Verifying SGAC Access Control Policies: A Comparison of ProB, Alloy and Z3

Diego de Azevedo Oliveira(✉) and Marc Frappier(✉)

Université de Sherbrooke, Québec, Canada
{dead1401,marc.frappier}@usherbrooke.ca

Abstract. This paper describes the formalisation of SGAC access control policies using Z3 and then we compare the performance with ProB and Alloy. SGAC is an attribute-based, fine-grain access control model that uses acyclic subject and resource graphs to provide rule inheritance and streamline policy specification. To ensure patient privacy and safety, four types of properties are checked: accessibility, availability, contextuality and rule effectiveness. Automatic translation of SGAC policies into each specification language has been defined. ProB offers the best verification performances, by two orders of magnitude. The performances of Alloy and Z3 are similar.

Keywords: Access control · Consent management · Verification · ProB · Formal model · Alloy · Z3

1 Introduction

SGAC (*Solution de Gestion Automatisée du Consentement/ Automated consent management solution*) [2] is a powerful, attribute-based, fine-grain access control model for EHR that uses acyclic subject and resource graphs to provide rule inheritance and streamline policy specification. To ensure patient privacy and safety, four types of properties are defined: *Accessibility*, *Availability*, *Contextuality*, and *Rule effectiveness.*

In [2], ProB [4] and Alloy [3] are investigated to verify these SGAC properties. ProB is mainly based on constraint logic programming using the CLP(FD) finite domain library of SICStus Prolog, while Alloy relies on Kodkod and SAT solvers. In this paper, we intend to complement this study by exploring a different technology, SMT solvers, using Z3 [1]. We present the translation of SGAC to SMT-LIB2 using the Python API for Z3. We then compare the performance of Z3 with that of ProB and Alloy using the translation described. We also improve this translation by fully taking into account rule conditions in contexts, instead of an abstraction as proposed in [2].

This work was supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada).

This paper is structured as follows. A brief overview of SGAC is presented in Sect. 2. Section 3 presents the formalisation of the SGAC model in Z3. Section 4 describes the formalisation of the properties to check. Section 5 brings the performance tests and compares each tool. We conclude this paper in Sect. 6.

2 Brief Introduction to SGAC

SGAC is an access control model with conflict resolution. Conflict resolution is based on a definition of precedence between rules; the rule with the highest precedence is chosen to determine the access decision. The precedence relation is not a total order. When there are several maximal elements, access is granted when all of them are permissions. The definitions provided in this section are taken from [2]. SGAC uses *directed acyclic graphs* (DAG). A sink of a DAG G is a vertex without any successor; $sink(G)$ denotes the set of all sinks of G.

An SGAC policy $P = (S, R, L)$ consists of a DAG S denoting subjects, a DAG R denoting resources, and a set of rules L. A rule $l \in L$ permits to specify who (subject) has access (action and modality) to what (resource) and when (priority and condition). A request is a demand the *subject* issues in order to execute an *action* on a *document*. A rule l applies to a request iff all of the following conditions are satisfied: the request subject is a descendant of the rule subject; the request resource is a descendant of the rule resource; the request action is the same as the rule action; the rule condition holds.

One strong point of SGAC is how it deals with conflict resolution. A *conflict* occurs when more than one rule apply to a request, and if they have different modalities. It is necessary to decide which rule has the highest precedence and determine the access decision. Let r_1, r_2 be two different applicable rules for a request:

1. If r_1 has a smaller priority than r_2, we say that r_1 has precedence over r_2.
2. If r_1 and r_2 have the same priority, and if the subject of r_1 is more specific than the subject of r_2 (i.e., the subject of r_1 is a descendant of the subject of r_2 in the subject graph), then r_1 has precedence over r_2.
3. If r_1 and r_2 have the same priority, and neither of their subjects is more specific than the other, then a prohibition has precedence over a permission.

Figure 1 provides a small example where a hospital has just one doctor, Edward, and he is part of the GP Physicians and the Psychologists groups. A patient was accepted to the hospital and the resources available are the exams: a blood test and an urine test. Four rules with the same priority are defined. In rule 1 there is a prohibition of access from the hospital to the exams. That way, just more specific groups may have access to content. In rule 2 the GP Physicians are permitted to access blood tests. In rule 3, the Psychologists are prohibited to access blood tests. In rule 4, Edward is allowed to access urine tests. Edward is only granted access to the urine test of the patient. Since rule 2 is overridden by rule 3, he is prohibited from accessing blood tests. This happens because the rules have the same priority, also neither rule2 is less specific than rule3 or vice-versa, and rule4 is more specific than rule1.

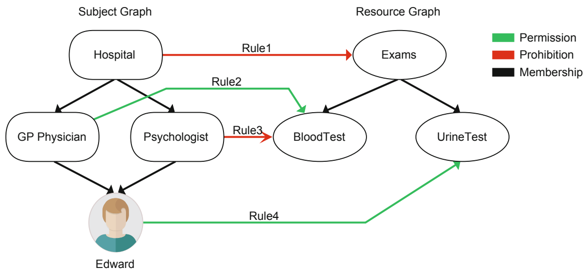

Fig. 1. SGAC graphs with rules

3 Formalisation of SGAC with Z3

Z3 [1] is a Satisfiability Modulo Theories (SMT) solver developed by Microsoft Research. It is specialized for solving background theories. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, algebraic datatypes, uninterpreted functions and quantifiers. Several programming languages are available as front-end to interface with Z3, such as Ocaml, C++ and Python. Z3 uses combination theory and novel algorithms. It is composed of a congruence closure engine, a SAT solver-based and several default theory solvers or plug-ins.

Z3 does not natively support sets and relations. A set S that is a subset of a sort T can be represented by a boolean function $S_f \in T \rightarrow BOOL$. The predicate $s \in S$ is represented by $S_f(s)$. Similarly, an n-ary relation $r \subseteq T_1 \times \ldots \times T_n$ is represented by a function $r_f \in T_1 \times \ldots \times T_n \rightarrow BOOL$. A record $w \in W$, with $W = \mathsf{struct}(a_1 : T_1, \ldots, a_n : T_n)$, is represented by one function $a_{i,f}$ for each attribute a_i such that $a_{i,f} \in T_W \rightarrow T_i$, where T_W is a sort representing the set of all records. The value of an attribute a_i of w is given by $a_{i,f}(w)$.

The formalisation of SGAC in Z3 is highly inspired from the B specification of [2]. Z3 is not able to solve the SGAC specification in a single model, thus model staging is needed. Z3 does not natively support model staging. Thus, we use Python scripts to do model staging. In the first stage we calculate the graphs, their transitive closure and determine rule precedence. The second stage calculates the maximal applicable rules. The third stage verifies the SGAC properties. After solving a stage, we use Python to get the instance found and generate new constraints representing the values of the symbols solved in the next stage.

The sets of clause SETS of the B model are represented by sorts in our Z3 model. Although a sort in Z3 is infinite, it is possible to restrain its set of elements using constraints. For SGAC it is mandatory to use the elements that we nominated, and not let the solver choose others.

It is then possible to name the elements of the sort, using constants, and use them in the constraints. In our model, each element of subject, resource, rule, context and the two modalities is unique. A constraint must be added to state that each pair of constants are distinct from each other (*i.e.*, pairwise inequality).

To build the subject and resource graphs, we use a relation as previously described. We also compute the transitive closure of the subject and resource graphs externally in Python, taking advantage of their acyclicity, which is more efficient than the generic transitive closure operator provided in Z3Py.

A rule is represented by a structure as explained above. The set of requests is represented by a relation using the sinks of the subject and resource graphs, as in B. The next step is to specify conflict resolution and how the rules are ordered. We then define: *applicable*, takes the pair subject-resource, as a request, and decides if the rule is applicable to the pair, returning a boolean; *maxElem*, a function that was declared in the B definitions, responsible for giving the maximal rule elements for a given request; *isPrecedeBy*, that connects a subject, a resource, to two rules (*r1*, *r2*) and a boolean. The boolean only holds true when *r1* is less specific than *r2* and the two rules are part of the same request, represented as the subject and resource; *pseudoSink* (psdSink), returns all maximal applicable rules for a given request for a given context.

4 Properties Verification

Accessibility and Contextuality. Accessibility verifies whether a subject *sub* can access a resource *res* in a context *con*. Contextuality determine which contexts make a given request granted. Access is granted when the maximal applicable rules of each request (*sub*, *res*) under the context *con* are all permissions. We define the function *accessibility*(*sub*, *res*, *con*) that returns true when access is granted. Then, we add a constraint that holds if the request for the given context is accessible and we ask Z3 to solve it. In contrast to [2], where two formulas are used, we use a single formula to compute both.

$$\begin{aligned}&accessibility(sub, res, con)\\&\quad\Leftrightarrow\ \forall(rule).(psdSink(sub, res, con, rule) \Rightarrow r\_mod(rule, perm))\\&\qquad\wedge \exists(rule).(psdSink(sub, res, con, rule))\end{aligned}$$

Availability. Finding hidden data allows one to warn the patient that within some conditions, their data may be out of reach. A document is defined hidden or unreachable under the context *con* if there is not a valid request under *con*.

The formalisation in Z3 checks if there is a document under the context that cannot be accessed by anyone. Z3 will return the context with hidden documents.

$$\begin{aligned}&hiddenDataSet(con, res)\\&\quad\Leftrightarrow\ res \notin dom(graph\_res) \wedge \forall(sub).(Request(sub, res)\\&\qquad\Rightarrow\ \neg(\forall(rule).(psdSink(sub, res, con, rule) \Rightarrow r\_mod(rule, perm))\\&\qquad\qquad\wedge \exists(rule).(psdSink(sub, res, con, rule))))\end{aligned}$$

Rule Effectivity. A rule that can never be the determinant for the evaluation of a request is said ineffective. For instance, if we take two rules with different priorities, one of them has to be ineffective since one will always have precedence over the other. Effectivity of a rule r is formally defined in [2] as follows: Case r is a prohibition: there is at least one pair request-context where r is a maximal applicable rule, and r is the sole prohibition among the maximal rules for this pair; Case r is a permission: there is at least one pair request-context where r is the sole maximal rule.

$$\begin{aligned}
&\mathit{ineffectiveSet}(\mathit{rule}1)\\
&\quad\Leftrightarrow\ \neg(\exists(\mathit{sub},\mathit{res},\mathit{con}).\\
&\qquad\qquad(\mathit{Request}(\mathit{sub},\mathit{res}) \wedge \mathit{conRule}(\mathit{con},\mathit{rule}1)\\
&\qquad\qquad\wedge\ \mathit{psdSink}(\mathit{sub},\mathit{res},\mathit{con},\mathit{rule}1)\\
&\qquad\qquad\wedge\ (\neg(\exists(\mathit{rule}2).(\mathit{psdSink}(\mathit{sub},\mathit{res},\mathit{con},\mathit{rule}2) \wedge\ \mathit{rule}1 \neq \mathit{rule}2)))\\
&\qquad\qquad\qquad\vee\ (\mathit{r\_mod}(\mathit{rule}1,\mathit{proh})\\
&\qquad\qquad\qquad\quad\wedge\ \forall(\mathit{rule}2).(\mathit{psdSink}(\mathit{sub},\mathit{res},\mathit{con},\mathit{rule}2)\\
&\qquad\qquad\qquad\qquad\qquad\wedge\ \mathit{rule}1 \neq \mathit{rule}2 \Rightarrow \mathit{r\_mod}(\mathit{rule}2,\mathit{perm})))))
\end{aligned}$$

5 Performance Test

In this section, we discuss the results of the performance tests we executed for the four checked properties. Tests were performed with randomly SGAC models. We vary the following parameters: the number of vertices in each graph (subject and resource), the number of rules, the number of contexts and the number of requests. We check all four SGAC properties by modifying only one parameter

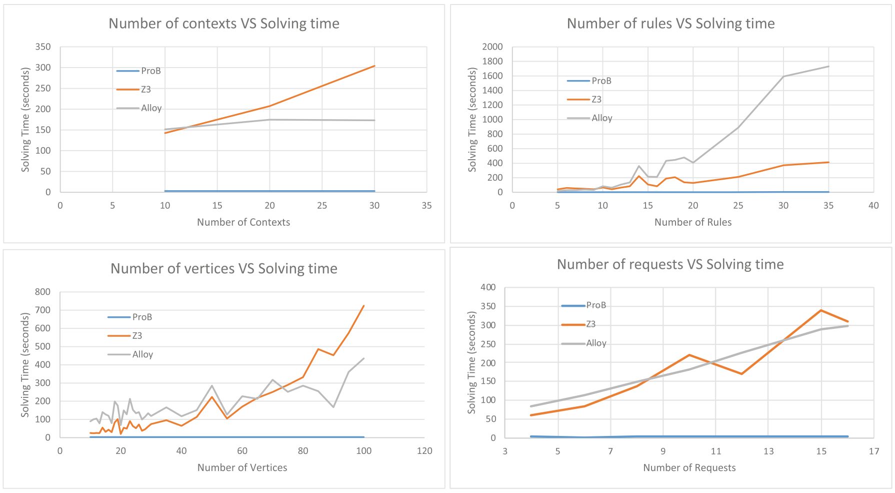

Fig. 2. SGAC performance tests.

at a time. For each defined value of the parameters, at least 6 randomly generated models are created and solved with Z3, ProB and Alloy. The tests were performed on a Windows 10 64-bit OS, with 16 GB of RAM and Intel®Core^{TM} i7-7700 3.60 GHz as CPU.

As shown in Fig. 2, ProB is faster than the other two solvers by two orders of magnitude in every occasion. Z3 is consistently better than Alloy when varying the number of rules, while Alloy outperforms Z3 when varying the the number of contexts. When varying the number of vertices, Z3 is slightly faster up to 75 vertices, after which Alloy performs better than Z3. As detailed in [2], we use a staged model finding in ProB to solve the properties. The B model of SGAC uses constants to define the subject and resource graphs. The transitive closure of graphs are computed using the B closure operator, for which ProB provides an efficient implementation. B machine operations using set and relation operators are used to solve the four properties checked.

In our experiment, Alloy is the only model that does not use staged model finding. We decided to investigate if staging could help in increasing its performance. We divided the Alloy model into three smaller models, following the approach used in the B model. The instances found in one stage are used to build the next stage. This staged model finding cuts the computation time in half, but it is still outperformed by ProB.

6 Conclusion

In this paper we compared Z3 with the B and Alloy models of SGAC [2] for checking SGAC properties. Our experiment shows that ProB is still the most adequate of the three solvers for this task. It is quite easy to use staged model finding in B to increase performance, compared to Z3 and Alloy. B operations can be easily used to compute the state variables needed to check the properties. During the development of the Z3 model, improvements were made to better take into account rule conditions. We were able to add constraints to the contexts, representing the formula of rule conditions. These modifications were also deployed on the B model. In future work, we plan to investigate the use of Z3 to further analyse rule conditions when a policy is constructed. Another approach would be to explore αRby [5], a deep embedding of Alloy in Ruby.

References

1. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
2. Huynh, N., Frappier, M., Pooda, H., Mammar, A., Laleau, R.: SGAC: a multi-layered access control model with conflict resolution strategy. Comput. J. **62**(12), 1707–1733 (2018)
3. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012)

4. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. JSTTT **10**(2), 185–203 (2008)
5. Milicevic, A., Efrati, I., Jackson, D.: αRby - an embedding of alloy in ruby. In: Ameur, Y.A., Schewe, K. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 56–71. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_5