Modelling, Verification and Proof
A conversation with Laurent Voisin

Ken Robinson & Laurent Voisin

September 2010

After the *Rodin User and Developer Workshop*, run as part of AVOCS'10 in Dsseldorf (September 2010) I had a discussion with Laurent Voisin initiated by an issue that arose in my talk *Reflections on the Teaching of System Modelling and Design.*

In the following discussion my part of the dialogue is in blue and Laurent's is in black and indented.

I've just written to Stefan asking for details of what and how much of Rodin is formally modelled. Do you have any comments on that?

> Basically, it is the static checker that has been formally modelled in event-B. This was carried with Atelier B as Rodin did not exist at that time. However, the model as never been ported to Rodin since. It is also getting more and more desynchronized with the platform as it is not updated anymore.
>
> The POG does not have any formal model per se. It is specified on paper. However, its specification is formal (rather than natural language) and contains a lot of manual proofs.
>
> The same applies to the prover (see Farhad's PhD thesis).
>
> I think that one of the reasons for not making event-B models of the POG and Prover was, beyond time and money, a matter of expressiveness of the event-B language which made very difficult to reason about mathematical formulas. This has changed with the coming of mathematical extensions that allow for inductively defined datatypes. However, we currently have no financement for launching such a modelling activity now.

Thank you very much for your extensive answer to my question.

The answer is very interesting. Would you mind me quoting you in an annotation I am making of my talk?

> Please do quote me, possibly making the text a bit more formal, as it was not intended for a general audience, but rather a private communication. I've also added some complements below which I think are relevant.
>
> This is something of a very vexed question and answer. It is pretty natural for people introduced to something like B (Classical or Event) to expect that the toolkit will be

modelled in itself, and they are very disappointed if they learn that it isn't, or not completely.

As concerns classical B, in my opinion, there is a strong distinction between the software you can model with the tool and the software of the tool itself, although people have a tendency to confuse them. All of the software produced industrially with Atelier B is safety critical embedded software, which imposes a lot of restrictions on the kind of software (most notably restrictions enforced by standards such as EN 50128). On the contrary, Atelier B implements none of these restrictions as it is rather a desktop CASE tool. A very simple and blatant example of this essential difference is that recursion is prohibited in safety critical software, while it is heavily used in Atelier B (as most data structures manipulated in the tool are recursive, such as abstract syntax trees).

Similarly for Rodin: event-B is a notation for modelling general reactive systems (not necessarily based on software) and proving properties of them, while the Rodin platform is rather a desktop tool. I agree that Rodin can be seen as a reactive system in some respects and thus modelled with event-B, but this is not as direct a mapping as one can imagine.

Last but not least, comes the matter of money well spent. For a safety critical software, my feedback is that it is wise to invest some effort in making a formal model, as the risks are high. Conversely, for a tool which is not in the critical path, this is more questionable. When developing the Rodin platform, we took a very pragmatic approach as we had important but limited resources. We thought that making a formal model of the static checker was interesting because, at that time:

1. Event-B was not so mature and it was a way to investigate further its usability on a medium-size model.
2. This allowed us to see where Atelier B for event-B was lacking support for usual tasks.
3. This provided us a strong basis for developing the first component of the platform (e.g., the static checker).

In retrospect, it was an excellent decision to run this task, and the return on investment was good.

As concerns the POG, we already had a good experience with the static checker development, the POG is much simpler and the modelling of PO generation was seen as difficult to carry in event-B. This is why we opted for not making an event-B model, but rather a formal specification (with proofs) on paper.

As you can see, the decision to make a formal model is not an absolute one, but rather contextual, depending on available resource, schedule, etc. I understand that this is not easy to explain to students, as it is, in my mind, a real engineering decision rather than a theoretical one.

As concerns the user interface, which I did not cite in my previous mail, we took very early the decision of not making a formal model of it, because it was too brittle. At that time, the principles of the GUI were not known and we made a lot of tries and errors before reaching the current interface (which still needs more polishing). For this particular component, it would have been really a mistake to attempt to make a formal model of it before experimenting with it. Still now that it has become more stable, I'm not sure it would be useful to make a formal model of it. The only thing that would be useful would be to model some key elements.

I don't know if you noticed, but I gave a justification in terms of something like Rodin being a tool that is strenuously exercised and hence becomes reliable. Cliff Jones obviously disagreed with this justification and I admitted that might not be a reasonable explanation. I've since decided that I should have stuck to my view. While I strongly believe in modelling design, I think we have to give significant credit to the skill of the designer, and it's very clear that something like Rodin is designed and implemented by very skilled people.

> Thank you very much for the compliment. This might also be explained by these people having a formal background. I've already noticed that people that are aware of formal models have a strong tendency to still think in more formal terms than the average programmer, thus producing neater designs. This might look a bit like the difference between an engineer and a craftsman: both can produce designs, but they will be very different (for the better or the worse).

I think it goes a bit further than "think in more formal terms". I think your later reference to "engineer" is important. I think that engineering incorporates informal design abilities. Something to do with structure, but not necessarily quantified. I am reminded of a comment made by Carroll Morgan about his Z specification of the Unix file system. He said, (something like) "the success of my specification is very much a reflection of the design of the Unix file system". Good designs (in many areas) generally depend on "spatial" relationships, which in many cases are not quantified in the way you might quantify them in Event-B. Indeed, despite my great regard for Event-B the formal specification —and even more so the discharge of proof obligations— loses a lot of the elegance of the structure being described.

This type of spatial design can show in the way an event is specified, as there is usually more than one way to specify an event, some more clearly correct than others.

> Could you please elaborate on this. Where is it that you loose the structure in the formal model? Do you have a concrete example so that I can grasp it?

As I said, this is difficult, so I may have difficulty explaining to you.

I think we are very good at grasping high level abstractions. The thing I have greatest difficulty with —difficulty is the wrong word— is with defining a context for some data structure for a model. I don't have difficulty with the definition, I have difficulty with the proof. When I say "difficult" I mean that the proof is laborious/tedious due the low level of the reasoning. I would like to raise the level of the reasoning. I define theorems to assist with that, but there are times when the proof is still too low level.

Clearly we need to build libraries and maybe this will become possible with the new generic extension capability announced at the workshop.

I have a number of Contexts with POs that have not been discharged, but in which I have very strong confidence that they are true. I just get to a point where I decide that "life is too short". Having said that, I hasten to add that I spend quite a lot of time on discharging POs.

As a corollary to the above I remember a talk —I think by someone from Matra Transport talking about the Paris line 14 development— in which it was stated when undischarged POs were investigated there were a substantial number that were not provable, but were able to be corrected without changing the model. I believe what was intended here was that some axioms or theorems were “not quite correct”. Something has been assumed, but not stated. Jean-Raymod may have some knowledge here.

This concurs with my experience.

That’s a very long winded and probably unsatisfactory answer.

BTW: the sort of thing I’m talking about here is very relevant to teaching. I would never expect my current students to be able to discharge the POs from these contexts, while discharging the POs for machines that use the contexts are reasonably easy–in comparison.

> Yes, I understand very well your answer and I fully agree with it. For all but the most trivial model, one has to construct an underlying theory for the data structures (in case of software) or the mathematical structure (for a system) that are needed for reasoning about an artefact (software or system). Developing this underlying theory is nothing simple and needs a lot of expertise (both in set theory and usage of proving tools at hand).
>
> I think that we can make here a parallel with applied maths and physics on one side, and engineering. Usually, engineers do not resort directly to the underlying theory but use something more practical such as tables (in old times) or generic software (in modern time). The engineer then needs to know the underlying theory (so that he can apply it wisely, knowing in particular its limits of application).
>
> For instance, when reasoning about strength of material, the engineer (at least me) knows about tensorial calculus and the underlying mechanics of a continuous medium. However, the engineer in his day to day life do not use it directly, but use a more direct application of it that contains some theorems and mathematical models that are directly applicable (e.g. a model using planes and beams instead of tensors).
>
> My strong belief (and I think Jean-Raymond shares it) is that mathematical extensions should allow us to develop such machinery so that Event-B modellers do not have to resort directly to the modelling of the underlying mathematical structures, developing again and again axioms and theorems. For example, mathematical extensions should provide reusable mathematical structures together with the means to use them efficiently in models. One of our aim in Deploy is to develop a library of such structures.
>
> As concerns the case of wrong or missing theorems, my return on experience with classical B is that, as you said, most of the time it can be fixed easily without much impact on the model. However, from time to time, this can hide a real problem in the software. I recall one case where a single PO (which was of course the last one undischarged) lead to make a big rearchitecting of a B model. This was caused by this PO concerning some invariant that was actually a cornerstone of the argument justifying the (correctness of the) model. In the original model, the invariant was true most of the time, except in some rare transitional state. However, when put in this transitional state, the software would crash!
>
> In retrospect, most of the time people make the right informal argument for proving their problem correct. Then, having a wrong theorem is not a problem. However, people also have a tendency to wishful thinking (e.g., taking an implication in the wrong direction). Then, mechanically-checked proofs detect the second case, except when wishful thinking occurs twice: once in the model and once in a used ”theorem”.

In my opinion, this explains why incorrect theorems are usually not a problem, but can really raise an issue in rare circumstances.