Solving the Bank with Rebel

On the Design of the Rebel Specification Language and Its Application inside a Bank

Jouke Stoel
Centrum Wiskunde & Informatica,
The Netherlands
stoel@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica,
University of Groningen,
The Netherlands
storm@cwi.nl

Jurgen Vinju
Centrum Wiskunde & Informatica,
TU Eindhoven,
The Netherlands
jurgenv@cwi.nl

Joost Bosman
ING Bank, The Netherlands
joost.bosman@ing.nl

Abstract

Large organizations like banks suffer from the ever growing complexity of their systems. Evolving the software becomes harder and harder since a single change can affect a much larger part of the system than predicted upfront. A large contributing factor to this problem is that the actual domain knowledge is often implicit, incomplete, or out of date, making it difficult to reason about the correct behavior of the system as a whole. With Rebel we aim to capture and centralize the domain knowledge and relate it to the running systems.

Rebel is a formal specification language for controlling the intrinsic complexity of software for financial enterprise systems. In collaboration with ING, a large Dutch bank, we developed the Rebel specification language and an Integrated Specification Environment (ISE), currently offering automated simulation and checking of Rebel specifications using a Satisfiability Modulo Theories (SMT) solver.

In this paper we report on our design choices for Rebel, the implementation and features of the ISE, and our initial observations on the application of Rebel inside the bank.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Languages

Keywords DSL, specification language, model checking, SMT, language design, industry case

ITSLE'16, October 31, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4646-7/16/10...
http://dx.doi.org/10.1145/2998407.2998413

1. Introduction

The ING bank is an organization with a long history and was among the first Dutch companies that started automating their processes. In the past decades many different systems on different technologies were created to support the ever growing need for process automation. With every new automated service and with the growing use of these services the demand on these systems also grew quickly. During these decades the underlying technologies in which new systems where implemented changed while often the underlying problem domain did not. This resulted in an application landscape with numerous applications implemented in different technologies running on different platforms.

Reasoning about the impact of change or the introduction of new features in such a large and technologically scattered application landscape is hard. Especially since the description of the domain knowledge which is captured by these applications if often missing, out of date, or incomplete. When the domain knowledge is captured it is written down in informal documents like Word files or Excel sheets without connection to the implemented application. Changing the software becomes a labour intensive task relying on the tacit knowledge of the people in the organization. As a result, the ability to predict and control the correctness (Meyer 1985), cost-of-ownership, performance and reliability is compromised.

In a public/private partnership between our research institute and the ING bank we set out to improve the quality of communication between stakeholders, to simplify the design and implementation of products and services using lightweight formal methods (Jackson 2001). The initial result of this collaboration is the Rebel specification language and its ISE. Rebel is built using of the language workbench Rascal (Klint et al. 2009), and employs the state-of-the-art SMT solver Z3 (Moura and Bjorner 2008) for simulation and

checking. In this paper we describe the current design and initial observations of Rebel within the bank.

The contributions of the current report can be summarized as follows:

- We provide a description of the requirements and design of Rebel (Section 3);
- We sketch how Rebel specifications are translated to SMT formulas for formal analysis and simulation (Section 5).
- We show how the simulation of specifications can be used to test existing applications (Section 6).
- We report on initial observations in applying Rebel and its ISE inside the bank (Section 7).

Ultimately Rebel specification could serve as the base for new applications.

2. Background

Based on discussions we had with various stakeholders within the bank we identified four challenges faced by the bank.

Dispersed Functionality. The current application landscape of the bank contains approximately 1400 applications with many interactions between them of unknown scale. This landscape is the result of nearly 50 years of software evolution within the bank incorporating many different technologies, frameworks and design styles. Some of the applications have overlapping functionality, but it is often unclear whether they behave similarly under equal conditions. As a result, changes to the software require extensive, labour intensive testing.

Scattered and Implicit Domain Knowledge. Which applications should encode which part of the domain is informally documented (if at all) and scattered across the landscape. There are many partial requirements documents, ranging from written documentation and presentation slides to Excel sheets and sometimes UML diagrams like Statecharts and sequence diagrams. These partial descriptions, however, are often ambiguous, incomplete and under-specified. It is eventually up to the developers to implement the requirements to the best of their knowledge. Ultimately, the only source of domain knowledge is the software itself, but eliciting this knowledge is a hard problem, well-known from research in reverse engineering (Tonella and Potrich 2004). As a result, questions like "What is a savings account?" or "Which operations are allowed on a savings account?" are very hard to accurately answer.

Stricter Regulations. Regulation of banking is becoming more strict. This has become even more urgent since the start of the economic crisis of 2008. As a result the accountability of the system as a whole must increase. Currently, questions like "Why did this transaction fail?" are hard to answer and require intensive manual labour like mining log files to trace calls through large parts of the application landscape.

Implicit Quality Assurance. There exists a conceptual gap between product owners—domain experts that are responsible for a specific product—and development teams. Currently the way to check whether development teams have implemented requested features correctly is by demonstrating the actual software itself or a prototype of it. It is then up to a product owner to decide whether the implemented feature meets its specifications. Since the specification are often informally documented there is currently no automated way to check whether the software meets this specification. Next to this it can be hard for product owners to find deficiencies in the product that are caused by under-specification. In other words, a product owner currently does not have many tools that help making this decision.

3. Design of the Rebel Language and ISE

The design of the Rebel language and ISE is guided by the challenges that are described in the previous section. In the coming section we will elaborate on the design choices. How these choices are reflected in the language and ISE is delayed to later sections.

Centralized and Unambiguous Specifications. To tackle the challenge of scattered and implicit domain knowledge we designed Rebel as a formal specification language in which it is possible to represent the essential characteristics of financial products at a very high level of abstraction. The language focusses on capturing domain knowledge and omits any details of technical implementation. Rebel is designed as a domain specific language (DSL) for the banking enterprise domain. This means that it should be possible to capture all parts of banking. For example, banking is both about financial transactions and customer relations. Both domains can be described with Rebel.

Increase Collaboration between Product Owner and Development Team. It is important that both product owner and development team have a thorough understanding of the product that was specified. For instance, a product owner must decide whether or not a specification is complete regarding the desired functionality. To check this so called external consistency of the product specification (Snoeck et al. 2003) the ISE offers a simulation environment as a means for rapid prototyping. Next to this it should be possible to visualize Rebel specifications in other formalisms like UML Statecharts, a formalism which is well known to the product owners of the bank. In our vision both product owner and development team take part in the specification process. By making use of techniques like simulation and other visualization methods we aim to minimize the possibility for miscommunication and thus ultimately, the inception of the wrong software.

Practical Automatic Reasoning. In light of stricter regulations reasoning on the level of specification is required. For instance in the Netherlands the legislator prohibits a person under the age of eighteen to buy certain financial products.

Being able to check whether the specification would allow or disallow this behavior is of great value to the bank. This kind of automatic reasoning should still be practical in its use, meaning that a user of Rebel should be able to check this kind of properties by the push of a button in a reasonable amount of time. In the Rebel ISE we try to balance completeness of reasoning with the response time of the reasoning process.

4. Rebel Specifications Explained

We introduce Rebel using an example specification. For explanatory reasons we chose a simplified version of the real ING savings account.

4.1 The `SimpleSavings` Account

The `SimpleSavings` account can be opened by a customer provided that the customer deposits more than or equal to 50 euro into the account on opening. After the account is opened the customer can deposit and withdraw money. Next to depositing money, money can also flow into the account when interest is received. The amount of interest that a customer gets is variable but it never exceeds a fixed percentage. This percentage differs over different saving accounts but for this `SimpleSaving` account it is fixed on 5%. In extreme cases, for instance when the customer is under suspicion of criminal behavior, the account can be blocked. This means that no money can be withdrawn from, or deposited to the account. In the end, the account can be closed provided that the remainder of money has been taken out of the account. When it is closed the account finally ends up in a state without any possible further interactions. It is invariant for every type of savings account that the balance is always positive. In other words, it is not allowed to overdraw a savings account.

4.2 Business Entities as State Machines

The Rebel implementation of the `SimpleSaving` account is shown in Fig. 1.

Each specification consists of four (optional) parts: state variables, event declarations, invariants and life cycles. The **`fields`** section describes state variables of this entity and their types (e.g., `balance`, `accountNumber`). Since Rebel has been designed specifically for the banking domain, some of the types are specific for this domain. For example, Rebel has built-in types for **`Money`**, **`Currency`** and **`IBAN`** (unique European bank account number), next to the standard types for booleans, numbers and strings. With this design we aim to maximize the range of banking domain problems which can be expressed, while using values which are natural to domain experts, and without forcing them to formalize "trivial" details (e.g. uniqueness of **`IBAN`**).

The **`events`** section contains all possible transition triggers (e.g., `openAccount`, `withdraw`, `deposit`, etc.). The **`lifeCycle`** section defines a state machine for each instance of the specification. Fig. 2 depicts the state machine of a `SimpleSaving` instance. The pattern for a transition $s_1 \rightarrow s_2 : e_1, ..., e_n$

```
specification SimpleSavings {
  fields {
    accountNumber: IBAN
    balance: Money
  }

  events {
    openAccount[minimalDeposit = EUR 50.00]
    withdraw[]
    deposit[]
    block[]
    unblock[]
    interest[maxInterest = 5%]
    close[]
  }

  invariants { mustBePositive }

  lifeCycle {
    initial init -> opened: openAccount

    opened -> opened: withdraw, deposit, interest
           -> blocked: block
           -> closed: close

    blocked -> opened: unblock

    final closed
  }
}
```

Figure 1. Rebel specification of a `SimpleSavings` account.

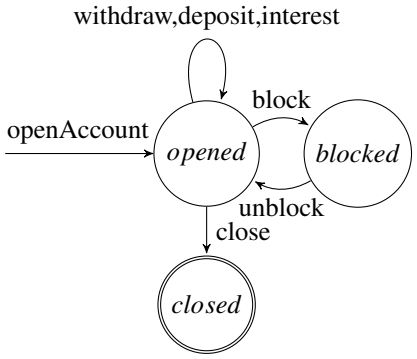

Figure 2. Graphical representation of the life cycle of the `SimpleSavings` example shown in Fig. 1

should be read as a transition from s_1 to s_2 is possible via any event in $e_1, ..., e_n$. For instance, it is possible to `block` an account that has been `opened`, which changes the state of the savings account to state `blocked`. A transition fires if and only if it is enabled by the life cycle and its conditions are satisfied (see section 4.3). The **`initial`** and **`final`** keywords mark the initial and final states of the machine.

Finally, the **`invariants`** defined for an entity capture predicates that always have to be true. For instance, it should not be possible to overdraw from the savings account, given the invariant `mustBePositive`; it is up to the conditions of the `withdraw` event to satisfy this invariant.

```
1  event openAccount[minimalDeposit: Money = EUR 0.00]
2    (accountNumber: IBAN, initialDeposit : Money) {
3
4    preconditions {
5      initialDeposit >= minimalDeposit;
6    }
7
8    postconditions {
9      new this.balance == initialDeposit;
10     new this.accountNumber == accountNumber;
11   }
12 }
```

Figure 3. Definition of the `openAccount` event

4.3 Declaring Events and Invariants

Note that Fig. 1 does not show the definition of the pre- and post conditions and invariants. These are specified elsewhere to promote reuse of events and invariants for specifying other, similar business entities, and for making Rebel specifications more concise. As an example, consider a possible definition of the `openAccount` event, shown in Fig. 3. Event definitions have two sets of parameters: configuration parameters (enclosed in `[...]`) and transition parameters (enclosed in `(...)`). Configuration parameters are bound at design time, and have default values. The default values can be overridden when an event is included in a specification. For instance, the configuration parameter `minimalDeposit` with the default value of `EUR 0.00` (Fig. 3, line 1) is bound in `SimpleSavings` (Fig. 1, line 7) to be `EUR 50.00`. With the use of these configuration parameters it is possible to reuse the `openAccount` event in the specification of other types of ING saving accounts.

Events can refer to entity fields using the keyword **`this`**. This notation is borrowed from object-oriented languages and refers to the current instance of the specification. The keyword **`new`** refers to the value of the variable in the post-state, after the transition has fired. This distinction is necessary to express logical and arithmetic constraints between the state of an instance before and after each transition. For instance, when withdrawing money the post-state of the balance state variable would be expressed using the current value (**`new`** `this.balance == this.balance - amount`). Semantically, all the constraints written in the preconditions must hold for the instance of the specification to make the transition and after the transition it is asserted that post-condition holds.

Invariants are also specified separately and they specify predicates that must be true at all times. For instance, the `mustBePositive` invariant exists to assert that the `balance` of the `SimpleSavings` is greater than or equal to `EUR 0.00` in all reachable states:

```
invariant mustBePostive { this.balance >= EUR 0.00 }
```

5. Simulation and Checking Specifications

Both simulation and checking share the the same underlying encoding strategy which will be explained in the next subsection. The subsections that follow will contain more details on the individual steps.

5.1 Overall SMT Encoding Strategy

We define the semantics of Rebel in terms of labeled transition systems (as introduced by Keller (Keller 1976) and popularized by Plotkin (Plotkin 1981)). The current state of the transition system for a given specification maps to a named state of the specification, tupled with the current field variable assignments and the event parameter assignments that led to the current state. The labeled transitions map to the events and their preconditions and postconditions. The invariants are used as external specifications of expected behavior and are simply mapped to additionally asserted formulas.

Labeled transitions systems can be checked using bounded model checking (Veanes et al. 2009). We use an encoding of symbolic bounded model checking (with data) as an SMT problem inspired by Milicevic (Milicevic and Kugler 2011) and Veanes et al. (Veanes et al. 2009). The goal of bounded model checking is to find a reachable state in which some property of interest does not hold (e.g. a state in which some invariant does not hold). We use the bound, encoded by k, as a parameter to balance efficiency and response time in the ISE with completeness of the check and we use it also to explore the state-space in a breadth-first manner for simulation purposes.

In bounded model checking, the *initial state* s_0 is constrained by some function: θ. For Rebel the semantics of the function θ constrains the initial state to represent an uninitialized specification.

Next, the *transition function* that constrains the valid transition from one state s_{i-1} to the next s_i is captured by $\rho(s_{i-1}, s_i)$. This means that in a valid trace—a chain of valid transitions from one state to the next—the following formula must hold: $\rho(s_0, s_1) \wedge \rho(s_1, s_2) \wedge ...\rho(s_{k-1}, s_k)$. For Rebel the semantics of the transition function is the exclusive disjunction of all defined events. For instance, a `SimpleSavings` specification can only transition via `withdraw` $\oplus$ `deposit` but never both. This means that a state transition in a Rebel specification is always constrained by only one of the defined events.

Like mentioned earlier, the goal of bounded model checking is to find a reachable state in which some property of interest does not hold. This property is also known as the *safety property* and captured by the function P. We are interested in a trace in which the safety property holds in all states except for the last. More formally, the following should hold: $P(s_0) \wedge P(s_1) \wedge ... \neg P(s_k)$. In the case of Rebel we are interested in finding states where the invariants do not hold. The safety property is defined as the conjunction of all defined invariants since a specification can contain multiple invariants.

So, using the above mapping of Rebel to SMT formulas we may verify—up to transition traces of length k—that the specified invariants hold during the entire life cycle of any entity. We may use the same mapping to infer single transition steps to run a simulator. In this case the SMT solver provides us with computations which satisfy the route from pre-condition to post-condition for every transition.

Effectively, the SMT solver has then become an interpreter for Rebel specifications.

In the coming sections we will give a more detailed description of the steps that are used when translating a Rebel specification to SMT constraints.

5.2 Normalization

Before we simulate or check a Rebel specification it is normalized. Normalization of the Rebel specification is not only done to make the mapping to SMT formulas easier; it also partially gives semantics. After normalization a specification contains all the necessary information for the aforementioned mapping to SMT. It consists of the following steps:

1. **Inlining.** Referenced events with their configurations and invariants are resolved and inlined with the specification.
2. **Desugaring the Life Cycle.** The life cycle is desugared by strengthening the pre- and postconditions of the events with the life cycle information. To achieve this two fields, `_state` and `_step`, are added to the fields of the specification. A distinct identity is assigned to each state and event. The `_state` field holds the identity of the current state and the `_step` field holds the identity of the event that led to the current state. This way the original life cycle can be expressed by adding constraints based on the two newly added fields to the pre- and postconditions of the events.
3. **Adding Frame Conditions.** To guard the fields that are not changed by the event frame conditions are added (Jackson 2012). These frame conditions make sure that a field has the same value after the transition as before.

5.3 Bounded Checking

The goal of checking Rebel specifications is to check whether a given specification is consistent. A specification is considered consistent if the invariants hold in all reachable states. A reachable state is a state which can be reached from the initial state via (a chain of) valid transitions. Since Rebel specifications have data encapsulated and life cycles may have loops model checking without reasonable bounds could quickly suffer from the state explosion problem (Clarke et al. 2009). Here we describe two verification techniques we implemented for Rebel (both use a similar encoding).

Step 1: Quickly Check if the Specification is (trivially) Consistent. First we use the SMT solver to try to inductively prove that the invariants hold in all possible transitions. This is expressed using these three formulas:

1. If the initial condition holds in some state then the safety property should also hold: $\theta(s_0) \Rightarrow P(s_0)$
2. If the initial condition holds in the first state and there is a transitions possible to a second state then the safety property should also hold in the second state: $\theta(s_{i-1}) \wedge \rho(s_{i-1}, s_i) \Rightarrow P(s_i)$
3. If the safety property holds in the first state and there is a transition possible to a second state then the safety property should also hold in the second state: $P(s_{i-1}) \wedge \rho(s_{i-1}, s_i) \Rightarrow P(s_i)$

If these three formulas can be proven by the SMT solver it means that the specification is consistent. In this case we report back to the user that the specification is found to be consistent. If they can not be proven it means that there might be a transition possible which leads to a state in which the safety property does hold.

This strategy can lead to false positives—a specification which is wrongly labeled as inconsistent—but never to false negatives. For instance: if the third hypothesis can not be proven it means that it is possible to construct a state in which the invariants hold, make a valid transition and end up in a state in which the invariants do not hold. However, whether the first state of the counter example is reachable from the initial state is unknown making it a potentially unreachable counter example. To find out if the counter example is actually reachable we run a bounded model check on the specification.

Step 2: Run Bounded Analysis. During bounded analysis we are interested in finding the smallest possible counter example, where smallest means in the least possible steps. The formulas that we try to prove are similar to those described in the previous section but the difference is that we now use explicit step unwinding.

The process of finding a counter example is fully automatic and incremental. We start by checking if an invalid state can be reached in one step. If not then we check if it can be reached in two steps. This is continued until a counter example is found or some k is reached [1]. More formally, we try to prove that: $\theta(s_0) \wedge P(s_0) \wedge \rho(s_0, s_1) \wedge P(s_1) \wedge ... \rho(s_{k-1}, s_k) \wedge \neg P(s_k)$.

If a counter example is found by the solver the found model is translated back to the Rebel simulator which can then visualize the counter example as a trace in the simulation environment. If no counter example can be found in the given k the ISE reports to the user that the specification might be consistent. The phrasing 'might' is used since it still could be the case that a counterexample can be found after n steps where $n > k$.

5.4 Simulation

The purpose of simulation differs from checking. Where checking is done to check the internal consistency, simulation is used to check the external consistency (Snoeck et al. 2003). Fig. 4 shows a screenshot of the implemented simulator in the Eclipse IDE. Using the simulator the user can quickly check whether the created specification behaves as (informally) expected. Where checking is about reasoning about all possible traces, simulation is about reasoning about

[1] In our implementation we have chosen to work with a configurable timeout given to the SMT solver instead of some fixed k. This choice is practical by nature, we want to control the maximum time spent waiting by the user.

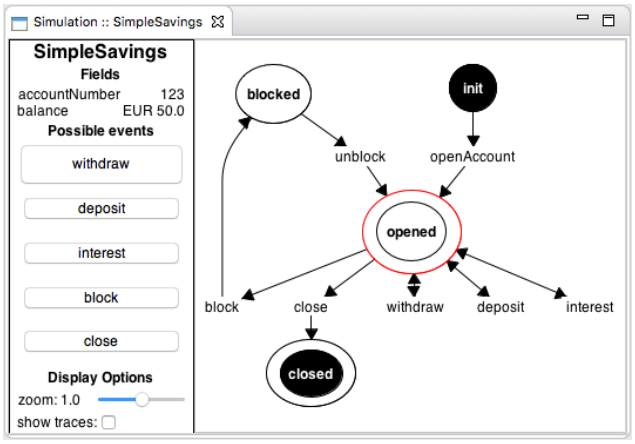

Figure 4. Screenshot of the Rebel simulator. The right side shows a graphical representation of the life cycle. The current state is highlighted by a red ellipse. The top left side contains the current values assigned to the fields, and the bottom part displays buttons for each possible transition that can be fired from the current state.

individual steps. This can be implemented using the similar strategy as described in the previous section. As mentioned earlier, when simulating we effectively use the SMT solver and our encoding as an interpreter of Rebel specifications.

Translate a Single Step to SMT. We translate the event the user wants to execute to SMT formulas. The transition function, $\rho(s_1, s_2)$, contains the pre- and postconditions of the to be executed event. The current values of the simulated specification are translated as constraints on the current state, s_1, and the user is asked to provide the data values for the transition parameters of the chosen event transition.

Next, the solver is asked whether it can indeed 'make the step'. This means that we check two things:

1. Whether it is possible to satisfy the constraints of the selected events given the current state and actuals of the transition parameters of the event: $\rho(s_1, s_2)$
2. Whether the invariants (the safety property) hold in the resulting state: $P(s_2)$.

If it can not make the step because the first check fails we make use of the *unsatisfiability core* functionality of the solver to find out which constraints are most likely the cause of the failure. The unsatisfiability core functionality reports an unsatisfiable subset of clauses of the asserted formulas (Cimatti et al. 2011). The returned constraints are mapped back to the original Rebel expressions and presented to the user. If it can not make the step because the second check fails we report back to the user which invariant is violated as a result of the step, and we roll back to the previous state so that the user can explore other options without having to restart the simulation.

6. Performing Model Based Testing of Existing Applications

Another important aspect of the Rebel ISE is the ability to check whether an existing application behaves according to the specification. Mismatches between specification and the system under test (SUT) can point to bugs in the existing applications or to erroneous assumptions in the specification. This is essentially a model based testing approach to test existing ING applications (Dalal et al. 1999).

To implement this functionality we use the traces that were described in Section 5. By automatically checking whether the SUT accepts the trace as a valid execution trace we can check whether the SUT behaves similar as the specification. To playback the steps in a trace on the SUT every transition is split into three steps:

1. Check whether the current state of the SUT conforms to the current state in the trace (pre-transition check)
2. Ask the SUT to perform the actual operation according to the trace (transition check)
3. Check whether the new state of the SUT conforms to the new state in the trace (post-transition check)

To implement the above steps we map every event declared in the specification to an operation in the SUT (to perform the transition check). Next to that we map fields of the specification onto the state of the SUT (to perform the pre- and post-transition checks). A requirement on these mappings is that all events and fields of the specification are completely mapped onto the SUT (otherwise the trace effectively can not be played back). Our first prototype used SOAP services provided by the SUT and performed the operations by sending SOAP messages and by checking whether the received responses were conform to the trace.

Currently, it is possible to perform the described testing interactively using the simulation described in Section 5.4. Every step made in the simulation is automatically also performed in the configured SUT. Any difference between the simulation and the SUT is then displayed in the simulation showing which values differ between the two. Expanding this functionality to also work completely automatically using a given trace is future work but should be straightforward to implement.

7. Applying Rebel inside the Bank

We implemented a prototype of the Rebel ISE and tested its implementation inside the bank. As a first test case we specified saving accounts (Peters 2014). Fourteen (out of seventeen) types of ING saving accounts where specified. These saving accounts were build up out of fifteen distinct events. With the use of the configuration parameters these events could be reused across different types of savings accounts.

The verification tool showed an unexpected counter example where the `mustBePositive` invariant would not hold,

caused by the —unlikely but real— possible circumstance of a negative interest rate.

Using the prototype of the model based testing tool we found a difference between the specification and an existing system where the existing system allowed for accounts with incorrect account numbers.

During the use of Rebel inside the bank we observed that our initial assumption that Rebel syntax would be understandable for product owners was incorrect. We informally evaluated the understandability of Rebel specifications by asking a handful of product owners whether they understood the specification. Some had more trouble than others in performing this task. When faced with a manually written document containing similar specifications and visualized using UML Statechart diagrams the same product owners were able to understand the specifications. This led us to develop transformations from Rebel specification to both natural language documents and interactive UML Statechart visualizations to increase the ease of understanding amongst product owners. A more thorough evaluation on the understandability of Rebel specification is left as future work.

8. Related Work

Formal Specification Languages. Rebel is inspired by other formal methods. We will discuss Alloy (Jackson 2002) and B (Abrail 1996) since Rebel has similarities with both.

Alloy is a specification languages based on relational logic. Alloy is positioned as a lightweight formal method (Jackson 2001) meaning that instead of demanding rigorous proofs of the specifications it uses the bounded model finder Kodkod (Torlak and Dennis 2006) to analyze the specification and gives counter examples if the assertions made in the specifications do not hold. Rebel is also a lightweight formal method in the sense that it uses similar bounded analysis of its specifications. Unlike Alloy, Rebel does allow other theories to be used next to relational logic. For instance, a defined Rebel parameter can be of type `Integer` allowing for all the usual arithmetic expressions to be used in the pre- and postconditions. Next to this Rebel and Alloy handle state differently. Alloy is a general purpose specification language allowing for different modeling paradigms. It is possible to model state based systems in Alloy but this is not a builtin, meaning that users should take special care when modeling state based systems. In Rebel State is a first class concern making it hard for other modeling paradigms to be used.

B is a formal method in which abstract machines play a central role (Abrail 1996). It uses first order logic and set theory to define operations on abstract machines. B was built with code generation in mind. To achieve this B uses a process called specification refinement. In every new refinement step the user adds more detail to the specification. The last level of refinement is the actual code which can be executed. On every refinement level B requires users to provide proof that the refinement is correct. Most of these proofs can be obtained automatically but, if not, they must be provided by the user. This proof obligation is the biggest difference with Rebel. Rebel follows the same philosophy as Alloy. Requiring full proofs can be experienced as 'too heavy' by our intended users.

Enterprise Modeling. Modeling enterprise systems is a well known topic in both research and industry (i.e. (Davies et al. 2014)). There are many approaches, but here we highlight only one, MERODE, since it has a unique formal analysis component. MERODE is a domain modeling approach for enterprise systems (Snoeck et al. 2003). MERODE defines static entity-relationship (ER) models and a dynamic model based on a process algebra. By combining the notion of existence dependency (Snoeck and Dedene 1998) in the ER model, with the process algebra of the dynamic model, an enterprise model can be checked for deadlocks (Dedene and Snoeck 1995). Although MERODE offers some formal analysis techniques it was not build with verification in mind. For instance, MERODE allows for the definition of (class, attribute and method) constraints (in an OCL like syntax) but there is no method to check whether certain assertions will hold. It is left to the user to transform these constraints to meaningful implementations using model-to-source transformations.

DSLs and Finance. Domain-specific languages (DSLs) have a long history in the domain of finance[2]. One of the earliest financial DSLs is RISLA (Arnold et al. 1995). The language was designed to capture the nature of interest products offered by banks. One of the findings of the authors was that financial engineering was extremely suitable as an area to apply formal methods, because financial damage inflicted by incorrect system behavior can be very severe.

The difference with Rebel and other financial DSLs is the scope of the problem domain that is covered by the DSL. While Rebel focusses on the whole of the banking enterprise, other financial DSLs, like RISLA, are specifically created to work on one specific financial problem domain.

Formal Methods and Finance. Gimblett, Roggenback and Schlingloff present a case-study on how to formally specify the international Electronic Payment System (ep2) standard in CSP-CASL (Gimblett et al. 2004). With the formal specification they were able to identify a number of deficiencies in the standard. As a source for the formalization they used the informal documentation of the ep2 standard which was mostly text-based, augmented with UML-like diagrams. They found that it was easy to formalize high level descriptions but when it came to the details the standard was often found lacking.

Gimblett, Roggenback and Schlingloff show that there is value in formally specifying financial standards. It is not their aim to provide a method for specifying these types of standards. With Rebel we aim to provide such a method.

[2] See (Christerson et al.) for an overview

9. Conclusion

In this paper we presented the Rebel language and its ISE, an integrated specification environment for defining financial enterprise systems. With the use of the Rebel ISE we were able to formally specify banking products like saving accounts. By using a mapping of the Rebel language to SMT formulas it is possible to simulate and check Rebel specifications. Simulation is useful for checking the external correctness of the specification ('does the product behave as I expected') and checking is useful to check the internal correctness of the specification ('do the specified invariants hold'). The mapping to SMT uses the same strategy for both simulation and checking using a bounded model checking encoding of the Rebel specifications.

Our first impression of the use of Rebel inside the bank is that formal specifications help in translating vague and ambiguous product description in precise product specifications. Simulation helps as an early prototyping mechanism with which users can verify whether the specified product is complete regarding its functionality. Checking helps users verifying internal consistency. It does not allow for traces where the system ends up in a state in which the defined invariants do not hold.

Next to this we observed that transforming the specifications into documentation that closely resembles the current documents that are written by hand seems to help understanding. As future work we would like to more thoroughly evaluate the understandability of Rebel specifications so we can further improve the communication between stakeholders using these specifications.

Based on these initial results we are now further investigating the optimization of the model checking and simulation processes, adding features of (parallel) composition of entities and communication between entities from Rebel specifications. Meanwhile, the bank has invested to produce more Rebel specifications of their products and services as they already see the benefit of having an unambiguous product specification as a method of communication.

References

J. Abrail. *The B-Book: Assigning programs to meaning*. Cambridge University Press, 1996.

B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, 1995.

M. Christerson, D. Frankel, and T. Schiller. Financial domain-specific language listing. `http://www.dslfin.org/resources.html`. Accessed: 4-8-2016.

A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 40:701–718, 2011.

E. Clarke, E. Emerson, and J. Sifakis. Model Checking : Algorithmic Verification and Debugging. *Communications of the ACM*, 52 (11):74–84, 2009.

S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz. Model-based testing in practice. *Proceedings of the 1999 International Conference on Software Engineering*, 1999 (May):285–294, 1999.

J. Davies, J. Gibbons, J. Welch, and E. Crichton. Model-driven engineering of information systems: 10 years and 1000 versions. *Science of Computer Programming*, 89:88–104, sep 2014.

G. Dedene and M. Snoeck. Formal deadlock elimination in an object oriented conceptual schema. *Data & Knowledge Engineering*, 15 (1):1–30, 1995.

A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of electronic payment systems in CSP-CASL. In *Recent Trends in Algebraic Development Techniques*, pages 61–78. Springer, 2004.

D. Jackson. Lightweight formal methods. In *FME 2001: Formal Methods for Increasing Software Productivity*, pages 1–1. Springer, 2001.

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

D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT press, revised edition, 2012.

R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

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.

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

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

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

C. Peters. FORS - Seperating Configuration From Formal Specification. Master's thesis, Universisty of Amsterdam, 2014.

G. D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Dept., Aarhus University, Denmark, 1981.

M. Snoeck and G. Dedene. Existence dependency: The key to semantic integrity between structural and behavioral aspects of object types. *Software Engineering, IEEE Transactions on*, 24 (4):233–251, 1998.

M. Snoeck, C. Michiels, and G. Dedene. Consistency by construction: the case of MERODE. In *International Conference on Conceptual Modeling*, pages 105–117. Springer, 2003.

P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science)*. Springer, 2004.

E. Torlak and G. Dennis. Kodkod for Alloy users. In *First ACM Alloy Workshop, Portland, Oregon*. ACM, 2006.

M. Veanes, N. Bjørner, Y. Gurevich, and W. Schulte. Symbolic bounded model checking of abstract state machines. *Int J Software Informatics*, 3:149–170, 2009.