Mathematics for the exploration of requirements

Michael Huth
Department of Computing
Imperial College London
London, SW7 2AZ, United Kingdom
M.Huth@doc.imperial.ac.uk

Abstract:

The exploration of requirements is as complex as it is important in ensuring a successful software production and software life cycle. Increasingly, tool-support is available for aiding such explorations. We use a toy example and a case study of modelling and analysing some requirements of the global assembly cache of .NET to illustrate the opportunities and challenges that mathematically founded exploration of requirements brings to the computer science and software engineering curricula.

Keywords: Requirements, under-specification, model checking, pragmatics.

1 Introduction

Requirements are a key ingredient in the process of designing and realizing complex systems. Requirements need to be discovered, elicited, documented, reviewed; and checked for topicality, consistency, and ambiguity [Gause89] *throughout* the software life cycle. Without clearly understood requirements and their proper management, large projects are likely to fail.

We can only guess that requirements were poorly understood or managed by the Daimler-Benz subsidiary Toll Collect when the German Ministry of Transport asked them to build and deploy a nation-wide toll system imposed on interstate usage of trucks by the end of 2003. This system is currently inoperable, the ministry wants payment of damages, whereas Toll Collect wants yet another deadline extension. Curiously enough, a similar system began its successful operation in Austria on 1 January 2004.

Can mathematics, integrated in simulation tools, help in the exploration and management of requirements? The emergence of such tools and their use in practice is too recent to answer with a resounding "yes," but the increased rate at which such tools find their way into practice may eventually result in that very answer [1]. Current tools and their supporting mathematics can already help considerably in making students realize the importance of requirements and their subtle exploration and management modalities. Such aid is needed since current courses in undergraduate curricula offer little opportunity to create such an awareness and students' individual or team projects rarely reach the degree of complexity at which requirements engineering would become operable and pay off.

[1] Evidenced by the effort of Microsoft Research to integrate testing and model-based validation of programs in A#.

The effectiveness of these emergent tools resides to no small degree in the fact that requirements engineers, designers, and implementers use abstraction for system modelling and comprehension. We cite two key examples. Aspect-oriented requirements engineering (see e.g. www.early-aspects.net) aims to modularise and reason about properties that affect the entire requirements or software-architecture level, we mention security and data integrity as possible aspects. Viewpoints [Nuseibeh94, Jackson95, Sommerville98] and behavioural goals [Clarke00] are complementary to aspect-driven exploration as abstractions that separate concerns in systems engineering.

2 Which Mathematics?

Abstraction through separation of concerns or aspect-oriented modelling and analysis is necessary when confronted with real requirements of real systems. Any mathematics that supports system modelling and analysis, be it for industrial use or educational mission, therefore has to be able to cope with and perhaps even capitalize on such aggressive abstraction techniques. Consequently, mathematics for the exploration of requirements has to work differently from the mathematics for proving, say, the (partial) correctness of a program that reverses a linked list --- where we know the entire state space and how program statements transform it.

Active use of such mathematics may also require problem-solving skills that are orthogonal to the ones that succeed when there is a single or few “canonical” solutions, e.g. using linear algebra to compute the (unique) steady-state distribution of a finite-state Markov chain.

What is required is a mathematics that can model and reason about artefacts in the *presence of uncertainty*. Some requirements may not be known (e.g. should an elevator system support a user interface for visually impaired people?), some design decisions may not yet have been resolved (e.g. should a component be able to add a plug-in from one of its own classes?) or the scope of analysis may cover several types of objects (e.g. we may have to check that, whenever a secured transaction occurs, the accessing party has proper security clearance regardless of the role of that party in the system).

3 Under-specified Relational Models

Exactly what differences in mathematical formalism arise from such a presence of qualitative uncertainty [2], especially when it comes to the active learning of students? For sake of illustration, we use D. Jackson's constraint language and analyser Alloy [Jackson01] to explain the kind of mathematics and pragmatics we have in mind [3]. The reflections and arguments made apply, by and large, to any model-based formalism for reasoning in the presence of qualitative uncertainty.

[2] This paper focuses on *qualitative* requirements only.

[3] Specifically, we use Alloy 2.0 throughout.

To re-consider the issue of whether the state space and the state transformer relation are completely determined (as *is* the case for a program manipulating a linked list), consider the Alloy declarations

```
sig Element {}

sig Graph {
  nodes : set Element,
  edges : nodes -> nodes
}
```

which under-specify an unstructured set `Element` and a set `Graph` which is structured since all elements of this type, `g : Graph`, contain a subset `g.nodes` of `Element` (the state space) and a binary relation `g.edges` (the state transformer) that relates elements of `g.nodes`.

A signature `sig T { … }` provides a *template* and constraints for the creation of *structured objects* (here, directed graphs). Declarations `x : m T` state that `x` is of type `T` and the optional `m` is a *multiplicity constraint.* If `T` is an atomic type (e.g. `Element`), then `m` could be `set` (making `x` a subset of `T`), `option` (making x the empty set *or* a singleton of type `T`, a *scalar*) or simply absent (making `x` a scalar of type `T` by default). If `T` is non-atomic (e.g. `nodes -> nodes`), then `m` is absent and there are other means of enforcing multiplicity constraints. These language-design decisions reflect that variables of atomic type are most often scalars and variables of non-atomic types are most often unconstrained. Note that `S -> T` is the type of relations that relate objects to type `S` to objects of type `T` and that `x.f` accesses the "field" `f` of `x`.

Students often model such a `g : Graph` and reason about it, e.g. nodes could be services and the edge relationship could express a dependency between such services. In dealing with requirements, however, we may not be in control of *choosing* `g`, as only constraints on graphs are declared and not a complete graph per se. For example, the requirements could say

(1) "We need a graph with at most five nodes such that every node can reach exactly four nodes, has no self-loop, and is on a cycle."

Fundamental questions and ensuing activities are then the following:

1. *Are the requirements over-constraining* the anticipated system? In our example, is it impossible to get a graph that meets all these constraints? Consistency checking is needed to ensure that the entire model, or aspects thereof (e.g. a business transaction in an e-commerce system) are consistent with other requirements.

2. *Are the requirements implying other goals/objectives* of the system? For example, is every graph that meets (1) also strongly connected? Goal checking is needed to ensure that our requirements entail needed system objectives.

3. *Are the requirements allowing interesting state or behaviour?* For example, can we generate a graph that satisfies (1) *and* has a cycle of length three? Simulation and the generation of scenarios (e.g. [Uchitel03]) are needed to provide such possibilities of exploration.

If we can generate *any* simulation that meets all requirements, we know that the requirements are consistent; a system meeting all expressed requirements is realizable. Thus, we can illustrate the first and third activity by means of the same example. Alloy's `fun`-statement is the declaration of a parameterised constraint that can either be *analysed* for consistency or *invoked* in declarations of other fun-statements etc. For example,

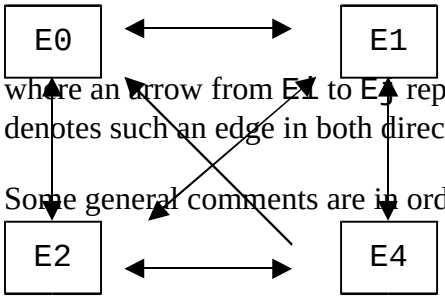
```
fun AGuidedSimulation(g : Graph) {
  all n : g.nodes | # n.^(g.edges) = 4
                    && not n in n.(g.edges)
                    && n in n.^(g.edges)
} run AGuidedSimulation for 5
```

if analysed, tries to non-deterministically generate a graph `g` with *at most* five nodes (the directive `run AGuidedSimulation for 5`) such that *all* constraints in its body are true. Here there is only one constraint, namely, that all nodes of that graph (the declaration `all n : g.nodes |`) satisfy a conjunction (&&) of three properties:

- that they have exactly four reachable states in the transitive closure of `edges` (`# S = k` declares that set `S` has exactly k elements and `^(g.edges)` is the transitive closure of `g.edges`);
- that they do not have an edge to themselves; and
- that they are on a cycle (a finite path from that node back to it).

The reader is welcome to generate such a graph unaided; if this is too easy, change 4 to 5 and 5 to 7 above, respectively. Expressions such as `n.(g.edges)` deserve explanation. For a relation `r : S -> T` and a subset `A` of `S`, the expression `A.r` denotes the set of those `t` in `T` for which there is some `a` in `A` that is related to `t` through `r`. Note how Alloy *identifies a scalar* (e.g. `n`) *with a singleton set* and so `n.(g.edges)` is the set of those nodes reaches via an edge from `n`. For example, if `A` is `{E0}`, then `A.edges` equals `{E1, E2}`in the diagram below.

The tool may provide such a solution or it may report that no solution could be found. Here the tool reports a solution which can be visualized or explored by clicking through the structure of the solution: we find `Element = {E0, E1, E2, E3, E4}`, `g.nodes = {E0, E1, E2, E4}`, and the relation `g.edges` as specified in the diagram



where an arrow from `Ei` to `Ej` represents and edge from `Ei` to `Ej` and a double arrow denotes such an edge in both directions.

Some general comments are in order.

1. This approach is *model-based and property-based*: we write a module that declares structure and its constraints; then we ask a specific question (the property) about these structures and their constraints and may get back as answer a suitable *model* of first-order logic. Therefore, modelling and reasoning about models are the core activities in this approach. We may want to ask many questions before we are satisfied. Each question and its answer are like an extensive *test* of our requirements.

2. *Checking whether properties are implied or consistent with constraints* written in (an extension of) first-order logic *is undecidable* but checking such relationships within fixed bounds on the sizes of models is decidable and can be done by SAT solvers (which check the satisfiability of Boolean formulas). This is why we specified an upper bound on model sizes (5 above) so that the problem is analysable.

3. *Abstraction is implicit* in these declarations as they state only structure and constraints that are of immediate concern. Our Alloy module declares no internal structure for type `Element` although its real-world counterpart, e.g. the set of nodes in a network, would have and need plenty of structure.

4. *Uncertainty is* expressed as *a form of abstraction.* The declaration of `Graph` allows many instances of that type so an analysis may make a non-deterministic choice of a graph meeting certain constraints [4]. Below we will see that uncertainty, as an abstraction, can also reside in other language features such as *multiplicity constraints*.

If requirements are not consistent with goals, points of friction need to be identified and trade-offs for their resolution have to be discussed. Alloy uses `assert`-statements for *goal checking*. Assertions cannot be invoked elsewhere but only analysed. The body of an assertion is a closed formula and the analysis (using `check` as a directive instead of `run`) tries to find a model in which that formula is *false*. Pragmatically, assertions formulate goals for a model that the analysis attempts to *refute*.

The declaration

```
assert OurRequirementsImplyStrongConnectedness {
  all g : Graph | AGuidedSimulation(g) =>
                    StronglyConnected(g)
} check OurRequirementsImplyStrongConnectedness for 5
```

states as goal that all graphs with at most five nodes are strongly connected if they meet the requirements in (1). We write => for logical implication, and invoke fun-statements in a way similar to method and procedure invocations (e.g. the invocation `AGuidedSimulation(g)` above). Recall that strong connectedness means that every state is reachable from every state:

```
fun StronglyConnected(g : Graph) {
  all n, m : g.nodes | m in n.^(g.edges)
```

[4] Indeed, the tool allows for the generation of a "next" solution if there is one.

```
                        && n in m.^(g.edges)
} run StronglyConnected for 5
```

The prudent specifier would analyze `StronglyConnected` for consistency, i.e. test whether it really generates a strongly connected graph. Analyzing the assertion we learn that no solution was found. A solution would have been a violation of our goal and studying such a scenario would have helped with identifying the sources of inconsistencies that can then be used to discuss the resolution of these inconsistencies. Such discussions may or may not involve use of this tool.

D. Jackson's *small-scope hypothesis* claims that violations will occur at a moderate scope already if they occur at all. This hypothesis is needed as pragmatics dictates the use of bounds, for the unbounded goal-checking problem is undecidable. As we have not found a violation of our goal in scope 5, we may have reason to believe that it holds for our currently specified requirements. If we need to be 100% certain about it, we have to prove the implication in the `assert`-statement above mathematically, e.g. using a full-fledged theorem prover.

The subtleties of changing bounds on model sizes for analysis are illustrated by repeating this goal check with at most five nodes (analysis scope 5 as before) and exactly two states reachable from all nodes (replacing 4 with 2 in the body of `AGuidedSimulation`). The tool then reports a violation with `g.nodes` = `{ E0, E1, E3, E4}` such that `g.edges` has an edge from `E0` to `E1` and vice versa; an edge from `E3` to `E4` and vice versa; and no other edges.

Another caveat concerning the pragmatics of such tool use is that we may be fooled even if the tool is always smart enough to decide whether a goal violation within the scope exists. If our requirements, here `AGuidedSimulation`, are inconsistent and never analysed for consistency, then no violation of any goal will be found as such a violation has to meet the requirements! To make matters worse, changing *any* constraint *anywhere* potentially questions that overall consistency anew.

Students find it generally challenging and instructive to emulate `check` instances as `run` instances and vice versa. For example, they need to appreciate that a `run` of a `fun`-statement corresponds to a `check` of the negation of the formula obtained from the body of the `fun`-statement by existentially quantifying all its parameters (using `some` instead of `all`).

4 A case study

We now re-iterate and deepen the points of the last section by means of a more realistic case study, some requirements on the global assembly cache of Microsoft .NET [Eisenbach03]. A cache is a “self-sufficient” set of components. In Alloy, the declaration

```
sig Component {
  name : Name,             -- name of component
  main : option Service,   -- possible main
  export : set Service,    -- services supplied
```

```
  import : set Service,    -- services required
}{ no import & export }    -- constraint
```

states that a component has a `name`, has *or does not have* a `main` service (which starts execution), has a set of services it can `export` to other components, as well as a set of services `import` it requires from other components (in order to be executable). The lexeme `--` may be followed with explanatory, non-executable comments. Brackets `{ … }` immediately following a declaration `sig T { … }` constrain, and apply to, all instances of that type. So the constraint `no import & export` applies to all elements of type `Component` and ensures that no component can offer a service that it also requires. Uncertainty is expressed in that `main` is either a *scalar*, an element of type `Service`, or the empty set; we also do not specify how many import or export services a component has.

A cache is then a set of components that is *coherent* in that all services required within the cache can be provided therein:

```
sig Cache {
  components : set Component
  scheduler : components -> Service -> components
}{ components.import in components.export
 }  -- cache is coherent
```

The constraint `components.import in components.export` applies to every cache. The expression `components.import` collects/unions all import sets of all components in `components` and `components.export` has a similar effect. In mathematical notation,

```
components.import = { s : Service | some c : components |
                                    s in c.import }.
```

Coherence therefore states that all import services of the cache can be exported within the cache: the cache is self-sufficient, i.e. executable.

The scheduler is declared as a *higher-order relation*: given a component `c`, we obtain a relation `c.scheduler` of type `Service -> components` that may associate with a service `s` a component `c'`; the intent being that `c'` provides service `s` to `c`. Such intent should be made explicit with an explanatory, non-executable comment. Further constraints are needed (and omitted here) to ensure the soundness of the scheduler, e.g. that `c.scheduler` associates to `s` some `c'` if and only if `s` is in `c.import`. Note that the meaning of `c.scheduler` is computed in the same way as that of `n.edges` and `components.import` above.

Uncertainty is present as we do not specify how many components a cache has, nor do we say how coherence has to be realized. We also under-specify the scheduler, which associates to a client component `cs` and a service `s` at most one supplier component `cs`. The "at most one" can be enforced by the multiplicity constraint `?` so that `components -> Service -> components` above becomes `components -> Service ->? components`.

Here is a specification of a cache-management transaction that adds a component to a cache:

```
fun AddComponent(G, G' : Cache, c : Component) {
  not c in G.components              -- pre-condition
  G'.components = G.components + c   -- post-condition
} run AddComponent for 3
```

Parameters `G` and `G'` represent the cache before and after the addition of component `c`. The use of the prime is entirely pragmatic, no special meaning is inferred from it by the tool. The declarative body states that `c` can only be added if it is not already in the cache, and that the new set of components equals the old one plus `c` (the `+` denotes the union of two sets where the scalar is, as always, identified with its singleton set). The use of type `Cache` for `G` and `G'` *implicitly* enforces that `G` and `G'` are coherent. Such hiding of pre- and post-conditions in types impacts the pragmatics (e.g. specifiers need to be aware of this or may prefer explicit constraints). This choice of type `Cache` in `AddComponent` over, say, `set Component` for parameters `C` and `C'`, also impacts the range of applicability of our model. Since coherence is enforced implicitly, we cannot model the addition of a component to a cache that somehow became incoherent and is therefore in need of repair.

Pragmatic issues abound in formulating and analysing such specifications. The tool's feedback may not indicate clearly enough whether it could determine that no solutions exist within these bounds or whether no such solutions were found by its search engine (here a SAT solver). The user-set bound 3 may implicitly conflict with constraints made in the body of the function or elsewhere. The user may think that `for 3` means *exactly three*. The user may write the body as an implication, saying that the pre-condition implies the post-condition; a solution could then be generated in which the pre-condition is false!

An analysis of the goal check

```
assert AddIsDeterministic {
  all G,G1,G2 : Cache, c : Component {
    AddComponent(G,G1,c) &&
    AddComponent(G,G2,c) => G1 = G2
} check AddIsDeterministic for 3
```

saying that `AddComponent` is a function in its second parameter, is likely to reveal that Alloy employs *name equality* and not *structural* equality of objects, which are structured elements. One could then refine the assertion above by replacing `G1 = G2` with `StructurallyEqual(G1,G2)`, an invocation of a `fun`-statement that constrains `G1` and `G2` to be structurally equal. With this fix in place, we still get violations which simply educate us, perhaps not surprisingly, that determinacy is uncalled for as the addition of a component to a cache creates new opportunities for scheduling required services.

Having specified all relevant transactions for cache management (removing, replacing a component etc), one can declare a signature `State` and its transition relation and

then ask, e.g., whether unsafe states can be reached. Unfortunately, certain questions are at odds with generating a model as diagnostic evidence, e.g. the analysis of

"Is every set of components realizable as `G.components` for some cache `G`?"

results in a model where the sets `Components` and `Cache` are chosen so as to make this impossible, resulting in frustration not insight [5].

The challenges that students and teachers face in this modelling and analysis activity are plenty. How stable are specifications under local change of constraints? When do we use conjunction, when implication; when `all` and when `some`? Why does a solution of a `fun`-statement show things which the user did not expect to see but which were unconstrained in the body of the `fun`-statement? And why would a user judge this to be problematic? And finally, when have we modelled enough and at the right level of granularity?

We don't have answers to all of these questions. In fact, the biggest challenge seems to be in conveying to students that the main objective of such an activity is not to learn a linear method for arriving, without fail, at a final and correct set of requirements (a traditional way of presenting engineering problems and solutions), but that such an activity triggers a discourse that has "therapeutic" effects in that it helps us with understanding important aspects of our requirements better. Ideally, the tool's feedback triggers a *discussion* of requirements. The ensuing, often cost-saving, discourse may not involve a formal tool but may never have happened without the feedback of such a tool.

Acquisition of soft skills through hard mathematics!

5 Conclusions

Executable mathematical formalisms that specify *sets* of first-order logic models by under-specifying structural and relational aspects of a system are an effective tool for teaching the challenges and vexations of modelling and exploring requirements. Such formalisms, however, involve activities with complex pragmatics. A programming course for C, Java or Haskell can safely focus on syntax and semantics. Languages for the exploration of requirements need a simple syntax and semantics so that a course can emphasize the pragmatics of that language and of the exploration of requirements per se.

Exploration tools such as Alloy meet most of these criteria and are increasingly being used in computer science education. Explicit and implicit design decisions of such languages impact the transparency of its pragmatics, although different users will judge the degree of transparency differently. Students will often complain about specific features of such a language, saying things such as ``If I could only write this in X, then I could make it work." where X stands for Prolog, Haskell etc. These complaints are most often rooted in the trials and tribulations of the *process* of declaring and exploring requirements and have little to do with the principles and particular feature choices of (declarative) language design [Schmidt94].

[5] Mark Ryan pointed this out to me.

This model-based approach of exploring the consistency and consequences of requirements naturally fits into several courses as a self-sustaining module, although it does assume a familiarity with notation similar to first-order logic and an open mind toward declarative programming. We can see it being employed in courses on discrete mathematics, declarative programming, formal methods and specifications, requirements engineering, and logic.

For further information on the tool Alloy and genuine introductions to it, please consult http://alloy.mit.edu

A more detailed exposition of the case study in Section 4 can be found in the second edition of the text book "Logic in Computer Science: Modelling and reasoning about systems" ([Huth00] co-authored with Mark Ryan), to appear in the Spring 2004 with Cambridge University Press; this book's home page is at http://www.cis.ksu.edu/~huth/lics/

Acknowledgements

Peter Henderson read various drafts of this article. His comments helped improve the overall presentation and clarity of this article. Daniel Jackson also helped with most useful comments on a late draft. He and his software design group at MIT made me want to teach the executable exploration of requirements. Many points in this article are merely re-iterations of points made in various papers authored by members of that group. Mark Ryan made me think about the expressive limitations of properties whose bounded check via a SAT solver produces inherently spurious diagnostics.

References

[Clarke00] E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.

[Eisenbach03] S. Eisenbach, V. Jurisic, and C. Sadler. *Modeling the evolution of .NET programs.* In: IFIP International Conference on Formal Methods for Open Distributed Systems, LNCS. Springer Verlag, 2003.

[Gause89] D. C. Gause and G. M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House Publishing, 353 West 12th Street, New York, NY 10014, 1989.

[Huth00] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, First Edition, January 2000.

[Jackson95] D. Jackson. *Structuring Z Specifications With Views.* ACM Trans. on Software Engineering and Methodology, 4(4):365--389, October 1995.

[Jackson01] D. Jackson, I. Shlyakhter, and M. Sridharan. *A Micromodularity Mechanism*. In: Proc. ACM SIGSOFT Conf. Foundations of Software

Engineering/European Software Engineering Conference (FSE/ESEC'01)}, Vienna, Austria, September 2001.

[Nuseibeh94] B. Nuseibeh, J. Kramer, and A. Finkelstein. *A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification.* IEEE Transactions on Software Engineering, 20(10):760--773, October 1994.

[Schmidt94] D. A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing Series. MIT Press, 1994.

[Sommerville98] I. Sommerville, P. Sawyer, and S. Viller. *Viewpoints for requirements elicitation: a practical approach.* In: Proc. 1998 International Conference on Requirements Engineering (ICRE'98)}, Colorado Springs, Colorado, April 6-10 1998. IEEE Computer Society Press.

[Uchitel03] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. LTSA-MSC: Tool *Support for Behaviour Model Elaboration Using Implied Scenarios.* In: Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), Lecture Notes in Computer Science, pages 597--601, Warsaw, Poland, 7-11 April 2003. Springer Verlag.