SAT-Based Arithmetic Support for Alloy

César Cornejo
Department of Computer Science,
National University of Río Cuarto, Argentina
National Council for Scientific and Technical Research (CONICET), Argentina
ccornejo@dc.exa.unrc.edu.ar

ABSTRACT

Formal specifications in Alloy are organized around user-defined data domains, associated with *signatures*, with almost no support for built-in datatypes. This minimality in the built-in datatypes provided by the language is one of its main features, as it contributes to the automated analyzability of models. One of the few built-in datatypes available in Alloy specifications are integers, whose SAT-based treatment allows only for small bit-widths. In many contexts, where relational datatypes dominate, the use of integers may be auxiliary, e.g., in the use of cardinality constraints and other features. However, as the applications of Alloy are increased, e.g., with the use of the language and its tool support as backend engine for different analysis tasks, the provision of efficient support for numerical datatypes becomes a need. In this work, we present our current preliminary approach to providing an efficient, scalable and user-friendly extension to Alloy, with arithmetic support for numerical datatypes. Our implementation allows for arithmetic with varying precisions, and is implemented via standard Alloy constructions, thus resorting to SAT solving for resolving arithmetic constraints in models.

ACM Reference Format:
César Cornejo. 2020. SAT-Based Arithmetic Support for Alloy. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3324884.3415285

1 PROBLEM STATEMENT AND MOTIVATIONS

Formal specification is an essential part of rigorous software development methodologies, in particular those that through the preciseness of mathematical and logical formalisms, have the potential of providing stronger guarantees of correct problem understanding and description, and the correct functioning of software [2]. While the advantages of formal methods to software development are acknowledged, it is generally agreed that their practical application requires mastering complex mathematical languages, as well as reasoning techniques associated with these, that exceed the typical experience and knowledge of the average software engineer. These issues can be tamed by developing formalisms that have simple constructions, closer to the abstractions that developers are accustomed with, accompanied by *automated* analysis mechanisms, supported by tools. Achieving these while at the same time attaining sufficient expressive power to capture properties of interest, and providing efficient analysis, is of course very challenging. Alloy is a formal specification language that achieves an outstanding compromise between the above-mentioned characteristics. It is expressive, it has a simple syntax and relational semantics, easy to be mastered by any trained software engineer, and provides efficient automated analysis for specifications, via a reduction of property checking in Alloy, to SAT solving [10]. Alloy follows the style of other model-based specification languages, and models software as well as its environment by describing data domains, properties of these, as well as operations/transformations between these domains.

Alloy's designers made an effort in keeping the language minimal, in particular in relation to built-in datatypes: the language features almost no built-in datatype, so users need to introduce ad-hoc data domains to capture intended properties of software. This is extremely important in the analyzability of the language. An exception to the "no built-in datatype" rule in Alloy is integers, which are supported by the language, and its treatment is limited to relatively small bit-widths (very small integer ranges). The reason is that, during analysis, all integers within the defined bit-width are explicitly present as atoms in the model's universe, together with all other atoms defined in the model [6]. The reason for this representation has to do with the fact that a non-exhaustive enumeration of integer atoms in a model may lead to satistiability outcomes in analysis that contradict the intuition, e.g., preventing a formula from holding only because the integer resulting from a fully legal arithmetic operation does not belong to the universe of discourse.

For many specifications, the restricted support for numerical datatypes is not a limitation, e.g., when the nature of the specification makes the relational ad-hoc constructions dominate; but in many cases, a more sophisticated provision of numerical datatypes and arithmetic for these, is essential [7]. A clear scenario witnessing this situation arises in the increasing setting of Alloy being used as a backend for automated analysis, e.g., for automated test input generation [8] or bounded verification for higher-level programming languages [3, 5], for extensions of Alloy for dynamic behavior [4, 11], or as a backend for the analysis of fully-fledged (informal) modeling languages [1]. In these contexts, a better support for numerical datatypes and arithmetic is essential, and is the main motivation of our work. The problem is challenging because it does not only comprise the integration of Alloy with SAT or SMT based implementations for numerical domains; it has to be done in a way that is faithful to the Alloy's specification style, and preserves the main intuitions in the use of the language. In particular, and

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3415285

related to the issue described above, it should never be the case that a formula has a counter intuitive analysis outcome, simply because there is an insufficiently large scope (maximum number of atoms of the corresponding domain) for numerical domains, within the bit width being considered.

2 CURRENT DEVELOPMENT

As part of my PhD, I am working on extensions to Alloy to better support software specification in particular contexts. A specific extension of my work is based on DynAlloy [11], which extends the language with dynamic logic elements, that allow one to specify sequential programs annotated with pre- and post-conditions. This language is in fact the vehicle to reduce program analysis tasks, such as test generation and bounded verification, to SAT-based analysis [5]. In analyzing programs in higher-level programming languages, the support for numerical datatypes and arithmetic becomes crucial.

A particular trend in providing Alloy with support for arithmetic is by interfacing with SMT solvers. This has numerous advantages, in particular in relation to efficiency as it would profit from advances in SMT; and it also has known drawbacks, in particular the limitation to decidable fragments of arithmetic constraints, e.g., linear constraints, and the non-trivial task of consistently merging SAT-based Alloy analysis for relations, with SMT-based analysis for numerical domains. Our approach instead attempts to capture numerical domains and arithmetic constraints within standard Alloy, and make use of the provided SAT-based analysis for arithmetic constraint resolution, i.e., using the so called *bit blasting* approach [9] to dealing with arithmetic. The idea is to provide a more powerful support for numerical datatypes for *any* application of Alloy, not just for applications using it as an analysis backend. Thus, our work does not only involve integrating Alloy with SAT-based mechanisms for arithmetic resolution, but also *designing* how this is going to be achieved, i.e., preserving the intuitive interpretation of Alloy models with its current integer representation. Our work requires the design of new *Int* and *Float* types, consistent with how the former is currently dealt with in the language. Using SAT for dealing with arithmetic will pay a price in efficiency, compared to using SMT, but in the context of Alloy we believe it is better not to be limited to decidable fragments of numerical domains.

The approach I am following essentially proposes to treat numerical values as atoms of library signatures, which will represent numbers as bit vectors, and will be equipped with operations (specified via predicates) for numerical manipulation. Libraries will provide different alternative precisions (8, 16, 32, 64-bit integers and floating point numbers), an their corresponding representations will follow the established IEEE standards. An example of such a representation is shown in Fig. 1, where a library for 8-bit integers is partially depicted. Signature `Number8` defines the bit-vector integer representation, and operations for integer arithmetic are captured via predicates, following the usual low-level logical operators. A main difference with integers as currently provided in Alloy is that they will *not* constitute a fully-populated domain (currently, Alloy forces to contain atoms for every integer representable within the selected bit-width); instead, numerical domains will be subject to scope definitions for analysis, as any regular signature in the language. Preliminary evaluations with our alternative show that the profit in scalability is significant. We also aim at providing a suitable visualization of numerical values, treating and operating with these as numerical constants, and hiding their corresponding bit-vector representations.

```
/** Representation of a Integer of 8 bits */
sig Number8 {
   b00: Bool,
  ...
   b07: Bool
}

fun AdderCarry[a: Bool, b: Bool, cin: Bool]: Bool {
  Or[ And[a,b], And[cin, Xor[a,b]]]
}

fun AdderSum[a: Bool, b: Bool, cin: Bool]: Bool {
  Xor[Xor[a, b], cin]
}

pred Sum[a: Number8, b: Number8, result: Number8] {
   let c_0 = False |
   let s_0 = AdderSum[a.b00, b.b00, c_0] |
  ...
   result.b00 in s_0
  ...
}

pred Eq[a: Number8, b: Number8] {
   a.b00 = b.b00 and a.b01 = b.b01 and
   a.b02 = b.b02 and a.b03 = b.b03 and
   a.b04 = b.b04 and a.b05 = b.b05 and
   a.b06 = b.b06 and a.b07 = b.b07
}
...
```

Figure 1: Part of an 8-bit Integer library.

3 FUTURE WORK

I have implemented Alloy libraries for integers and floating-point numbers, as described in the previous section, for several precisions. A first step of future work (besides implementing the libraries for other precisions), which I have already started performing, is making extensive experimental evaluation, to debug these implementations, perform and propose improvements, and overall evaluate the performance of our approach. I am currently considering the Roops benchmark [12] for this task. Once this stage is completed, my main task will be on usability, extending Alloy's parser in such a way that allows the user to refer to numerical values using the usual numeric notations, instead of having to resort to the low-level bit-vector representation. As described earlier, our new representation of numerical domains will have to guarantee that property satisfiability/unsatisfiability is not compromised by the scope of numerical domains, maintaining the semantics of arithmetic constraints in Alloy, in its current version.

Besides providing arithmetic support for Alloy, our main goal will be to incorporate these representations into the core of DynAlloy, and make it part of the translation from Java programs into SAT-based analysis [5]. Finally, a longer-term objective is to replace the low-level bit-blasting that SAT will be performing through our characterization of numerical domains and operations, by the direct use of the hardware's arithmetic-logical unit. This will demand tailoring the SAT solving process, e.g., to select variables for splitting, when these correspond to numerical values, from the least significant bit onward, and calling the arithmetic-logical unit when sufficient parts of the operands have been set.

REFERENCES

[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2007. UML2Alloy: A Challenging Model Transformation. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings (Lecture Notes in Computer Science)*, Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil (Eds.), Vol. 4735. Springer, 436–450. https://doi.org/10.1007/978-3-540-75209-7_30

[2] Edmund M. Clarke and Jeannette M. Wing. 1996. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.* 28, 4 (1996), 626–643. https://doi.org/10.1145/242223.242257

[3] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular verification of code with SAT. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, Lori L. Pollock and Mauro Pezzè (Eds.). ACM, 109–120. https://doi.org/10.1145/1146238.1146251

[4] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. 2005. DynAlloy: Upgrading Alloy with Actions. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. ACM, New York, NY, USA, 442–451. https://doi.org/10.1145/1062455.1062535

[5] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Trans. Software Eng.* 39, 9 (2013), 1283–1307. https://doi.org/10.1109/TSE.2013.15

[6] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press.

[7] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

[8] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2011. TestEra: A tool for testing Java programs using alloy specifications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 608–611. https://doi.org/10.1109/ASE.2011.6100137

[9] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Springer. https://doi.org/10.1007/978-3-662-50497-0

[10] Sharad Malik and Lintao Zhang. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* 52, 8 (2009), 76–82. https://doi.org/10.1145/1536616.1536637

[11] Germán Regis, César Cornejo, Simón Gutiérrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti, and Marcelo F. Frias. 2017. DynAlloy analyzer: a tool for the specification and analysis of alloy models with dynamic behaviour. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 969–973. https://doi.org/10.1145/3106237.3122826

[12] Roops Benchmark: [n.d.]. https://github.com/taoxiease/roops