted than traditional logic programming (the reader may refer to [MarTru99] for a discussion). In the following, we will implicitly refer to the ground version of $DATALOG^{\neg}$ programs.

A logic program may contain negated atoms of the form $\neg a$. A rule ρ is defined as usual, and can be seen as composed of a conclusion $head(\rho)$, and a set of conditions $body(\rho)$. The latter can be divided into positive conditions $pos(\rho)$ each one of the form A, and negative conditions $neg(\rho)$, each one of the form *not* A. The literal A is either an atom a, or a negated atom $\neg a$.

The answer sets semantics [GelLif88,GelLif91] is a view of logic programs as sets of inference rules (more precisely, default inference rules). Alternatively, one can see a program as a set of constraints on the solution of a problem, where each answer set represents a solution compatible with the constraints expressed by the program. Consider the simple program $\{q \leftarrow$ *not* $p.\ \ p \leftarrow$ *not* $q.\}$. For instance, the first rule is read as "assuming that p is false, we can *conclude* that q is true." This program has two answer sets. In the first one, q is true while p is false; in the second one, p is true while q is false.

A subset M of the Herbrand base B_P of a $DATALOG^{\neg}$ program P is an answer set of P, if M coincides with the least model of the reduct P^M of P with respect to M. This reduct is obtained by deleting from P all rules containing a condition *not* a, for some a in M, and by deleting all negative conditions form the other rules. Answer sets are minimal supported models, and form an anti-chain.

Unlike with other semantics, a program may have several answer sets, or may have no answer set, because conclusions are included in an answer set only if they can be justified. The following program has no answer set:
$\{a \leftarrow$ *not* $b.\ b \leftarrow$ *not* $c.\ c \leftarrow$ *not* $a.\}$
The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom. Whenever a program has no answer sets, we will say that the program is *inconsistent*. Correspondingly, checking for consistency means checking for the existence of answer sets. Assume the standard definitions of (propositional) general logic program and of answer set semantics [GelLif88]. Whenever we mention consistency (or stability) conditions, we refer to the conditions introduced in [GelLif88]. Let Π be a general logic program. In the following, we will often simply say "logic program" to mean a general logic program. By abuse of notation, we will often say *stable models* to mean *answer sets*.

The rest of this paper is heavily based on identifying the cycles that occur in a logic program under the answer set semantics, and the connections between cycles. Below is the definition of a cycle:

Definition 1. *A set of rules C is called a cycle if it has the following form:*

$\lambda_1 \leftarrow$ *not* λ_2, Δ_1
$\lambda_2 \leftarrow$ *not* λ_3, Δ_2
$\dots$
$\lambda_n \leftarrow$ *not* λ_1, Δ_n

where $\lambda_1, \dots, \lambda_n$ are distinct atoms. Each Δ_i, $i \leq n$, is a (possibly empty) conjunction $\delta_{i_1}, \dots, \delta_{i_h}$ of literals (either positive or negative), where for each δ_{i_j}, $i_j \leq i_h$, $\delta_{i_j} \neq \lambda_i$ and $\delta_{i_j} \neq$ not λ_i. The Δ_i's are called the AND handles of the cycle. We say that Δ_i

is an AND handle for atom λ_i, or, equivalently, an AND handle referring to λ_i. Δ_i is called simple *if it is composed of a single literal.*

We say that C has size n and it is even (respectively odd) if $n = 2k$, $k \geq 1$ (respectively, $n = 2k + 1$, $k \geq 0$). By abuse of notation, for $n = 1$ we have the (odd) self-loop $\lambda_1 \leftarrow$ *not* λ_1, Δ_1. In what follows, again by abuse of notation λ_{i+1} will denote $\lambda_{(i+1) mod\, n}$, i.e., $\lambda_{n+1} = \lambda_1$.

Given cycle C, we call $Composing\_atoms(C) = \{\lambda_1, \ldots, \lambda_n\}$ the set containing all the atoms *involved* in cycle C. We say that the rules listed above are *involved* in the cycle, or *form* the cycle. In the rest of the paper, sometimes it will be useful to see $Composing\_atoms(C)$ as divided into two subsets, that we indicate as two *kinds* of atoms: the set of the *Even_atoms(C)* composed of the λ_i's with i even, and the set *Odd_atoms(C)*, composed of the λ_i's with i odd.

Conventionally, in the rest of the paper with C or C_i we will refer to cycles in general, with OC or OC_i to odd cycles and with EC or EC_i to even cycles

In the following program for instance, there is an odd cycle, that we may call OC_1, with composing atoms $\{e, f, g\}$ and an even cycle, that we may call EC_1, with composing atoms $\{a, b\}$.

— OC_1
$e \leftarrow$ *not* f, *not* a
$f \leftarrow$ *not* g, b
$g \leftarrow$ *not* e
— EC_1
$a \leftarrow$ *not* b
$b \leftarrow$ *not* a
OC_1 has an AND handle *not* a referring to e, and an AND handle b referring to f.

Definition 2. *A rule is called an* auxiliary rule of cycle C *(or, equivalently,* to *cycle C) if it is of this form:*
$\lambda_i \leftarrow \Delta$
where $\lambda_i \in$ Composing_Atoms(C), and Δ is a non-empty conjunction $\delta_{i_1}, \ldots, \delta_{i_h}$ of literals (either positive or negative), where for each δ_{i_j}, $i_j \leq i_h$, $\delta_{i_j} \neq \lambda_i$ and $\delta_{i_j} \neq$ not λ_i. Δ is called an OR handle of cycle C (more specifically, an OR handle for λ_i or, equivalently, and OR handle referring to λ_i).

A cycle may possibly have several auxiliary rules, corresponding to different OR handles. In the rest of this paper, we will call *Aux(C)* the set of the auxiliary rules of a cycle C.

In the following program for instance, there is an odd cycle OC_1 with composing atoms $\{c, d, e\}$ and an even cycle EC_1 with composing atoms $\{a, b\}$. The odd cycle has three auxiliary rules.

— OC_1
$c \leftarrow$ *not* d
$d \leftarrow$ *not* e
$e \leftarrow$ *not* c
—— $Aux(OC_1)$
$c \leftarrow$ *not* a
$d \leftarrow$ *not* a
$d \leftarrow$ *not* b
— EC_1
$a \leftarrow$ *not* b
$b \leftarrow$ *not* a

In particular, we have $Aux(OC_1) = \{c \leftarrow \textit{not } a, d \leftarrow \textit{not } a, d \leftarrow \textit{not } b\}$.

In summary, a cycle may have some AND handles, occurring in one or more of the rules that form the cycle itself, and also some OR handles, occurring in its auxiliary rules.

A cycle may also have no AND handles and no OR handles, i.e., no handles at all, in which case it is called *unconstrained.*The following program is composed of unconstrained cycles (in particular, there is an even cycle involving atoms a and b, and an odd cycle involving atom p).

— EC_1
$a \leftarrow$ *not* b
$b \leftarrow$ *not* a
— OC_1
$p \leftarrow$ *not* p

It is useful to collect the set of handles of a cycle into a set, where however each handle is annotated so as to keep track of its *kind.* I.e., we want to remember whether a handle is an OR handle or an AND handle of the cycle.

Definition 3. *Given cycle C, the set H_C of the handles of C is defined as follows, where $\beta \in Composing\_Atoms(C)$:*

$$H_C = \{(\Delta : AND : \beta) \mid \Delta \textit{ is an AND handle of } C \textit{ referring to } \beta\} \cup \\ \{(\Delta : OR : \beta) \mid \Delta \textit{ is an OR handle of } C \textit{ referring to } \beta\}$$

Whenever we need not care about β we shorten $(\Delta : K : \beta)$ as $(\Delta : K)$, K = AND/OR. By abuse of notation, we call “handles” the expressions in both forms, and whenever necessary we implicitly shift from one form to the other one. Informally, we will say for instance “the OR (resp. AND) handle Δ of β” meaning $(\Delta : OR : \beta)$ (resp. $(\Delta : AND : \beta)$).

Definition 4. *Let C be a cycle. The program $C + Aux(C)$ is the* completed cycle *corresponding to C.*

Definition 5. *Let C be a cycle. Let $Z \subseteq H_C$. The program $C + Aux(C) + Z$ is an* extended cycle *corresponding to C.*

Definition 6. *Let C_i be a cycle occurring in Π. We say that $S_{C_i} \subseteq \mathbb{B}_{\mathrm{C_i+Aux(C_i)}}$ is a* partial stable model *for Π relative to C_i, if $\exists X_i \subseteq Atoms(H_{C_i})$ such that S_{C_i} is a stable model of the corresponding extended cycle $C_i + Aux(C_i) + X_i$. The set X_i is called a positive base for S_{C_i}, while the set $X_i^- = Atoms(H_{C_i}) \setminus X_i$ is called a negative base for S_{C_i}. The couple of sets $\langle X_i, X_i^- \rangle$ is called a base for S_{C_i}.*

In the ongoing, for the sake of simplicity we consider only simple OR handles, and simple AND handles, i.e., handles composed of a single literal. More precisely, we consider logic programs in a special *canonical form*. Rules in a program are in a simple uniform format. This is for the sake of clarity, and without loss of generality. Every program in fact can be reduced to this form [CosPro03]. The formal properties of the canonical form and the algorithm for obtaining it are discussed in [CosPro03]. Here we need to notice that a program in canonical form has no positive circularities, and cannot have an exponential number of cycles.

Definition 7. *A logic program Π is in canonical form (or, equivalently, Π is a canonical program) if it is negative (i.e., does not contain positive literals), and fulfills the following syntactic conditions.*

1. *every atom in Π occurs both in the head and in the body of some rule;*
2. *every atom in Π is involved in some cycle;*
3. *each rule of Π is either involved in a cycle, or is an auxiliary rule of some cycle;*
4. *each handle of a cycle C consists of exactly one literal, either α or not α, where atom α does not occur in C.*

Since the above definition requires handles to consist of just one literal, it implies that in a canonical program Π : (i) the body of each rule which is involved in a cycle consists of either one or two literals; (ii) the body of each rule which is an auxiliary rule of some cycle consists of exactly one literal.

Handles may help avoid inconsistencies in two ways. An AND handle Δ_i which is false allows the head λ_i of the rule to be false. An OR handle Δ which is true forces the atom λ_i to which it refers to be true as well. This idea is formalized into the following definitions of *active handles.*

Definition 8. *Let $\mathcal{I}$ be an interpretation. An AND handle Δ of cycle C is active w.r.t. $\mathcal{I}$ if the corresponding literal is false w.r.t. $\mathcal{I}$. We say that the rule where the handle occurs has an active AND handle. An OR handle Δ of cycle C is active w.r.t. $\mathcal{I}$ if the corresponding literal is true w.r.t. $\mathcal{I}$. We say that the rule where the handle occurs has an active OR handle.*

Assume that $\mathcal{I}$ is a model. We can make the following observations. (i) The head λ of a rule ρ with an active AND handle is not required to be true in $\mathcal{I}$. (ii) The head of a rule $\lambda \leftarrow \Delta$ where Δ is an active OR handle is necessarily true in $\mathcal{I}$: since the body is true, the head λ must also be true.

Observing which are the active handles of a cycle C gives relevant indications about whether an interpretation $\mathcal{I}$ is a stable model.

Consider for instance the following program:

— OC_1
$p \leftarrow$ *not* p, *not* a
— EC_1
$a \leftarrow$ *not* b
$b \leftarrow$ *not* a
— OC_2
$q \leftarrow$ *not* q
—– $Aux(OC_2)$
$q \leftarrow f$
— EC_2
$e \leftarrow$ *not* f
$f \leftarrow$ *not* e

Interpretations $\{a, f, q\}$, $\{a, e, q\}$, $\{b, p, f, q\}$ $\{b, p, e, q\}$ are minimal models. Consider interpretation $\{a, f, q\}$: both the AND handle *not* a of cycle $p \leftarrow$ *not* p, *not* a and the OR handle f of cycle $q \leftarrow$ *not* q are active w.r.t. this interpretation. $\{a, f, q\}$ is a stable model, since atom p is forced to be false, and atom q is forced to be true, thus avoiding the inconsistencies. In all the other minimal models instead, one of the handles is not active. I.e., either literal *not* a is true, and thus irrelevant in the context of a rule which is inconsistent, or literal f is false, thus leaving the inconsistency on q. These minimal models are in fact not stable.

In conclusion, the example suggests that for a minimal model $\mathcal{M}$ to be stable, each odd cycle must have an active handle. This will be stated formally in the next section.

Another thing that the example above shows is that the stable model $\{a, f, q\}$ of the overall program is actually the union of the stable model $\{a\}$ of the program fragment $OC_1 \cup EC_1$ and of the stable model $\{f, q\}$ of the program fragment $OC_2 \cup Aux(OC_2) \cup EC_2$.

In fact, for checking whether a logic program has answer sets (and, possibly, for finding them) one can do the following.

(i) Divide the programs into pieces, of the form $C_i + Aux(C_i)$, and check whether every odd cycle has handles; if not, then the program is inconsistent;
(ii) For every cycle C_i with handles, find the sets X_i that make the subprogram $C_i + Aux(C_i)$ consistent, and find the corresponding answer sets S_{C_i}'s; notice that in the case of unconstrained even cycles, H_{C_i} is empty, and we have two answer sets, namely $M^1_{C_i}$ = Even_atoms(C_i) and $M^2_{C_i}$ = Odd_atoms(C_i).
(iii) Check whether there exists a collection of X_i's, one for each cycle, such that the corresponding S_{C_i}'s agree on shared atoms (and then are called *compatible*): in this case the program is consistent, and its answer set(s) can be obtained as the union of the S_{C_i}'s.

The following theorem (proved in [Cos03]) formally states the connection between the answer sets of Π, and the partial answer sets of its cycles.

Theorem 1. *An interpretation $\mathcal{I}$ of Π is a stable model if and only if there exists a compatible set $\mathcal{S} = S_1, \ldots, S_w$ of partial answer sets for its composing cycles such that $I = \bigcup_{i \leq w} S_i$.*

3 The Cycle Graph CG

Clearly, it is possible to uniquely identify the set $\{C_1, \ldots, C_w\}$ of the cycles that occur in program Π. This set can be divided in the two disjoint subsets of the even cycles $\{EC_1, \ldots, EC_g\}$, and of the odd cycles $\{OC_1, \ldots, OC_h\}$.

Then, the program structure in terms of cycles, handles and handle paths can be described by means of a graph, where cycles are the vertices and handles are the edges. Below in fact we introduce the novel notion of a cycle graph.

Definition 9. *The Cycle Graph CG_{Π}, is a directed graph defined as follows:*

– **Vertices.** *One vertex for each of cycles $C_1, \ldots, C_w$. Vertices corresponding to even cycles are labeled as EC_i's while those corresponding to odd cycles are labeled as OC_j's.*
– **Edges.** *An edge (C_j, C_i) marked with $(\Delta : K : \lambda)$ for each handle $(\Delta : K : \lambda) \in H_{C_i}$ of cycle C_i, that comes from C_j.*

Each marked edge will be denoted by $(C_j, C_i | \Delta : K : \lambda)$, where however by abuse of notation either $(C_j$ or C_i or $\lambda)$ will be omitted whenever they are clear from the context, and we may write for short $(C_j, C_i | h)$, h standing for a handle that is either clear from the context or does not matter in that point.

An edge on the CG connects the cycle a handle comes from to the cycle to which the handle belongs. Edges on the CG make it clear that handles *connect* different cycles: a handle Δ being or not being assumed to be active, corresponds to the atom α which occurs in Δ to be required to take a certain truth value in the cycles the handle comes from, depending of the kind of the handle. Precisely, if α is required to be true, then it must be concluded true in at least one of the cycles it is involved in. If α is required to be false, it must be concluded false in all cycles it is involved in.

As a consequence of Theorem 1, the odd cycles need to have at least one active handle, since on their own they would be inconsistent. If such a handle comes from another odd cycle, then we can repeat the same reasoning. Therefore, any odd cycle, for being consistent, must be directly or indirectly connected to some even cycle, through a "chains" of handles. On the CG, for every odd cycle it is possible to check whether such a connection may exist.

First, one has to check that any odd cycle OC has at least one handle. Secondly, one has to check that the different handles marking the edges of the CG are compatible, in the sense that the truth values required for the atoms occurring in the handles so as to make some of them active and some of them non-active as required by the definitions below should not be in contrast (i.e., the same atom cannot be required to be both true and false).

Last, one has to check that there may exist partial stable models for the cycle C the handle comes from, so as to make that handle active.

From [Cos03] we take the following definition (where we basically leave the concept of incompatible handles to the reader's intuition):

Definition 10. *Given program Φ, let a* CG support set *be a couple*

$$S = \langle ACT^{+}, ACT^{-} \rangle$$

of subsets of the handles marking the edges of CG_{Φ}, represented in the form $(\Delta : K)$ $(K = AND/OR)$, where handles in ACT^{+} are supposed to be active, and handles in ACT^{-} are supposed to be not active, and we have:
(i) $ACT^{+} \cap ACT^{-} = \emptyset$.
(ii) neither ACT^{+} nor ACT^{-} contain a couple of incompatible handles.

Given S, we will indicate its two components with $ACT^{+}(S)$ and $ACT^{-}(S)$.

A CG support set represents the handles that are supposed to be active/not active for making the odd cycles and the whole program consistent.

As discussed before, consistency is strongly conditioned by the odd cycles of the program. So, we have to restrict the attention on CG support sets including at least one active handle for each odd cycle, and then we have to check that the assumptions on the handles are mutually coherent, and are sufficient for ensuring consistency. An CG support set is *potentially adequate* if it provides at least one active incoming handle for each of the odd cycles.

Definition 11. *An CG support set S is potentially adequate if for every odd cycle C in Π there exists a handle $h \in H_C$ such that $h \in ACT^{+}(S)$.*

The definition of *adequate* support sets (omitted here) formalizes a more strict requirement, to make sure that the requirement to be active/non-active for the handles occurring in a support set can possibly be fulfilled in the cycles they come from.

Then, we have proved the following:

Theorem 2. *A program Π has answer sets if and only if there exists and adequate CG support set S for Π.*

4 The methodology

In this Section, we aim at proposing a view of a logic program under the answer set semantics as a collection of components, where a component is a set completed cycles. This novel view leads to new ways of building programs, and to new ways of checking consistency of large programs. In fact, a large program can be assembled starting from existing components, and its consistency can be checked simply by checking the *connections* between components, i.e., the handles that join them. Also, we will discuss how components can be understood as agents.

Let a *component* $\mathcal{C}$ be a bunch of cycles, plus the indication of its incoming handles whose value can be possibly provided by other components.

Definition 12. *A component $\mathcal{C}$ is a set of completed cycles, each of the form $C + Aux(C)$, together with a set $H_{\mathcal{C}}$ composed of some of the incoming handles of these extended cycles. We call $H_{\mathcal{C}}$ the set of the* input handles *of $\mathcal{C}$.*

Definition 13. *A program is a set $\mathcal{S} = \{\mathcal{C}_1, \ldots, \mathcal{C}_s\}$ of components.*

A component can be developed on its own, or it can be identified on the CG of a larger program. Similarly to a cycle however, $\mathcal{C}$ is not meant to be an independent program, rather it is connected to other components by means of handles.

Here we mean to propose a program development methodology for Answer Set Programming based on defining, over the CG, a higher level graph where vertices are *components,* and edges are a subset of the edges of the CG, connecting components instead of single cycles.

Definition 14. *The Component Graph $CompG_{\Pi}$, is a directed graph defined as follows:*

– **Vertices.** *One vertex for each of the components in $\mathcal{S}$*
– **Edges.** *An edge $(\mathcal{C}_j, \mathcal{C}_i)$ marked with $(\Delta : K : \lambda)$ for each handle $h = (\Delta : K : \lambda) \in H_{\mathcal{C}_i}$ of component $\mathcal{C}_i$, that comes from $\mathcal{C}_j$.*

Each marked edge will be denoted, like for the CG, by $(\mathcal{C}_j, \mathcal{C}_i | \Delta : K : \lambda)$. An edge $(\mathcal{C}_j, \mathcal{C}_i)$ marked with handle $h = (\Delta : K : \lambda)$ means that component $\mathcal{C}_i$ *accepts as input* the value of Δ from component $\mathcal{C}_j$.

We give below a sketch description of the methodology, that we will illustrate by means of an example. A full definition is postponed to the extended version of the paper. The input handles of each component will be listed in the short form $\Delta : K$.

Let us consider the following example. We have a traffic light (called tl) that, for the sake of simplicity, can just take the colors green (g for short) and red (r for short). We have two lanes, one going north-south (ns for short) and the other one east-west (ew for short), crossing at the traffic light. If the traffic light is green in one direction it must be red in the other one, and vice versa. The traffic light is subject to faults, ans precisely it does not work whenever $fault\_tl$ is true. This is an input handle which should come from a “controller” component, not specified below.

Each car $c1$, $c2$, $c3$ wants to go, but it is allowed to go only if it gets the green traffic light. Otherwise, it remains dummy. Then, all cars behave in the same way. Only, cars $c1$ and $c2$ want to go north-south, and then their input handles consist in the possible colors of the traffic light for lane north-south. Instead, car $c3$ want to go east-west, and consequently its incoming handles consist in the possible colors of the traffic light for lane east-west.

— Component: The traffic light
— Input handles: *not* $fault\_tl : AND$
— — Cycle lane ns
$tl(g, ns) \leftarrow$ *not* $tl(r, ns),$ *not* $fault\_tl.$
$tl(r, ns) \leftarrow$ *not* $tl(g, ns),$ *not* $fault\_tl.$
$tl(r, ns) \leftarrow tl(g, ew).$
$tl(g, ns) \leftarrow tl(r, ew).$

— — Cycle lane ew
$tl(g, ew) \leftarrow$ *not* $tl(r, ew),$ *not* $fault\_tl.$
$tl(r, ew) \leftarrow$ *not* $tl(g, ew),$ *not* $fault\_tl.$
$tl(r, ew) \leftarrow tl(g, ns).$
$tl(g, ew) \leftarrow tl(r, ns).$

— Component(cycle): car c1
— Input handles: *not* $tl(r, ns) : AND$, $tl(g, ns) : OR$
$go(c1, ns) \leftarrow$ *not* $go(c1, ns),$ *not* $tl(r, ns).$
$go(c1, ns) \leftarrow tl(g, ns).$

— Component(cycle): car c2
— Input handles: *not* $tl(r, ns) : AND$, $tl(g, ns) : OR$
$go(c2, ns) \leftarrow$ *not* $go(c2, ns),$ *not* $tl(r, ns).$
$go(c2, ns) \leftarrow tl(g, ns).$

— Component(cycle): car c3
— Input handles: *not* $tl(r, ew) : AND$, $tl(g, ew) : OR$
$go(c3, ew) \leftarrow$ *not* $go(c3, ew),$ *not* $tl(r, ew).$
$go(c3, ew) \leftarrow tl(g, ew).$

It is easy to see that the above program has two answer sets, one when we have the green light on the north-south lane, and the other one when we have the green light on the east-west lane. In the former case $c1$ and $c2$ can go, while $c3$ is dummy. In the latter case it is $c3$ that can go.

On the CG, we would have as many nodes as the cycle. In particular, the traffic light subprogram is divided into two cycles, each one with two incoming AND handles and two incoming OR handles. Each car is a cycle on its own, with an AND incoming handle that, if active, means that the car cannot go, and an incoming OR handle that, if active, means that the car can go.

Instead, in the $CompCG$: the traffic light part is a single vertex, with an input AND handle coming from a controller component (here unspecified) able to signal a fault; each car is a single component, with the input handles that signal whether it can go or must stop. Consistency of the overall program based on the assumption of consistency of the single components can be checked on the $CompCG$ exactly like it is done on the CG, i.e., on the handles connecting components.

Components can also be understood as agents, that can in principle be located on different nodes of a network. The traffic-light agent is always consistent, unless it receives a fact $fault\_tl$ by the controller component. In this case, the traffic-light agent becomes inconsistent, and stops working. Otherwise, it has two answer sets, one of which can be randomly selected, thus giving the green light on one lane, and the red light one the other one.

Each car agent is always consistent, but: $go(Car, D)$ is false if the agent receives the red light $tl(r, D)$ on its lane by means of the incoming AND handle and it is instead true if the agent receives the green light $tl(r, D)$ by means of the incoming OR handle. In this case, the answer set is $go(Car, D)$.

These components/agents can be instantiated from the following non-grounded program:

— Component: The traffic light
$tl(C1, D1) \leftarrow$
$\quad color(C1), color(C2), C1 \neq C2, lane(D1),$
$\quad \textit{not}\ tl(C2, D1), \textit{not}\ fault\_tl(D1).$
$tl(C1, D1) \leftarrow$
$\quad color(C1), go\_color(C1), lane(D1), lane(D2),$
$\quad D1 \neq D2, \textit{not}\ tl(C1, D2), \textit{not}\ fault\_tl(D1).$
$tl(C1, D1) \leftarrow$
$\quad color(C1), color(C2), C1 \neq C2,$
$\quad lane(D1), lane(D2), D1 \neq D2, tl(C2, D2).$
— Component: a car
$go(Car, D) \leftarrow$
$\quad color(C), wrong\_color(C), lane(D),$
$\quad \textit{not}\ go(Car, D), \textit{not}\ tl(C, D).$
$go(Car, D) \leftarrow$
$\quad color(C), ok\_color(C), lane(D), tl(C, D).$

Then, the above program can be seen as a template for creating as many traffic lights and cars as one needs.

5 Concluding Remarks

We have proposed a methodology for developing answer set programs based on components, that can possibly be understood as agents. This in the framework of Answer Set Programming, where every form of true modularization seemed to be impossible. The conceptual tool is the $CompCG$, based on the CG. We have given just a sketch of the proposed methodology. In more detail, each component should be defined together with an *interface*, establishing the requirements on the incoming handles, and the *promises* for the values of the outcoming handles. On the $CompCG$, the combinations of components can be checked for consistency by means of the formal tools outlined in the previous Sections. Combinations of components can be taken as new components, thus defining higher-level graphs. A formal definition of the methodology and experiments on real applications are the main future directions of this work.

References

[Cos03] Costantini, S., 1995. *On the existence of stable models of unstratified programs,* submitted, 2003 (draft available at the URL http://costantini.di.univaq.it).

[CosPro03] Costantini, S., and Provetti A., 2002. *Normal Forms for Answer Set Programming.* submitted, 2003 (draft available at the URL http://costantini.di.univaq.it).

[GelLif88] Gelfond, M. and Lifschitz, V., 1988. *The Stable Model Semantics for Logic Programming,* In: R. Kowalski and K. Bowen (eds.) Logic Programming: Proc. of 5th International Conference and Symposium: 1070–1080.

[GelLif91] Gelfond, M. and Lifschitz, V., 1991. *Classical Negation in Logic Programming and Disjunctive Databases,* New Generation Computing 9, 1991: 365–385.

[Lif99] Lifschitz V., 1999. *Answer Set Planning.* in: D. De Schreye (ed.) Proc. of the 1999 International Conference on Logic Programming (invited talk), The MIT Press: 23–37.

[MarTru99] Marek, W., and Truszczyński, M., 1999. *Stable Models and an Alternative Logic Programming Paradigm,* In: The Logic Programming Paradigm: a 25-Year Perspective, Springer-Verlag: 375–398.

[Systems] CCALC: *http://www.cs.utexas.edu/users/mcain/cc*
DeReS: *http://www.cs.engr.uky.edu/ lpnmr/DeReS.html*
DLV: *http://www.dbai.tuwien.ac.at/proj/dlv/*
SMODELS: *http://www.tcs.hut.fi/Software/smodels/*