Modeling information repositories consistency and auditing through Alloy and CCalc

Hector G. Ceballos, Ramon F. Brena and Francisco J. Cantu

Tecnologico de Monterrey
Monterrey, Mexico
{ceballos, ramon.brena, fcantu}@itesm.mx

Abstract. We present a case study on which a model checking language (Alloy) and a non-monotonic causal language (CCalc) are used for modeling the auditing process of an information repository through a MultiAgents System. Alloy and CCalc are used for identifying inconsistencies and simulating an automatic correction process carried out by intelligent agents supervised by human users. Similarities and strengths of both languages are pointed out.

Keywords: MultiAgent Systems, Model checking tools, Independent Choice Logic, Simulation

1 Introduction

For a University, the intellectual production of its professors is an important asset valued by accreditations, rankings and founding organisms [1]. Nevertheless, this information does not reside in a unique repository; rather than it is managed by multiple entities such as editorial companies, organizations and institutions. In most of the cases, each professor compiles and organizes his own publication record.

Maintaining an institutional repository updated and consistent is a permanent time-consuming activity that involves the participation of professors and auditors. This kind of tasks requires intensive application of knowledge in regular basis. The goal is not only performing routines or processes periodically, but translating daily operation in a source of knowledge and learning. Guidance of auditors is indispensable in this case for the correct design and application of policies and norms.

This paper presents the use of two formal languages for the description of the elements and interactions on the described domain: Alloy and CCalc. The former is a model checker used to describe in detail the objects on the domain and the rules that allow verifying the consistency of the information stored in the repository. The latter was used for modeling a MultiAgent System on charge of controlling the auditing process.

This paper is organized as follows. First it is presented the domain and the scenario. Next there are briefly introduced Alloy and CCalc. In section 4 it is described how these languages were used for modeling the domain. Finally we conclude comparing both languages in section 5.

Advances in Computer Science and Electronic Systems
Research in Computing Science 52, 2011, pp. 169-178

Received 23/01/11
Accepted 20/02/11
Final version 07/03/11

2 Information auditing in a University repository

The Tecnologico de Monterrey counts with an information system where information about publications is fed in an institutional repository by the professor itself. Human auditors are responsible for classifying and complementing the information [2]. Auditing is made asynchronously and the results are notified to the professor when changes affect his/her personal record. Additionally, after a publication is registered, its coauthors are notified and empowered to provide additional information or do corrections.

2.1 The Publications Repository

The publications repository registers the scientific production of professors organizing it in an institutional taxonomy. For instance, articles in journals, articles in proceedings and thesis are different types of publications. The information stored in the repository is actually the metadata of the publication; hence we have common data elements like authors, publication date, title, etc.

It is considered additional information and constraints for each type of publication. For example, a journal article is published by a journal, meanwhile that a proceedings article is published in the proceedings of a conference. It is important maintaining a differentiated catalogue of journals and conferences that allows not only quantifying but qualifying professors' scientific production.

There are some common inconsistencies that expert auditors have already detected and modeled. One of them is the duplicity of the publication in the repository and consists on the existence of two publications in the repository having such a degree of similarity that make the auditor suspect that both are in fact the same publication registered twice. This and other types of inconsistencies must be corrected off-line, on regular basis and after information modification.

2.2 The automated auditing process

We propose to implement a MultiAgents System as back-end platform on charge of monitoring, auditing and correcting information feed by professors. The architecture of the system is shown in Figure 1.

Expert auditors define and supervise inconsistency and correction rules. The *LogMonitor* Agent keeps track of actions performed by auditors and professors in the web information system that updates the repository. The *RepGuardian* Agent is responsible for instantiating service agents required for auditing the publications. *Auditor* agents evaluate auditing rules on demand; they are responsible for gathering the necessary information to audit the record and request a correction whenever there is one available and trustworthy. *Corrector* agents apply the correction rules. The *Notifier* Agent communicates the result of auditing to the human expert or professor responsible through a *User* agent.

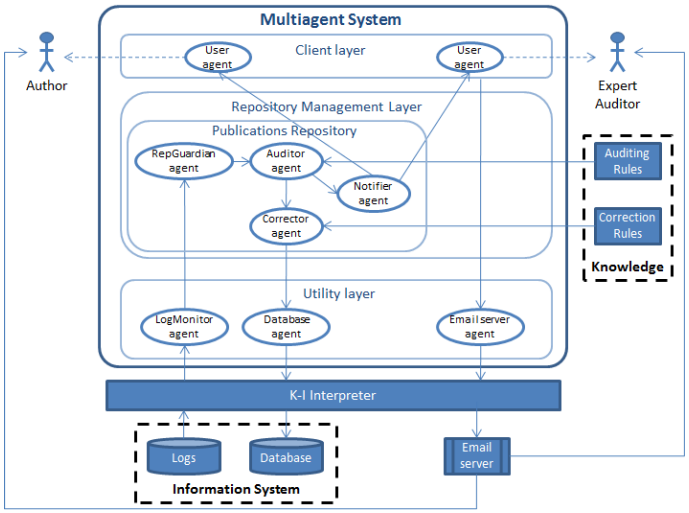

Fig. 1. MultiAgent system architecture for publications auditing.

3 Formal Languages

Current formal languages have started to incorporate modeling primitives such as classes and inheritance that eases the expression of a domain. These object oriented characteristics are exploited during the inference process, allowing generalizing the application of rules and limiting the branching factor. In this paper we explore two formal languages: Alloy and CCalc.

3.1 Alloy Analyzer

Alloy Analyzer [3] is a lightweight modeling language for software design. Its notation uses the sets and relations nomenclature. Alloy is considered a model checking tool because generates all the possible combinations in a certain scope and tries to identify counterexamples for the declared axioms in the model.

Every value in Alloy logic is a relation: a set is a unary relation, scalars are singleton sets, and predicates are n-ary relations. Rows are unordered, columns are ordered but unnamed and all relations are first-order. Alloy syntax define some set constants and operators for representing set operations. Additionally counts with Boolean operators, quantifiers that can be used to declare sets or for construct quantified expressions, and cardinality expressions.

Alloy has upper level constructors such as signatures, represented `sig`, which are used to represent sorts of similar things. Signatures have attributes and constraints associated. A signature can extend other signature, inheriting its attributes and constraints. Facts, represented `fact`, introduce constraints that are assumed to always hold. Functions, represented `fun`, are named expression with declaration parameters and a declaration expression as a result invoked by

providing an expression for each parameter. Predicates, represented `pred`, are named formula with declaration parameters.

Assertions, denoted `assert`, are constraints intended to follow from facts of the model. Assertions doesn't constraint the model, but are checked in the generated model with the command `check`. Check instructs the analyzer to search for counterexamples to a given assertion within a scope. The scope is expressed in quantities of individuals for all available signatures or for every signature individually. The `run` command can be used for instructing the analyzer to search for instances of a given predicate or a given function within a scope.

Alloy can be used for elaborating static and dynamic models. Static models describe a single state where properties are invariant. Facts constraint all possible relations between individuals of the defined signatures. Predicates are used to express definitions, i.e. a named pattern showed in a (set of) individual(s).

Dynamic models describe transitions between states meanwhile predicates are used for describing operations. One way of representing a dynamic model is defining the `state` signature as ordered which allows that each instance of this signature represent a state of the transition model. Predicates receiving two parameters representing contiguous states are used to indicate valid transitions in the model. These predicates are the equivalent of action predicates and hence on its definition must include preconditions and post-conditions; in the post-conditions it must be included frame properties, i.e. indications of the things that must remain unaltered. A special fact denominated `traces` is used to relate all the consecutive states or time frames indicating the valid transitions.

3.2 CCalc

The Causal Calculator (CCalc) is a system for representing commonsense knowledge about action and change. It implements a fragment of the causal logic C+ [4]. The semantics of the language of CCalc is related to default logic and logic programming. Computationally, CCalc uses ideas of satisfiability planning. CCalc runs over Prolog and connects to a SAT solver for generating possible plans.

CCalc uses the nonmonotonic causal logic C+ which through formulas and axioms expresses causal rules. This logic advantages to other action description languages by distinguishing between variant and invariant symbols and assuming that nothing changes unless there is a cause for it. A causal theory in CCalc is constituted by sorts, variables, constants and axioms (causal laws). CCalc allows to express sorts of things identified by a single name. A mechanism of simple inheritance is represented. Variables are typed with the defined sorts and are used in the construction of axioms (rules) to indicate the possible values that can be used in a given predicate.

CCalc define constants instead of predicates. Constants are typed too, allowing not only associating Boolean values to them but objects (instances of sorts) too, which allows expressing the current value of an object property in a given time. Constants can be of type fluent or action. Fluent constants receive any kind of value and can be inertial or rigid, depending of its value is allowed to

change or not in time. Nevertheless, the value of fluent constants doesn't change unless there is an axiom indicating so. A fluent formula is a formula such that all constants occurring in it are fluent constants; meanwhile an action formula is a formula that contains at least one action constant and no fluent constants.

CCalc has a single construct for expressing rules or axioms, called causal laws. Causal laws can express static and dynamic laws which relate events occurring in the same time or in consecutive time frames, respectively. Causal laws have the general form `caused` F `if` G, where F and G are formulas. Through this formalism CCalc can express *static laws* (if F and G are fluent formulas), *action dynamic laws* (if F is a fluent formula and G is an action formula), and *fluent dynamic laws* (if it is added `after` H to the causal law, being H any formula).

Time in CCalc is expressed explicitly at two levels: in dynamic causal laws (through the `after` clause) and on the declaration of facts or queries (indicating the time slice, which ranges from 0 to the `maxstep` variable). CCalc represents a state with a set of instantiated fluent formulas and transitions with events that result of the instantiation of (action or fluent) dynamic causal laws.

CCalc provides a set of abbreviations over its general form that synthetically expresses causal axioms. For instance, the statement `nonexecutable` F `if` G is an abbreviation for `caused` $\perp$ `after` $F \wedge G$, and indicates that the action formula F cannot be executed if G holds.

4 Modeling publications auditing

Alloy was used for modeling the properties of the domain and representing constraints of the domain. On the other hand, CCalc was used for modeling the process involved in information auditing.

4.1 Modeling repository consistency with Alloy

The different types of publications were declared by extending the signature `Publication`. Common metadata was declared as attribute of `Publication`. Specific constraints were declared on each publication type. For instance, see the declaration of articles published in proceedings of conference in Figure 2. The attribute `publishedIn` is constrained to be a single instance of `Proceedings`.

```
sig InProceedingsArticle extends Publication {
publishedIn: one Proceedings
}
```

Fig. 2. Declaration of the InProceedingsArticle signature.

Information like publication's status and the current,previous and actual year were declared through typed constants. The signature `Year` was declared ordered with respect to the predicate `next`.

Consistency rules were expressed through predicates indicating the possible inconsistencies. Figure 3 shows an example of an inconsistency rule. The similarity between publications uses simple comparison between titles, but given that titles are defined as signatures it is possible to use an attribute representing similarity measures. This rule also compares the list of authors.

```
pred samePublication[p1, p2: Publication] {
p1 != p2
p1.title = p2.title and p1.year = p2.year
all a: Person | a in p1.author <=> a in p2.author
}
```

Fig. 3. The SamePublication inconsistency predicate.

For modeling the valid operations in the repository, the signature `Repository` was declared ordered, indicating on each time step which publications were contained in it. Publications were included at model generation and it was simulated their insertion, deletion and modification by introducing and extracting them from the `Repository` state.

Figure 4 shows the operation for adding a publication to the repository; the first two lines represent preconditions and the last one is the postcondition. `r` and `r'` are used for relating two repository states. In the `traces` fact there are indicated four valid operations between consecutive repository states (`r'=r.next`). We also modeled as operations: the deletion of a publication, the change of the publication year and the change of the publication status.

```
pred addPublication[r, r': Repository, p: Publication] {
some p2: Publication | p.id = p2.id => p2 not in r.contains
p not in r.contains
r'.contains = r.contains + p
}
```

Fig. 4. The addPublication operation.

Consistency of the repository was expressed through predicates `IsConsistent(r,p)` and `AllConsistent(r)`, which verifies that all consistency predicates hold for every publication p contained in the repository at state r.

For determining if the given definitions are capable of generating valid models, we used the command `run` with the predicate `AllConsistent`. Limiting to a single repository state we validated (statically) the existence of models satisfying the definitions of publication types and consistency constraints. The Alloy visualizer was useful for inspecting graphically these models.

For validating actual information from the repository, we introduced publication information using constants. Properties like title similarity was calcu-

lated previously. Additionally, the initial conditions for the repository were given through a set of facts.

The detection of inconsistencies was made through asserts. See for instance the assert in Figure 5, which through a command `check` produces models on which the duplicated record inconsistency holds.

```
assert assert_duplicated_pub {
all r: Repository | all disj p1, p2: pubsInRep[r] | not
samePublication[p1, p2]
}
```

Fig. 5. Duplicated publication verification through an assert.

The last task with Alloy was generating a recovery plan for a given example or scenario. The following command asks Alloy for possible models where the last state of the repository is consistent (within 3 states):

`run AllConsistent[ro/last] for 3`

Plans for recovering the consistency when publications $P1$ and $P2$ are the same include: 1) removing $P1$, 2) removing $P2$, and 3) removing $P1$ and $P2$. Identified plans were inspected in the visualizer projecting the solutions along the `Repository` signature. Nevertheless, the predicate that produces the transition was not shown graphically.

4.2 Modeling the Multiagents System with CCalc

CCalc use was used for modeling the agents constituting the proposed Multiagents System. Modeling was divided in three sections: the MAS framework, the domain and the actions. Agents modeling followed the definitions of the Independent Choice Logic (ICL) [5]. The main sorts of the MAS framework were: `agents`, `beliefs` and `entities`. FIPA ACL[1] messages were also represented as sorts.

In the domain were modeled classes of agents, types of beliefs and discourse objects like publications, inconsistencies and persons. Discourse objects are used during agent communication and reasoning and were defined without properties. Unlike Alloy, in CCalc the declaration of a type of object is made separately through sorts and constants. See for instance the declaration of the LogMonitor agent class in Figure 6.

According to ICL, an agent has observables (beliefs) and can perform actions; both are defined through constants where, by convention, the first argument identify the agent class. Observables are defined as inertial fluents (that can be true or false along time); meanwhile actions identifiers are defined as exogenous actions (controlled by the agent itself). Additional agent characteristics can be

[1] FIPA Agent Communication Language Specifications. `http://www.fipa.org/repository/aclspecs.html`

```
:- sorts
agent >> agLogMonitor.
:- constants
believes(agLogMonitor, belNewPublication, publication, person) ::
inertialFluent;
actInform(agLogMonitor, agRepGuardian, belNewPublication, publication,
person):: exogenousAction.
:- constants
monitors(agLogMonitor, repository) :: inertialFluent.
```

Fig. 6. LogMonitor agent class definition in CCalc.

expressed through constants, like in the inertial fluent `checks` that expresses the capability of a LogMonitor agent for monitoring some repository.

External events to the MAS are defined and controlled through two constants: a fluent identifying the occurrence of the event (controlled on the execution) and an action identifying the consequences of the event. For instance, Figure 7 shows the constants used for simulating the insertion of a new publication in the repository.

```
:- constants
exNewPublication(publication, person) :: inertialFluent;
evNewPublication(publication, person) :: exogenousAction.
```

Fig. 7. New publication event constants.

Specific beliefs were declared as subclasses of the sort `belief` in order to reduce the branching factor on model generation. For instance, `belNewPublication` represents the belief an agent has regarding the existence of a new publication in the repository. For instance, see how this belief is used in the declaration of the *LogMonitor* (Figure 6).

Agent's beliefs were initialized on time 0. For example,

```
0: [/\ LM /\ P | -believes(LM, BNP, P, PER)].
```

indicates that all *LogMonitor* agents (`LM`) believe that there is no new publication in the repository (`BNP`) at time 0.

An example of perception of an external event by an agent is shown in Figure 8; whenever a new publication is registered in the repository, the *LogMonitor* agent becomes aware of it. `evNewPublication` is an auxiliary predicate that might consider additional information in the transformation of perceptions into beliefs. Other changes in agent beliefs were also represented by causal rules.

Action descriptions were codified in terms of agent beliefs. For instance, Figure 9 shows the causal laws used for constraining the execution (preconditions) and indicating the effects (postconditions) of the action on which an *Auditor*

```
evNewPublication(P, PER) causes believes(LM, BNP, P, PER),
-exNewPublication(P, PER).
```

Fig. 8. Example of agent perception.

agent (AUD) informs a *RepGuardian* (RG) that the publication P is inconsistent (INC) but has no correction (CO). Note the use of the universal quantifier (\/) and negation (-) for triggering this action when *some* inconsistency has no correction.

```
nonexecutable actInform(AUD, RG, BNR, P) if -[\/ INC \/ CO |
believes(AUD, BIC, P, INC, CO) & believes(AUD, BNR, P, INC)].

caused believes(RG, BNR, P, AUD), -believes(AUD, BIC, P, INC, CO),
-believes(AUD, BNR, P, INC) after believes(AUD, BIC, P, INC, CO) &
believes(AUD, BNR, P, INC) & actInform(AUD, RG, BNR, P).
```

Fig. 9. Example of action description.

Similarly to Alloy, we used constants for declaring actual agents and publications. Their invariant properties were declared through unconditional static causal laws like: `caused checks(aud1, duplicated)`.

CCalc queries were used for: 1) a progressive specification of agents, and 2) identification of plans. In the first case, the specification of the scenario was given step by step and it was verified its feasibility. In this way we debugged agent actions and determined the rules and beliefs required for goal achievement.

Once that agents were fully specified, we calculated those plans capable of satisfy such specifications. For instance, the query in Figure 10 was used for generating a valid plan given two new publications in the repository, each publication with a different type of inconsistency. The goal is expressed in the step previous to the last by indicating that the *RepGuardian* must believe that both publications were audited and that the *User* agent of the person responsible for p2 was notified of the automatic correction of its inconsistency.

```
:- query
label :: 4;
maxstep :: 15;
0: exNewPublication(p1, per), -exNewPublication(p1, per2);
0: exNewPublication(p2, per), -exNewPublication(p2, per2);
0: -aprioriInconsistent(p1, status), aprioriInconsistent(p1, duplicated);
0: aprioriInconsistent(p2, status), -aprioriInconsistent(p2, duplicated);
maxstep-1: believes(rg, audited, p1), believes(rg, audited, p2),
believes(us, corrected, p2, status, changeStatus).
```

Fig. 10. Auditing plan generation.

CCalc generated valid plans showing the sequence of actions executed and those predicates holding on each time step. This query produced a valid plan on 15 steps where each one contained the simultaneous execution of 2 or 3 actions. Running this query with less time steps didn't produce a valid plan, meanwhile a higher number of steps produced many other valid plans.

5 Conclusions

Alloy and CCalc demonstrated to be useful tools in the modeling and construction of the MultiAgent System for the auditing scenario. Both tools use SAT solvers which makes them very efficient for an agile testing and execution.

Alloy's compatibility with the Object Oriented Paradigm eased modeling the domain. Beyond that similarity, Alloy permitted to express internal conditions that should be satisfied in every instance of a given class. Its rich set of operators and constructors allowed to make complex definitions in a few lines. Its tools for visualizing the generated cases eased the specification of domain constraints.

Nevertheless, even when Alloy provides the facilities for representing dynamic models, the specification of frame conditions was cumbersome. In order to satisfy the model, Alloy produces changes during the execution that must be controlled with more frame conditions. Besides, tracking the operations performed during a simulation requires analyzing every two consecutive states, which is not necessary with CCalc due to its printing of the executed actions.

CCalc demonstrated superiority in the specification of frame conditions through the notions of causal laws and inertial fluents. Its abbreviations for expressing causal laws provided an easy implementation (and reading) of the model. Its support for specifying and monitoring concurrency was remarkable.

In both languages it was possible to generate plans towards a given goal and evaluate given scenarios. The Alloy model will permit to evaluate a set of similar publications extracted from the database and evaluate their consistency (in a single state scenario). The CCalc model will allow generating plans that a set of agents should follow in order to audit and correct automatically a set of publications.

References

1. M. Zuckerman, "America's best graduate schools. news & world report." U.S. News & World Report, USA, 2004.
2. F. Cantu, H. Ceballos, S. Mora, and M. Escoffie, "A knowledge-based information system for managing research programs and value creation in a university environment," in *Proceedings of the Eleventh Americas Conference on Information Systems*, (Omaha NE, USA), Association for Information Systems (AIS), August 11-14 2005.
3. D. Jackson, *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.
4. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, H. Turner, and J. L. V. Lifschitz, "Nonmonotonic causal theories," *Artificial Intelligence*, vol. 153, p. 2004, 2004.
5. D. Poole, "The independent choice logic for modelling multiple agents under uncertainty," *Artificial Intelligence*, vol. 1-2, no. 94, pp. 7–56, 1997.