

Designing Agent Chips

Insu Song
School of Information Technology & Electrical
Engineering
The University of Queensland
Brisbane, QLD, 4072, Australia
insu@itee.uq.edu.au

Guido Governatori
School of Information Technology & Electrical
Engineering
The University of Queensland
Brisbane, QLD, 4072, Australia
guido@itee.uq.edu.au

ABSTRACT

We outline meta-encoding schemas for compiling nonmonotonic logic theories into Verilog HDL (Hardware Description Language) descriptions. These descriptions can be synthesized into gate level specifications for direct fabrication of silicon chips¹. The method is applied for designing agent chips incorporating similar features found in the BDI (Belief, Desire, and Intention) and Brooks' subsumption architectures.

Categories and Subject Descriptors

I.2 [Computing Methodologies]: Artificial Intelligence

General Terms

Agent architecture, Agent Chips

Keywords

agent programming languages

1. INTRODUCTION

We develop a formal declarative specification language called Layered Non-monotonic Logic (LNL) and meta-encoding schemas to generate two types of executable specifications (general logic programs and synthesizable Verilog descriptions). The reasoning mechanism of each layer is an argumentation system and lower layers are subsumed by higher layers. The semantics of the language is based on the ambiguity blocking Dung-like argumentation system [2]. The language can represent relative certainties among the components of a system and provides behavior based decomposition of a system similarly to the subsumption architecture [1]. The subsumption architecture decomposes a system into (possibly prioritized) parallel behaviors rather than

¹Refers to large scale integrated circuits such as ASICs (Application Specific Integrated Circuits) and FPGAs (Field Programmable Gate Arrays).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...\$5.00.

© ACM 2006.

This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, May 8–12 2006, Hakodate, Hokkaido, Japan. ISBN: 1-59593-303-4. www.acm.org

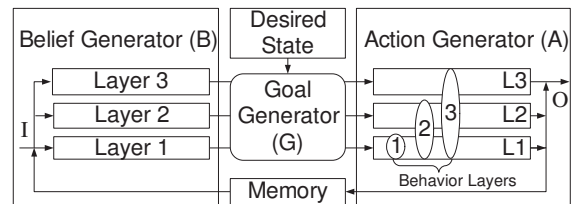


Figure 1: Conceptual view of an agent block.

the standard functional decomposition. Benefits of this approach in designing control systems are discussed in [1]. For instance, a typical functional decomposition of a cleaning robot might resemble the sequence:

perception→planning→task selection→motor control.

The decomposition of the same system in terms of behaviors would yield the following set of parallel behaviors:

1:avoid objects < 2:avoid water < 3:clean < 4:wander

where the numbers denote the layers and < denotes increasing levels of competence (i.e., more specific tasks). However, the layers of LNL represent relative confidences between layers such that layer- n is more confident than layer- $(n + 1)$. Thus, each behavior will be represented by one or more layers of LNL as shown in Figure 1. The figure also shows that less competent behaviors tend to be related with more confident LNL layers. The reason is because less competent behaviors are usually more reactive or urgent behaviors. LNL supports the behavior based decomposition and conceptual decomposition by *conclusion subsumption* and *rule-subsumption* which will be noted when we give the formal definition of LNL.

2. AGENT BLOCK

We briefly discuss an agent architecture called *Agent Block* to which the specification language has been applied. Each agent block is an autonomous system that performs actions in order to achieve a certain desired state. As we can see in Figure 1, an agent block receives signals from its input (I), and reasons about the current state of the world with its background knowledge in the belief generator (B). The background knowledge is decomposed into several layers. The goal generator (G) in the figure then decides what goals to be achieved in order to make the world conform to the desired state. The set of goals then instantiates a particular behavioral specification from a set of plans in the action gen-

erator (A) and produces appropriate actions for the output (O).

Given a desired state (or a set of desired states), the goal generator can be automatically created from the plans in A and the background knowledge in B . In a very simple case, to maintain a desired state d , whenever d is not true, we can activate a set of goals defined in A that are known to be related to achieving d . Suppose the relations between each goal of A and d are described in the background knowledge B . Then, in this case G consists of rules ($\neg d \Rightarrow s$) for each goal $s \in S$ where S is the minimal subset of goals defined in A such that $S \cup B$ entails d .

An agent block specification consists of descriptions of its input, output, memory, desired state, belief generator, goal generator, and action generator. In addition, connection details between these components need to be given for a complete specification of an agent block. However, in this paper, we only outline how to specify and compile the three main parts of an agent block: belief generator, goal generator, and action generator.

Example 1. Let us consider an example specification for the part that controls a vacuum cleaning unit of the cleaning robot. Suppose that the robot performs vacuuming action ($aVac$) if it detects (on sensor sA) that the area is dirty, but it stops the action if it detects some water on sensor sB . That is, we have two parallel behaviors: avoiding water and cleaning. However, avoiding water has higher priority than cleaning. This specification can be represented as a set of defeasible rules decomposed in to two layers as follows (each number of a rule denotes the confidence level of the rule):

$$\begin{aligned} B &= \{sA \Rightarrow_1 \text{dirty}, sB \Rightarrow_1 \text{wet}\} \\ G &= \{\text{wet} \Rightarrow_1 \neg \text{gclean}, \text{dirty} \Rightarrow_2 \text{gclean}\} \\ A &= \{\text{gclean} \Rightarrow_2 aVac\} \end{aligned}$$

B , G , and A represent the belief, goal, and action generator specifications, respectively. The arrows represent defeasible inferences. For instance, $sA \Rightarrow_1 \text{dirty}$ is read as ‘if sA is true, then it is usually dirty’. The numbers represent relative confidences between layers such that layer- n conclusions are more confident than layer- $(n+1)$ conclusions. Then, if an area is both dirty and wet, the vacuuming unit will be turned off: $aVac$ is not true. The reason is that since wet is a layer-1 conclusion in B , $\neg \text{gclean}$ is a layer-1 conclusion in G . As layer-1 conclusions are more confident than layer-2 conclusions, $\neg \text{gclean}$ is also a layer-2 conclusion overriding gclean in G . Thus, in A we cannot conclude $aVac$.

3. LAYERED NON-MONOTONIC LOGIC

We now give a formal definition of the language and its proof theory. From now on, we use *level- n* to mean the degree of confidence of layer- n . A rule r consists of its antecedent (or body) $A(r)$ which is a finite set of literals, an arrow \Rightarrow_n (n is a positive integer denoting a level- n rule), and its consequence (or head) $C(r)$ which is a literal. Level- n rules are more confident than level- $(n+1)$ rules. If l is a literal, $\sim l$ is its complement. An LNL theory is a tuple $T = (R, N)$ where $R = R_1 \cup \dots \cup R_n \cup \dots \cup R_N$ is a finite set of rules where R_n is a set of level- n rules and N is the number of layers. Then, $\mathbb{R}_n = R_1 \cup \dots \cup R_n$ is the set of layer- n rules. That is, layer- n subsumes (includes) layer- $(n-1)$ rules: *rule-subsumption*.

We obtain the meta-program representation of an LNL

theory from the meta-program formalization of Defeasible Logic (DL) given in [3] by removing defeaters, strict rules, priority relations, and converting the relationship between strict rules and defeasible rules in DL in to the relationship between layer- $(n-1)$ and layer- n in LNL. Thus, it is also an ambiguity blocking Dung-like argumentation system [2]. The details of the meta-program representation of an LNL theory $T = (R, N)$ and its layers are now described. First, we obtain the *conclusion-meta-program* $\Pi_C(T)$ consisting of the following general logic programs of each layer- n where $1 \leq n \leq N$:

- C1.** $\text{conclusion}_n(x) :- \text{conclusion}_{n-1}(x).$
- C2.** $\text{conclusion}_n(x) :- \text{supported}_n(x), \text{not supported}(\sim x), \text{not conclusion}_{n-1}(\sim x).$

where **not** denotes the negation as failure, \sim maps a literal x to its complement, and $\text{conclusion}_n(\mathbf{q})$ means that \mathbf{q} is a layer- n conclusion. C1 represents *conclusion subsumption* because layer- n includes layer- $(n-1)$ conclusions. Next, we obtain the *rule-meta-program* $\Pi_R(T)$ consisting of the following general logic programs for each rule $(q_1, \dots, q_m \Rightarrow_i p) \in \mathbb{R}_n$ of each layer- n :

- R1.** $\text{supported}_n(p) :- \text{conclusion}_n(q_1), \dots, \text{conclusion}_n(q_m).$

Then, the corresponding general logic program of T is

$$\Pi(T) = \Pi_C(T) \cup \Pi_R(T).$$

The meta-program counter part of the action generator of Example 1 is shown below:

$$\begin{aligned} \text{conclusion}_2(x) &:- \text{conclusion}_1(x). \\ \text{conclusion}_1(x) &:- \text{supported}_1(x), \text{not supported}_1(\sim x). \\ \text{conclusion}_2(x) &:- \text{supported}_2(x), \text{not supported}_2(\sim x), \\ &\quad \text{not conclusion}_1(\sim x). \\ \text{supported}_2(aVac) &:- \text{conclusion}_2(\text{gclean}). \end{aligned}$$

4. LOGIC PROGRAM REPRESENTATION

We now formally define the meta-encoding schema that translates LNL theories into compact propositional general logic programs. First, we define two literal encoding functions that encode literals in an LNL theory to previously unused positive literals. Let q be a literal and n a positive integer. Then, these functions are defined below:

$$\begin{aligned} \text{Supp}(q, n) &= \begin{cases} p_n^{+s} & \text{if } q \text{ is a positive literal } p. \\ p_n^{-s} & \text{if } q \text{ is a negative literal } \neg p. \end{cases} \\ \text{Con}(q, n) &= \begin{cases} p_n^+ & \text{if } q \text{ is a positive literal } p. \\ p_n^- & \text{if } q \text{ is a negative literal } \neg p. \end{cases} \end{aligned}$$

$\text{Supp}(q, n)$ denotes a support of q at layer- n and $\text{Con}(q, n)$ denotes a conclusion q at layer- n : $\text{Supp}(q, n)$ corresponds to $\text{supported}_n(\mathbf{q})$; $\text{Con}(q, n)$ corresponds to $\text{conclusion}_n(\mathbf{q})$. Let $\text{Con}A(A, n)$ be a set of new positive literals obtained from a set A of literals by replacing each literal $q \in A$ by $\text{Con}(q, n)$: $\text{Con}A(A, n) = \{\text{Con}(q, n) | q \in A\}$. With these functions, we now define the meta-encoding schema.

Let $T = (R, N)$ be an LNL theory and $\Pi(T)$ the corresponding meta-program. Let H_T be the Herbrand universe of $\Pi(T)$ which is the set of all possible grounded terms in $\Pi(T)$. Then, the Herbrand base $H_T(T)$ of $\Pi(T)$ is obtained by applying all possible consistent substitutions of

ground terms in H_T with variables in $\Pi(T)$. Thus, the meta-encoded Herbrand base (denoted as $P(T)$) of $\Pi(T)$ is obtained according to the following guidelines for each layer- n ($1 \leq n \leq N$):

P1: For each $q \in H_T$, add $Con(q, n) \leftarrow Con(q, n-1)$.

P2: For each $q \in H_T$, add

$Con(q, n) \leftarrow Supp(q, n), not Supp(\sim q, n), not Con(\sim q, n-1)$

P3: For each $r \in \mathbb{R}_n$, add $Supp(C(r), n) \leftarrow ConA(A(r), n)$.

For most of LNL theories, this direct translation of T results in a lot of redundant rules that will be never used for generating conclusions. For instance, we can safely remove the rules produced by P2 that will be never supported. Thus, we replace ‘For each $q \in H_T$ ’ with ‘For each $q \in SL_n$ ’ in P2 where SL_n is the set of all layer- n literals that might be supported in $P(T)$. SL_n can be obtained as follows:

$$SL_n = \{C(r) | r \in \mathbb{R}_n\}$$

Let $P2(T)$ be the set of rules introduced by P2 in $P(T)$. Then, we can also reduce the number rules in P1 by replacing ‘For each $q \in H_T$ ’ with ‘For each $q \in CL_n$ ’ where CL_n is the set of all conclusion literals in layer- n of T . This is obtained as follows:

$$CL_0 = \emptyset$$

$$CL_n = CL_{n-1} \cup \{q | Con(q, n-1) \in P2(T)\}$$

Let us consider an LAS theory $T = (R, N)$ containing x unique rules in each layer. The total number of rules is $X = xN$. Then, the number of rules and facts created by the guidelines is bounded by the following equation:

$$|P(T)| \leq X(0.5 + 1.5N)$$

That is, the size of $P(T)$ is linear to the size of T . Particularly, if $N = 1$, $|P(T)| \leq 2|T|$.

The general logic program corresponding to the action generator of Example 1 is shown below:

$$\begin{aligned} aVac_2^+ :- aVac_2^{+s}, not aVac_2^{-s}, not aVac_1^- \\ aVac_2^{+s} :- gclean_2^+ \end{aligned}$$

5. VERILOG HDL DESCRIPTION

Verilog is one of widely used HDL languages. An HDL is a language for formal description of electronic circuits. Most of VLSI (Very-large-scale integrated) circuit design tools can import Verilog files to generate a netlist which can be directly used to fabricate ICs. A netlist is a list of logic gates and their interconnections which make up a circuit.

The meta-encoding of an LNL theory into a Verilog module is similar to the definition of $P(T)$ which is given in the previous section. The differences are that all the used literals (referred as ‘signals’ in Verilog) in rules must be declared separately and signals have binary values: 1 and 0. We use value 1 to represent that a literal (a signal) is ‘true’ and 0 to represent ‘unknown’. All defined signals are assumed to have value 0 by default but can be changed to value 1 if it is derived by a logic gate with the output value of 1. The exact implementation of this property is device specific, and thus it will be ignored in this paper.

Let $T = (R, N)$ be an LNL theory. The corresponding Verilog description $V(T)$ is obtained from T by adding the Verilog statements according to the following guidelines for

each layer- n ($1 \leq n \leq N$):

V1: For each $q \in CL_n$, add
wor $Con(q, n)$;
assign $Con(q, n) = Con(q, n-1)$;

V2: For each $q \in SL_n$, add
wor $Supp(q, n), Supp(\sim q, n), Con(q, n)$;
assign $Con(q, n) = Supp(q, n) \wedge$
 $\sim Supp(\sim q, n) \wedge \sim Con(\sim q, n-1)$;

V3: For each $r \in \mathbb{R}_n$, add
wor $Supp(C(r), n), Supp(\sim C(r), n), Con(C(r), n)$;
assign $Supp(C(r), n) = \bigwedge_{q \in A(r)} Con(C(r), n)$;

The Verilog module description of the action generator of Example 1 is shown below:

```
module A (input gclean2+, output aVac2+);
  wor aVac2+, aVac2+s, aVac2-s, aVac1-, gclean2+;
  assign aVac2+ = aVac2+s & ~aVac2-s & ~aVac1-;
  assign aVac2+s = gclean2+;
end module
```

In addition to the above description, we also need to add some device specific codes to set the default values (0) of the signals and port mapping codes between the components (B , A , G , and the memory) of the agent block. A generated Verilog description can be directly compiled to a netlist using a freely available synthesizer such as XilinxTM ISE Webpack. In the module definition, both $aVac_2^-$ and $aVac_1^-$ are not necessary as they will never be true. Moreover, in practice many of subsumed rules are redundant because only few of the literals in layer- $(n-1)$ are referred in layer- n . Thus, if such redundant codes are removed, the actual size of a Verilog description of an LNL theory can be made close to the size of the original theory independent of the number of layers.

6. CONCLUSION

We have devised a formal translation method which compiles high level (non-monotonic logic based) specifications of an agent into synthesizable Verilog HDL descriptions. In contrast to current agent systems, an agent chip is realized as a concurrent computational structure on a silicon chip. This makes agent chips very robust and reactive. Since the specification language is a consistent instantiation of an argumentation system, it also has a well defined semantics which is critical for system verification and maintenance. Thus, agent chips can be readily used for mission critical applications such as distributed plant controllers, mobile robots, and patient caring systems.

7. REFERENCES

- [1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
- [2] G. Governatori, M. J. Maher, G. Antoniou, and D. Billington. Argumentation semantics for defeasible logics. *Journal of Logic and Computation*, 14(5):675–702, 2004.
- [3] M. J. Maher and G. Governatori. A semantic decomposition of defeasible logics. In *AAAI '99*, pages 299–305, 1999.