A Framework for Automating Design Level Exploration within the Rodin Toolset

by Peter Kovacs
with contributions from: Gudmund Grov and Andrew Ireland

School of Mathematical and Computer Sciences, Heriot-Watt University

October 2014
1. Introduction

Automated design space exploration in the context of formal modelling is the process of automated generation new models/specifications. This can e.g. be used to fix faulty models, either by abstraction away the mistake or by adaptation/mutation of the model. This report details a tool developed by the first author over the summer of 2014, working with the second and third authors. The tool will act as a framework for future research into this exciting and novel area, targeting the Event-B formalism and the Rodin toolset.

The tool includes an operator language, where an operator describes a particular transformation of a model. Such operators can be combined to describe larger transformations. The objective was to implement a set of atomic operators that can be used efficiently to define new operators and could be applied on a model easily to generate new transformations. Several tools have been integrated to perform analysis on the models.
2. Implementation and Design

Event-B is a refinement-based methodology used in formal modelling. The dynamic part of each abstraction is represented by a Machine, the static part by a Context. Models may consist of multiple machines and contexts that are managed in a project by Rodin. Rodin Platform is an Eclipse based IDE that supports the development and analysis of formal models written using the Event-B method. It is extendable, the framework is implemented as a plug-in to Rodin platform.

2.1 Abstract Syntax Tree

The models are accessible programatically using the RodinDatabase, however for the purpose of performing transformations in a more efficient way they are translated to an intermediate representation developed as a part of this framework using Scala. This new representation is better for exploring the design and generating a high number of transformations.

The operator language being developed for generating transformations of the models, are based on this intermediate representation.

The Rodin Machine to be analyzed is translated to this new representation first. The case classes composing the AST and the Parser used for parsing RodinFiles are part of the hw.cs.eventb.translate.ast package developed. The project/model can be transformed in a whole or by individually. The Parser has the following public methods:

- `getContext(ctxRoot: IContextRoot): Context`
- `printElement(element:Element )` used for printing any part of the AST
The AST generated by the Parser is completely immutable and composed of algebraic datatypes defined by ‘case classes’ in Scala. This representation is more suitable for generating transformations in the operator language.

Figure 2.1: The internal representation of an Event-B machine
Each element of the AST has a type of Element. There are 4 Element types:

- EventElement
- MachineElement
- ContextElement
- ProjectElement

Figure 2.2: The internal representation of Rodin Element


2.2 Rodin Builder

The intermediate representation can be transformed back to Rodin using the EventBCreator class which is the part of hw.cs.eventb.rodin.generator package. It also generates a unique name for the created elements. EventBCreator provides an interface to transform a single Context or Machine back to Rodin representation. EventBCreator has the following methods:

- `generateEventBContext(context: Context, projectName: String): IRodinFile`
- `generateEventBMachine(machine: Machine, projectName: String): IRodinFile`

Each of these method returns the IRodinFile of the given element which can passed on to the TypeChecker and Proof Obligation Manager and then to Pro-B for running simulations. Discussed in the next Chapter.
2.3 Operator Language

The Operator language is used for generating transformations from a model. The idea is to implement a language which is extensible meaning new operators can be written based on other operators, and can be combined and applied easily on the models.

An operator represents a certain type of transformation. Several operators have been implemented, each of them generates a large amount of transformations, but this could be reduced significantly by specifying the domain on which the operators work.

An Eval state used as a domain of the operators has been introduced in order to apply operators and write new operators based on old ones in a more natural way.

Eval contains all the transformations resulted by applying the operators. For each transformation which is represented as a MachineState class, the number of active events constrainig the domain of the transformation can be specified. If not specified all the events are considered active. The depth representing the number of transformations lead to a certain MachineState is also stored.

The Operator Language and all relevant classes are part of hw.cs.eventb.operator package.
Currently the following operators have been implemented. These are likely to change and will be extended.

- deleteArgument Deletes one of the arguments of the active events.
- deleteWitness Deletes one of the witnesses of the active events.
- deleteGuard Deletes one of the guards of the active events.
- deleteVariable Deletes a variable from the machine, replaces all elements of a predicate with True and deletes all actions that contains it.
- deleteInvariant Deletes an invariant of the machine.
- deleteAction Deletes one of the actions of the active events.
- deleteEvent Deletes an event of the active events.
- negateGuard Negates one guard of the active events.
- negateAction Negates one action of the active events.
- mergeEvents Merges two of the active events.
• combineEvents Combines two events of the active events, which means creating a new event with the actions of event1 and the guards of event2.

Each Operator generates all possible transformations. The operators can be further constrained by providing an additional argument, a restricting function. This is an optional argument, if not provided operators work on the active Event domain.

2.3.1 Combinators

The purpose of the Combinator to combine the application of operators following a certain logic.

• or(eval: Eval,ops: Eval => Eval *):Eval  
  Applies each operator of ops independently on eval and return Eval containing all transformations

• orCond(eval:Eval,condition:ProjectElement => Boolean,op1:Eval => Eval,op2:Eval => Eval):Eval  
  Applies op1 if condition is met, otherwise applies op2

• repeat(eval: Eval, condition: ProjectElement => Boolean,op: Eval => Eval):Eval 
  Applies an operator until a condition is met.

• first(eval: Eval, op: Eval => Eval) 
  Applies an operator but only generates the first transformation from each MachineState in eval.

2.3.2 Generator and OpWrapper

To be able to apply a series of operators concurrently, a Generator Actor has been implemented. This is useful when there are several series of operators that we want to apply on a model independently and see the result of the different series. These can be applied on the model concurrently as they dont affect each other. When using the Generator Actor an OpWrapper class must be used in order to pass the operators to the Generator Actor as a message.
2.4 Tools

hw.cs.eventb.tools package contains the following tools:

- Generator
- TypeChecker
- POManager
- ProBSimulator

subsectionGenerator It is used for applying the operators on the MachineState. It is useful when several operators that result in a high number of transformations are applied independently. (2.3.2) The Generator applies the Operators on the Eval state and creates the corresponding Rodin Elements then passes them to the TypeChecker.

The Number of elements generated depends on the number and nature of the operators, the constraints provided to the Operators, the number of active events and the model itself.

2.4.1 Event-B Tools

TypeChecker

The TypeChecker uses the org.eventb.core.machineSC/contextSC tool to statically check the generated elements. All the elements that fail the typechecking are discarded.

POManager

The POManager runs the machinePOG (machine Proof Obligation Generator) and the autoPOM (automated Proof Obligation Manager) of org.eventb.core package. It has two public methods:

- process(machine:IMachinRoot) running the tools mentioned above.
- getFailedPOList(machineRoot: IMachineRoot): List[IPOSequ]ent returns a list of failed proof obligations of the given machine

Although these tools are started automatically by Rodin when a machine/context is created, it is necessary to call them manually to avoid any exceptions that may be caused by the delay of the tools.

2.4.2 ProBSimulator

This is based on Teresa Llano’s TraceGenerator plugin and the classes running the simulations are adapted from that plugin as well. It runs Pro-B Simulations on the machine
and analyses the generated Animation history then a .pl trace file is generated that is handled by HR3, which is a theory exploration tools that we hope to use in the future to support this work.
3. Illustration

The following example illustrates how the framework and the Operators work. The example on which the operators are applied is a model of transferring money between bank accounts taken from the Edge Proposal. Observing that the error occurs between two events an abstraction is applied. The two events are combined and a variable is deleted.

```
Figure 3.1: Abstraction
```

This particular transformation have been implemented by first setting the active events to start and debit:

```
mchState.setActiveEvents(List("start","debit"))
```

Then, given an Eval object, the relevant operators are applied to these two events:

```
def transferAbstractCase(a:Eval):Eval =
    a apply(Operator.deleteVariable(_),Operator.mergeEvents(_))
```

This creating 4 machines

Time: 19milliseconds

After the transformations are generated, they are passed on to the TypeChecker which
discards all transformations that fails the typeChecking.
Checking Machines
Time: 890 milliseconds
Number of Elements: 4
Number of Discarded Elements: 3

After the typechecking finished there is one transformation left which is the one we aimed for.

If the domain of the transformations, the active events are not constrained then the operators generate all possible transformations. Applying the same operators as in the previous example now without setting the active events:

Creating 24 machines
Time: 21 milliseconds
Checking Machines
Time: 2780 milliseconds
Number of Elements: 24
Number of Discarded Elements: 22

There are two transformations left after the typechecking finished. One is the same that the previous example returned.

The typechecking is really expensive in terms of time as it requires to generate RodinMachines first, then apply the tools of Rodin to perform typechecking. Doing some refactoring it can be improved.
4. Future Work and Conclusion

The main goal of this project was to develop an extensible framework for generating transformations. Using this tool it would be possible to analyze a model and apply operators based on the results. The framework and the integration of the tools can be further improved to make it more suitable for analyzing models and generating transformations.

- Experiment with Operators and Tools - Analyze a model with the available tools and generate transformations using the Operators
- Integrate HR3 - Run HR3 with the generated trace file of Pro-B
- Use tools together to generate data about a model, evaluate the results and apply Operators
- Automate the process