A case for Rebel

A DSL for Product Specifications

Jouke Stoel
CWI, Amsterdam, The Netherlands
jouke.stoel@cwi.nl

1. Introduction

Large service organisations like banks have a hard time keeping grips on their software landscape. This is not only visible while performing maintenance on existing applications but also when developing new applications.

One of the problems these organisations face is that they often do not have a clear and uniform descriptions of their products like savings- and current account, loans and mortgages. This makes it hard to reason about changes to existing products and hampers the introduction of new ones. The specifications that *are* written down often contain ambiguities or are out-of-date. Next to this, specification are almost always written down using natural language which is known to lead to numerous deficiencies [1].

To counter these problems we introduce Rebel, a DSL for product specifications. Rebel lets users specify their product in a high-level, unambiguous manner. These specification can then be simulated which enables users to explore their products before they are build.

We have created Rebel for a large Dutch bank and are currently in the process of specifying existing banking products.

Since Rebel is in the early stages of development we would like to use DSLDI to gather feedback on its current design and proposed future directions.

2. Rebel

Rebel is a domain specific language for product specifications. It is inspired on formal methods like Z [2], B [3] and Alloy [4]. It is aimed at helping a large Dutch bank in bridging the gap between informal specifications written down in natural language or passed on mouth-to-mouth towards unambiguous, machine interpretable specifications. The main idea behind Rebel is to present the user with a easy to understand syntax and interface while it exploits powerful tooling like verification to check whether the specifications hold under the hood.

Rebel is implemented in RASCAL [5] as a stand-alone DSL.

2.1 Requirements

The language needed to fit the following requirements:

- Flexibility - it should be possible to tune it to the problem of the bank we were working with.
- Integration - it should be possible to integrate existing tools like model checkers and connect to existing systems in the banks application landscape.
- Adaptation - it should be easy to learn and the tooling like an IDE should be similar to the tooling currently used.

Considering these requirements we decided to create a new language. This new language needed to be a linguistic hybrid to be able to support both the definition of single products as well as the overlying process.

2.2 Design

Rebel is a declarative language and centres around *specifications*. Figure 1 shows an example of such a specification.

A specification describes one product. Specifications contain *fields*, *events*, *invariants* and *life cycle*. Fields declare the data used in the specification. Events describe the possible mutations on the data under certain conditions. Invariants describe global rules which should always hold and life cycle constrains the order of events.

The definition of events and invariants is separated from usage in specifications. This is to promote reuse and to separate the responsibility of implementing an event from using an event in a specification.

Defined fields can only be of built-in types. Events can only reference fields declared in the specification, not fields of other specifications. We made this choice so that the potential state space is smaller when applying verification techniques like model checking.

Events are described using pre- and postconditions. An example event is shown in Figure 2. The semantics are straightforward; if the precondition holds then the postcondition will hold after the event is raised. Events contain runtime instance variables as well as configuration variables. Configuration variables are keyword parameters that can have a default value and can be set when the event is referenced by a particular specification. For instance, the us-

```
specification SavingsAccount {
    fields {
        balance: Time -> Integer
    }

    events {
        openAccount[minimumDeposit=50]
        withdraw[]
        deposit[]
        close[]
    }

    invariants {
        positiveBalance
    }

    lifeCycle {
        initial new -> opened: openAccount
        opened  -> opened: withdraw, deposit
                -> closed: close
        final closed
    }
}
```

Figure 1. Example Rebel specification

```
initial event openAccount
    [minimumDeposit : Integer = 0]
    (accountNumber: String, initialDeposit : Integer) {
    preconditions {
        initialDeposit >= minimumDeposit;
    }
    postconditions {
        new this.balance(now) == initialDeposit;
    }
}
```

Figure 2. Example of an event definition

age declaration of `openAccount` (Figure 1) sets the event configuration parameter `minimumDeposit` meaning that the `SavingsAccount` uses 50 as a `minimumDeposit` when an account is opened.

Invariants are global rules. They use quantifiers over data to express certain constrains that should always hold. Figure 3 shows an example that states that at all time, saving accounts should have a balance equal to or above zero.

```
invariant positiveBalance {
    all sa:SavingsAccount | all t:Time {
        sa.balance(t) >= 0
    }
}
```

Figure 3. Example of an invariant

2.3 Simulating specifications

The simulation is aimed at helping product owners and developers gain insight into their specified product. It can be used to check if the specification meets the expectations of the user. Figure 4 shows a screenshot of the simulation of

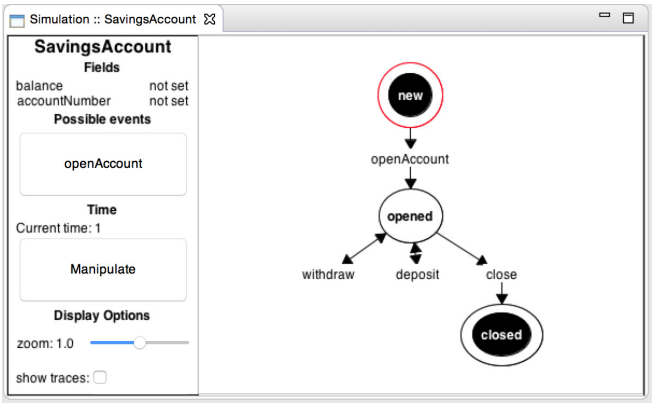

Figure 4. Screenshot of simulating the SavingsAccount specification

a `SavingsAccount`. The simulation is implemented with the use of the Z3 SMT solver [6].

3. Future work

The current version of Rebel supports the definition of single products. Next to this it is also needed to define composition of these products. In other words, the process. Since the specifications only contain fields of built-in types and can only reference themselves it is not possible to compose specifications. To overcome this we propose the use of process algebra [7] for specifying how the individual specification events should be composed. This will give us the ability to specify choices, sequencing, concurrency and communications between specifications. The question will be if we will still be able to reason about (certain parts of) the specifications since composing the specifications will have a large impact on the state space.

An orthogonal aspect is the tooling for Rebel specifications. Next to the simulation we will explore the possibility of model checking. The model checker could be used to find event traces that lead to violations of the invariants. Earlier work has shown that it is possible to translate Rebel specification to Alloy. Alloys analyser was used to find traces which would break the specification. The problem with this approach was scalability. An alternative would be to exploit an SMT solver for the same purpose [8]. One of the challenges here will be how we can bound the data in a smart way to limit the state space.

Ultimately, running systems should be generated from Rebel specifications. Since Rebel is a declarative language it will not always be straightforward to generate a correct system from this. Again SMT solvers might hold the key as shown in other work like [9].

References

[1] B. Meyer. On Formalism in Specifications. *IEEE Software*, 2(1):6–26, 1985.

[2] J.M. Spivey and J.R. Abrial. *The Z notation: A Reference Manual*. Prentice Hall Hemel Hempstead, second edition, 1992.

[3] J.R. Abrail. *The B-Book: Assinging programs to meaning*. Cambridge University Press, 1996.

[4] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[5] P. Klint, T. van der Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009.

[6] L. De Moura and N. Bjorner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[7] J. Baeten, T. Basten, and M.A. Reniers. *Process algebra: equational theories of communicating processes*. Cambridge university press, 2010.

[8] A. Milicevic and H. Kugler. Model checking using SMT and theory of lists. In *NASA Formal Methods*, pages 282–297. Springer, 2011.

[9] R. Alur, R. Bodik, G. Juniwal, M.M.K. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design*, pages 1–17, October 2013.