



# Model Generation in Description Logics: What Can We Learn From Software Engineering?

Miguel Garcia, Alissa Kaplunova, and Ralf Möller

16th August 2007

## Abstract

Model checking as well as model generation (aka model finding) are well-established methodologies for formally verifying properties of possibly time-evolving systems. In order to support the ontology development process in an incremental way, our thesis is that well-known model-generation tools can be adopted accordingly and provide major benefits for human ontology designers. In this work we evaluate pros and cons of applying an existing model checking and generation tool in this context.

# 1 Introduction

Model checking as well as model generation (aka model finding) are well-established methodologies for formally verifying properties of possibly time-evolving systems. A recent survey can be found in [2]. Usually, some aspects of real-world systems have to be abstracted away in order to make them accessible to formal logical modeling: continuous vs. discrete behavior, granularity, stochasticity, etc. Nevertheless, model-checking tools are successfully applied in practice. Indeed, improvements in the underlying decision procedures (most notably SAT and BDDs) together with higher-level specification languages have broadened the applicability of these techniques. New application fields have been identified recently. One such field comprises solving selected problems arising in ontology management and evolution as a complement to dedicated DL engines.

The *model generation problem* is postulated as follows. Given an ontology  $O$ , which is a pair  $(\mathcal{T}, \mathcal{A})$  of a TBox  $\mathcal{T}$  and an ABox  $\mathcal{A}$ , find an interpretation  $\mathcal{I}$  which satisfies all axioms of  $\mathcal{T}$  and  $\mathcal{A}$ . In case of a *model checking problem* the goal is to prove whether a given interpretation is a model.

In order to support the ontology development process in an incremental way, our thesis is that well-known model-generation tools can be adopted accordingly and provide major benefits for human ontology designers. In this work we evaluate pros and cons of applying an existing model checking and generation tool in this context.

In fact, the ontology designer is often not interested in just testing the satisfiability of an ontology by checking whether one single model exists, but possibly wants to inspect a number of generated models instead. This way, unintended models might be identified. This kind of modeling methodology has been proven to be very effective in software engineering (e.g., [4]). The ontology designer should be offered a possibility to adjust the ontology by examining automatically generated relational model structures. Model generators support this process quite well whereas for checking the satisfiability of ontologies and computing the taxonomy tableau algorithms have been proven to be very effective. Thus, it seems attractive to augment tableau provers to also support model generation. Current tableau algorithms are not well applicable as model generation procedures since they only return (a description of) a so-called single canonical model. Instead, model finders are able to enumerate all models systematically. This can indeed be useful for ontology design tasks.

To illustrate this, we discuss the following simple example. Let  $A, B$  be concepts and  $R$  be a role. Suppose the satisfiability of the following concept is checked by both a DL system and a model finder:  $(\exists R.A) \sqcap (\exists R.B)$ . The model generated by a tableau algorithm is shown on the left-hand side of the vertical bar in Figure 1. However, the ontology designer may be more interested in inspecting models computed by a model finder (see Figure 1, to the right of the vertical bar). The latter four models are not considered by the rules of tableau prover because if the left structure is model, then the structures to the right of the bar are also models. Thus, it suffices to consider only one model (the one to the left of the bar) in order to show satisfiability. In order to evaluate an ontology (i.e., the concepts, roles, and axioms in it), considering all models is nevertheless interesting as we argued above. Thus, it makes sense to investigate contemporary model finders, study the state of the art in this field from a practical point of view, and identify possible limitations.

In our case study, we adopt a particular finite model finder, namely Alloy Analyzer 3.0 [4], whose language is based on relational calculus and thus allows for straightforward

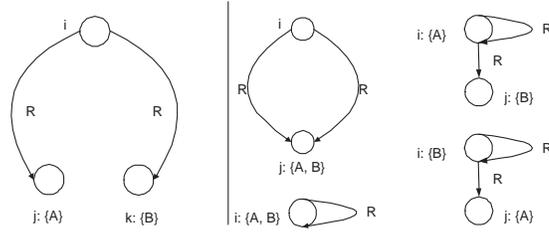


Figure 1: Models of  $(\exists R.A) \wedge (\exists R.B)$

representation of  $\mathcal{ALC}$  knowledge bases. The technique reported here is not the first attempt at rephrasing ontology languages in Alloy, yet it has been developed independently. In [7], case studies have been published where an ontology has been formulated (and further analyzed) in Alloy, Z, and (recently) HOL. Unlike our approach to capture the semantics of  $\mathcal{ALC}$  constructs directly, the authors define a translation schema that considers the meta-level of the ontology language in terms of individuals for concepts and properties as well as relationships among these individuals. We show results achieved so far for several case studies:

- Model inspection (Sect. 3.1);
- Visual display of counterexamples for a subsumption assumption (Sect. 3.2) and for a concept equivalence assumption (Sect. 3.3);
- Finding models for  $\mathcal{ALC}$  terms whose satisfiability analysis is expensive for contemporary tableaux-based reasoners (Sect. 4).

In the next section we address the translation rules for DL into Alloy, starting with the base logic  $\mathcal{ALC}$ .

## 2 Translation from Description Logics into Alloy

The logic underlying the Alloy Analyzer is Relational Logic (RL) whose syntax, type rules and semantics are described in [3]. This logic is more than a syntactical variation of first-order logic, because it includes transitive closure. An automatic model finder requires the specification of a *scope*, a bound on the number of atoms in the universe (cardinalities of concepts). This limitation is not as dramatic as it might seem, given the so-called *small-scope hypothesis*:

“First-order logic is undecidable, so our analysis cannot be a decision procedure: if no model is found, the formula may still have a model in a larger scope. Nevertheless, the analysis is useful, since many formulas that have models have small ones. [...]

Given a relational formula, we can construct a boolean formula that has a model exactly when the original formula has a model in some given scope.” [3]

Given that  $\mathcal{ALC}$  exhibits the finite model property, it is thus amenable to circumvent the finite-scope limitation. In fact, we can compute worst case concept cardinalities according to the maximum concept branching factor and the maximum depth of nested existential quantifiers.

## 2.1 Translation Rules for $\mathcal{ALC}$

**Definition 1 (Alloy Translation Rules for  $\mathcal{ALC}$  Concepts)** If  $A$  is a concept name,  $C$ ,  $D$  are concepts,  $R$  is a role name, the following translation rules can be applied to  $\mathcal{ALC}$  concepts in order to obtain semantically equivalent Alloy formulas:

$A$	$A$
$C \sqcap D$	$C \ \& \ D$
$C \sqcup D$	$C \ + \ D$
$\neg C$	$\text{univ} - C$
$\forall R.C$	$\text{univ} - (R.(\text{univ} - C))$
$\exists R.C$	$R.C$

Here,  $A$ ,  $C$ ,  $D$ ,  $R$  denote Alloy relations,  $\&$ ,  $+$ ,  $-$  are set operators (intersection, union and difference, respectively), “.” stands for the relational join operator. The unary relation  $\text{univ}$  represents the set containing every instance of the universe (interpretation domain).

**Definition 2 (Alloy Translation Rules for  $\mathcal{ALC}$  TBox and ABox axioms)** We summarize translation rules for  $\mathcal{ALC}$  terminological and assertional axioms into Alloy.

- In  $\mathcal{ALC}$ , expressions  $\top$  (universal concept) and  $\perp$  (unsatisfiable concept) are used as abbreviations for  $A \sqcup \neg A$  resp.  $A \sqcap \neg A$ , where  $A$  is a concept name. In Alloy, we define the TOP relation as subset of  $\text{univ}$ : `abstract sig TOP`, and BOTTOM as subset of TOP containing no instances: `sig BOTTOM in TOP {} fact { #BOTTOM = 0 }`.

A signature (denoted as `sig`) introduces a set of atoms. The declaration `sig A { }` introduces a set named  $A$ . An `abstract` signature has no elements except those belonging to the extension of its subsignatures.

- Elementary descriptions are atomic concepts and atomic roles.

Atomic concepts are declared in Alloy as non-empty subsets of TOP. For example,  $A$  is declared as atomic concept: `sig A in TOP {} fact { }`.

Atomic roles are specified in Alloy with a set TOP as both domain and range. For example, the role `hasChild` is an atomic role: `abstract sig TOP { hasChild : set TOP }`.

- If  $C$  is an atomic concept and  $D$  is a concept, then  $C \sqsubseteq D$  is called generalized concept inclusion, or GCI. GCIs are translated into Alloy using the set operator `in` (subset): `fact { C in D }`.
- Concept definitions of the form  $A \equiv C$ , where  $A$  is an atomic concept, are called equality axioms. Equalities are translated using Alloy’s set equality operator `=`: `fact { A = C }`.
- Instances of a given concept description are called individuals. If  $i$  is an individual, then it can be defined in Alloy as follows: `sig i in TOP {} fact { #i = 1 }`.

In Alloy, a multiplicity keyword placed before a signature declaration constrains the number of elements in the signature’s set. E.g., the keyword `one` allows for defining a signature whose set contains exactly one element. Thus, `one sig i in TOP {}` declares instance  $i$ , having the same effect as the specification above. To implement the unique name assumption, additional constraints are generated to ensure that these singleton sets are pairwise different, for example: `fact { no ( polyneikes & iokaste ) }`.

- If  $a, b$  are individual names,  $C$  is a concept and  $R$  is a role name, than the following assertions about named individuals can be built by using constructs above:

$C(a)$  (concept membership assertion),  $R(a, b)$  (role membership assertions)

In terms of Alloy we define concept membership assertions as `fact { a in C }` and role membership assertions as `fact { a -> b in R }` ( $->$  denotes the relational product operator).

## 2.2 Translation of *SHIQ* and *SROIQ*

Alloy's underlying logic is expressive enough to encode *SHIQ* or even *SROIQ* formulas. As an outlook, tables below depict the Alloy formulation of *SHIQ* and *SROIQ* concepts and role constructors as well as of additional role constructs possible in Alloy. Here,  $:>$  denotes the range restriction and  $\sim$  is the relational transpose operator defined over binary relations. The operator  $\#$  applied to a relation gives the cardinality of the relation as an integer value. The binary relation `iden` relates all the instances of the universe to themselves.

<i>SHIQ</i> concepts	Alloy translation
$\leq nR.C$	<code>{ a : univ   #(a.(R :&gt; C)) =&lt; n }</code>
$\geq nR.C$	<code>{ a : univ   #(a.(R :&gt; C)) =&gt; n }</code>
inverse	<code>~R</code>
<i>SROIQ</i> concepts	Alloy translation
$\{o\}$	<code>sig i in TOP { } fact { #o = 1 }</code>
$\exists R.Self$	<code>(R &amp; iden).univ</code>
Further role terms	Alloy translation
$R \sqcap S$	<code>R &amp; S</code>
reflexive transitive closure	<code>*R</code>

*SHIQ* allows for defining role hierarchies, which is a finite set of role inclusion axioms  $R \sqsubseteq S$  where  $R$  and  $S$  are roles, and transitive roles ( $R \circ R \sqsubseteq R$ ). In Alloy, we achieve the same expressibility using the set operator `in` and the relational composition (join) operator `.: R in S` and `(R . R) in R`.

In *SROIQ*, a role inclusion axiom is of the form  $w \sqsubseteq R$ , where  $w$  is a finite string of roles (e.g.,  $S1, S2$ ) and  $R$  is a role name. The appropriate translation into Alloy is: `(S1 . S2) in R`.

## 3 Case Studies

In follows, we illustrate advantages of our proposal in the context of ontology design by discussing several case studies.

### 3.1 Model Inspection by Counterexample Extraction

As an introductory example of model inspection, we use the Oedipus example (see [1, p. 73]). In this example, the following ABox with some facts about the Oedipus story is supposed:

$$\begin{array}{ll}
 \text{hasChild}(\text{iokaste}, \text{oedipus}) & \text{hasChild}(\text{iokaste}, \text{polymeikes}) \\
 \text{hasChild}(\text{oedipus}, \text{polymeikes}) & \text{hasChild}(\text{polymeikes}, \text{thersandros}) \\
 \text{Patricide}(\text{oedipus}) & \neg \text{Patricide}(\text{thersandros})
 \end{array}$$

Now, we want to find out whether some individual exists that have a child that is a patricide and that itself has a child that is not a patricide. This can be seen as a problem of checking the satisfiability of the concept

$hasPatricideChildWithNonPatricideChild \equiv (\exists hasChild. (Patricide \sqcap \exists hasChild. \neg Patricide))$

Applying  $\mathcal{ALC}$  translation rules to the Oedipus knowledge base, we obtain the following Alloy specification:

```

module oedipus
abstract sig TOP {
/* atomic roles */
    hasChild : set TOP }
sig BOTTOM in TOP {}
fact { #BOTTOM = 0 }
/* atomic concepts */
sig hasPatricideChildWithNonPatricideChild in TOP {}
sig Patricide in TOP {}
/* individuals */
one sig polyneikes in TOP {}
one sig iokaste in TOP {}
one sig thersandros in TOP {}
one sig oedipus in TOP {}
/* pairwise disjointness of individuals */
fact { no ( polyneikes & iokaste ) }
fact { no ( polyneikes & thersandros ) }
fact { no ( polyneikes & oedipus ) }
fact { no ( iokaste & thersandros ) }
fact { no ( iokaste & oedipus ) }
fact { no ( thersandros & oedipus ) }
/* equality axioms */
fact { (hasPatricideChildWithNonPatricideChild) =
    ( (hasChild).((Patricide)&((hasChild).((univ - Patricide)))) ) }
/* concept assertions */
fact { oedipus in Patricide }
fact { thersandros in ( univ - Patricide ) }
/* role assertions */
fact { oedipus -> polyneikes in hasChild }
fact { iokaste -> polyneikes in hasChild }
fact { polyneikes -> thersandros in hasChild }
fact { iokaste -> oedipus in hasChild }
pred show() { #univ = 4 }
run show for 4

```

Alloy presents the following model (Figure 2). We will discuss next how it relates to the ontology given above. In summary, Iokaste is shown to have a patricide child (Oedipus in this model) who in turn has a non patricide child (a choice of individuals in this model).

Alloy offers a choice of customization capabilities for visualizing models but we will stick with the default settings. Particular models can be shown graphically as “snapshots” where individuals are represented as rectangles. Each such rectangle is identified by the internal name used by Alloy (e.g., “TOP1”) which appears on the upper part of the rectangle. Arcs between rectangles stand for binary relations (roles), a label on the arc makes clear which role is being referred to. For each individual, the sets (concepts) it belongs to are shown as a comma separated list of labels on the lower part of the rectangle in question. Absence of one

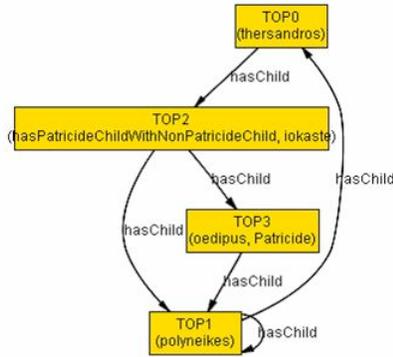


Figure 2: Oedipus example

such labels means that the individual does not satisfy that concept. For example, the labels of the node “TOP2” “iokaste, hasPatricideChildWithNonPatricideChild” reveal that the individual *iokaste* is described by the concept `hasPatricideChildWithNonPatricideChild`. Browsing further models will show other constellations under which this concept is satisfied. Note however that in this particular model, *polyneikes* is considered to have himself as child (nothing in the TBox prevents this). Inspecting models may give rise for adjusting the ontology by adding further axioms.

### 3.2 Counterexamples for a Subsumption Assumption

We use the following example from [1, p. 82] to demonstrate how a subsumption relation can be explained using Alloy. Assume that we want to check whether  $(\exists r.a) \sqcap (\exists r.b)$  is subsumed by  $\exists r.(a \sqcap b)$ . This is equivalent to the satisfiability test of the concept *ifUnsatisfiableThenSubsumes*  $\equiv (\exists r.a) \sqcap (\exists r.b) \sqcap \neg(\exists r.(a \sqcap b))$ .

Letting Alloy analyze the predicate *ifUnsatisfiableThenSubsumes* results in several models. If left to its own devices, Alloy presents a model that minimizes the number of individuals. Alloy can be instructed however to look for models of a certain shape. We will do just that in order to display the solution presented in [1, p. 82], which is computed by a tableau algorithm. In order to achieve this, we will constraint those model we are interested in to those having exactly three individuals, with no individual in *a* nor *b* having a role filler over *r*. The model we are looking for is depicted in Figure 3. The technique described above is, of course, generally applicable and results in shorter response times as only a subset of all possible models is explored.

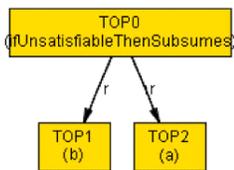


Figure 3: Model of the concept *ifUnsatisfiableThenSubsumes*



Figure 4: Model of the concept *counterExample*

To gain a better understanding for this result, one must recall that labels of particular

individuals (nodes) contain concept names the individual belongs to. Absence of some concept name  $C$  in the label of an individual means that the individual belongs to the concept  $\neg C$ . Therefore, the node TOP1 explicitly has  $\mathbf{b}$  in its label and implicitly  $\neg \mathbf{a}$ . A similar explanation holds for the node TOP2.

### 3.3 Counterexamples for a Concept Equivalence Assumption

As a next test case we assume the following simple TBox:

$$\begin{aligned} \text{dogholder} &\equiv (\text{person} \sqcap (\geq 1 \text{ hasdog.dog})) \\ \text{houndholder} &\equiv (\text{person} \sqcap (\geq 1 \text{ hashound.hound})) \\ \text{dog} &\equiv \text{hound}, \text{ hashound} \sqsubseteq \text{hasdog} \end{aligned}$$

Suppose we expect that concepts *dogholder* and *houndholder* must be equivalent. In order to check this we let Alloy Analyzer generate models of the concept  $\text{counterExample} \equiv (\text{dogholder} \sqcap \neg \text{houndholder}) \sqcup (\neg \text{dogholder} \sqcap \text{houndholder})$ .

One model found by Alloy as a counterexample is shown in Figure 4. In this model, the individual named TOP is found that belongs to the concept *dogholder* but not to the concept *houndholder*. The reason is that the roles *hasdog* and *hashound* are not equivalent.

## 4 Evaluation of Practical Usefulness

In order to empirically study the performance of model generation with Alloy we have chosen a benchmark originally proposed for comparing DL systems (DL-98 systems comparison). We consider k-branch, which evaluates satisfiability-testing performance for large concept expressions without reference to a TBox. Progressively larger expressions (from size 1 to size 21) are presented in two variants: all k-branch-p expressions are unsatisfiable while all k-branch-n expressions are satisfiable. These (and other) benchmarks are available at <http://dl.kr.org/dl98/comparison/data.html>. We also used RacerPro 1.9.1 beta to measure the times for (un)satisfiability checking with a tableau prover.

Summing up, Alloy exhibits a competitive performance for satisfiable input concepts if models can be found with up to 10 individuals. If models have more than 10 objects, performance quickly degrades (in particular, if unsatisfiable concepts are used as input). Apparently, BDD optimizations used in these systems cannot cope with combinatorial explosion, as more models are explored by Alloy than by tableau-based algorithms. Thus, there is good news when models are small enough (as full information can be presented to the ontology designer). If large model structures have to be explored, we found that model generators such as Alloy are not applicable.

As explained in the Alloy literature, the guiding principle for their construction was the “small scope hypothesis”, which k-branch does not exhibit. Had we chosen a benchmark where this is the case, the results would have been more favorable to Alloy. For comparison, the time spent by RacerPro in this problem for different problem sizes is also shown in Table 1 and Table 2. The results of Alloy’s runs are shown for different scope sizes (we did not yet implement an algorithm to compute the scope size according to the maximum concept branching factor and maximum depth of nested existential quantifiers as mentioned in Sect. 2).

#	Alloy, 10 inds	Alloy, 15 inds	RacerPro
1	265	110	3
2	110	328	5
3	1,797, NMF	5,281	24
4	1,422, NMF	21,921, NMF	31
5	2,562, NMF	43,687, NMF	164
6	1,469, NMF	31,125, NMF	288
7	6,828, NMF	61,625, NMF	681
8	6,906, NMF	42,625, NMF	1,809
9	6,250, NMF	53,000, NMF	4,392
10	7,375, NMF	4,970,229, NMF	9,714
11	7437, NMF	3,024,688, NMF	23,623
12	17,50, NMF	575,407, NMF	51,266
13	27,640, NMF	4,215,123, NMF	119,628
14	4,281, NMF	3,654,211, NMF	294,519
15	38,577, NMF	1,282,483, NMF	765,325

Table 1: Concept satisfiability benchmarks (k-branch-n, all times in milliseconds, NMF = no model found, # = problem size)

#	Alloy, 10 inds	Alloy, 15 inds	RacerPro
1	47	94	1
2	1,532	531	2
3	875	44,624	4
4	1,281	34,421	5
5	2,610	30,953	11
6	9,828	56,422	24
7	5,781	63,935	29
8	1,984	41,578	218
9	6,578	70,466	113
10	58,718	716,200	225
11	12,500	520,077	638
12	30,484	345,288	711
13	6,500	409,849	1,099
14	10,624	811,636	3,517
15	11,719	3,129,982	4,143
16	5,066	845,979	11,742
17	7,219	1,204,383	24,594
18	94,466	1,401,839	31,498
19	8,141	660,968	63,331
20	15,090	2,096,752	187,332
21	135,829	563,153	197,030

Table 2: Concept unsatisfiability benchmarks (k-branch-p)

As we can conclude from the measurement results, modern highly-optimized tableau-based provers far outperform model finders such as Alloy. However, in order to improve the usefulness of tableau-based reasoners also for ontology design tasks, it may be a good idea to equip them with model generation capacities like those provided by model finders for identifying unintended models. In the other direction, namely for increasing performance of model finders, DL prover techniques might also be helpful (given the expressivity of the input formulas is in the DL fragment). A tableau prover could be used for satisfiability checking. If there exists a model, the tableau describes a canonical model, which could be further modified in order to derive all models in the sense of model finders. If the expressivity is too high, models might be infinite, however, so the details of this idea have to be investigated carefully.

## 5 Conclusion and Further Work

In this work, we have studied an applicability of finite model finders (by the example of Alloy Analyzer) for ontology design tasks. For this, we have proposed translation rules from DL ( $\mathcal{ALC}$  and some constructs of  $\mathcal{SHIQ}$  and  $\mathcal{SROIQ}$ ) into Alloy and illustrated the translation by several examples. We conducted experiments that demonstrate the benefit model finders but also indicate their weaknesses.

While originally addressing interactive systems, in particular communication protocols, model-checking techniques are now applied to general imperative algorithms, as exemplified by the  $^+$ CAL algorithm language [5]. Given that  $^+$ CAL allows specifying pre- and postconditions alongside imperative statements, it constitutes a viable mechanism for automatically

testing the correctness of a proposed algorithm. A setting where this can be seen at work is the validation of the optimized implementation of a decision procedure. Indeed, model checkers might also be successfully applied for developing robust and scalable optimized description logics systems. Unlike testing, checking a specification even for a finite set of individual using model checking techniques might dramatically reduce development efforts.

The crucial requirement for integrating model finders in practical applications like ontology development tools is the effectiveness of constraint-solving engines they are based on. One of the recent investigations in producing high-performance tools is a Kodkod, a prototype SAT-based model finder designed for a relational logic [6]. Besides of promising performance results, the system provides for further relevant features like optimized handling of assertional knowledge (in Alloy, specifying partial solutions is possible only in the form of additional constraints that increases the complexity of the solving process).

**Acknowledgements.** This work was partially supported by the EU funded project TONES (Thinking ONtologiES FET-FP6-7603).

## References

- [1] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
- [2] Michael Huth. Some current issues in model checking. *Software Tools for Technology Transfer*, 8(4):1–10, 2006.
- [3] Daniel Jackson. Automating first-order relational logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA, 2000. ACM Press.
- [4] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
- [5] L. Lamport. The +CAL algorithm language. Technical report, Microsoft, 2006.
- [6] E. Torlak and D. Jackson. The Design of a Relational Engine. Technical Report MIT-CSAIL-TR-2006-068, MIT CSAIL, 2006.
- [7] Hai Wang, Jin Song Dong, Jing Sun, and Jun Sun. Reasoning support for semantic web ontology family languages using alloy. *International Journal of Multiagent and Grid Systems, Special issue on Agent-Oriented Software Development Methodologies*, 2(4), December 2006.