Modeling with Mocking

Jouke Stoel*, Tijs van der Storm*† and Jurgen Vinju*‡
*CWI, Amsterstam, †University of Groningen, Groningen, ‡TU/e, Eindhoven,
The Netherlands
{j.stoel, t.van.der.storm, jurgen.vinju}@cwi.nl

***Abstract*—Writing formal specifications often requires users to abstract from the original problem. Especially when verification techniques such as model checking are used. Without applying abstraction the search space the model checker need to traverse tends to grow quickly beyond the scope of what can be checked within reasonable time.**

The downside of this need to omit details is that it increases the distance to the implementation. Ideally, the created specifications could be used to generate software from (either manually or automatically). But having an incomplete description of the desired system is not enough for this purpose.

In this work we introduce the REBEL2 specification language. REBEL2 lets the user write full system specifications in the form of state machines with data without the need to apply abstraction while still preserving the ability to verify non-trivial properties. This is done by allowing the user to *forget* and *mock* specifications when running the model checker. The original specifications are untouched by these techniques.

We compare the expressiveness of REBEL2 and the effectiveness of *mock* and *forget* by implementing two case studies: one from the automotive domain and one from the banking domain. We find that REBEL2 is expressive enough to implement both case studies in a concise manner. Next to that, when performing checks in isolation, mocking can speed up model checking significantly.

***Index Terms*—formal specifications, bounded model checking, compositional reasoning, smt solving, mocking, lightweight formal method**

I. Introduction

Formal specifications combined with model checking have been shown to be very effective in capturing and verifying desired system behavior. However when applying model checking, the user is forced to think about the potential state space the model checker needs to traverse [1]. Not taking this into sufficient consideration will lead to a state space that is too large to check within reasonable time.

Applying *abstraction* is one of the most used techniques to overcome the state space explosion problem [2], [3]. Coming up with an abstraction that models the problem in sufficient detail but is abstract enough such that it can be model checked is in most cases left up to the specifier. Because of this need of omitting details the specifications become too abstract. Translating such abstract specifications to code, either automatically or manually, becomes very difficult since the software that ultimately needs to run needs to contain all the details.

The methods that do allow to specify systems up till the level of executable code such as Event-B [4] often require proofs that show that a refinement of an abstract specification is indeed true to its abstraction. Writing such specifications and proofs can increase the required effort both in time and expertise [5]. This effort can be considered as too high and might be one of the reasons that industry adoption of these techniques, aside from safety critical systems, is as of yet still low.

In this work we approach this problem from another angle, by combining formal specifications, model checking and *mocking*. Mocking is a well known testing technique from object-oriented programming where mock objects are created to replace domain code with a dummy implementation for the purpose of emulating its behavior [6]. These mock objects are passed to the objects under test to allow for the testing of features in isolation with respect to the rest of the system. Similarly, we propose to use *mock specifications* to replace specified behavior with a dummy specification for the purpose of model checking. This allows us to perform model checks for parts of the system, isolated from other specifications. Although this compositional reasoning technique is inherently unsound, it can still be of value. It allows the user to make a pragmatic tradeoff between completeness of the check versus timely feedback from the checker. This type of trade-off is not new. For instance, the lightweight formal methods tool Alloy is based around the "small-scope" hypothesis which says "many bugs are found in a small scope" favoring partiality over completeness as well, although in a different dimension [7].

We implement this mocking technique in REBEL2, a lightweight formal specification method.[1] REBEL2 specifications use the notation of state machines with data and guarded transitions. Assumptions and assertions can be expressed using Linear Temporal Logic (LTL). To check assertions REBEL2 uses a bounded model checking technique. This is realized by translating the specifications to the relational algebra of ALLEALLE [9] which in turn translates it to SMT and uses the Z3 SMT solver [10].

When model checking, the user can mock parts of the specifications by using two language constructs: `forget` and `mock`. The `forget` construct slices out data and constraints on this data. The `mock` construct replaces a specification entirely with a mock specification. Like mock objects, mock specifications can emulate parts of the original specification in isolation, which in turn can potentially reduce the state space considerately. Although we introduce the constructs `forget` and `mock` in

[1]The term *lightweight formal method* was coined by Daniel Jackson in 2001 [8]. It describes methods with emphasis on partiality (partiality in language, modeling, analysis and composition).

```
spec Account
  nr: AccountNumber, balance: Integer, openedOn: Date;

  init event open(nr: AccountNumber, openedOn: Date)
    post: this.balance' = 0, this.nr' = nr,
          this.openedOn' = openedOn;

  event deposit(amount: Integer)
    pre:  amount > 0;
    post: this.balance' = this.balance + amount;

  event withdraw(amount: Integer)
    pre: amount > 0, this.balance >= amount;
    post: this.balance' = this.balance - amount;

  event payInterest(rate: Integer)
    post: this.balance' =
            this.balance + ((this.balance * rate) / 100);

  final event close()
    pre: this.balance = 0;

  event block()
  event unblock()
  final event forceClose()

  states:
    (*)        -> activation: open;
    activation -> opened: deposit;
    opened     -> opened: deposit, withdraw, payInterest;
    opened     -> blocked: block;
    blocked    -> opened: unblock;
    blocked    -> (*): forceClose;
    opened     -> (*): close;
```

Listing 1: REBEL2 specification of an `Account`.

the context of the REBEL2 specification language, the ideas are general and could be implemented in other state-based techniques as well.

To test the expressiveness of REBEL2 and the effectiveness of model checking with mocking we evaluate REBEL2 on two different case studies, one from the automotive domain and one from the financial domain.

To summarize, the contributions of our work are:

1) A description of the lightweight formal specification language REBEL2 by example (Section II).
2) A formalization of the **forget** and **mock** constructs (Section III).
3) A prototype implementation of REBEL2 and model checking with mocking using **forget** and **mock** (Section IV).
4) An evaluation of the REBEL2 language and the forget and mock constructs both in terms of expressiveness and effectiveness (Section V).

We conclude our paper with a discussion of related work (Section VI) and future work (Section VII)

II. REBEL2 BY EXAMPLE: MONEY TRANSFER

In this section we introduce REBEL2 by specifying a simple bank account state machine. REBEL2 is inspired by earlier work. [11].

Accounts can be opened, and after an initial deposit, any number of deposit, withdraw and pay interest events are possible. An account can be temporarily blocked (e.g., in case of suspicious transactions) and unblocked. When an account

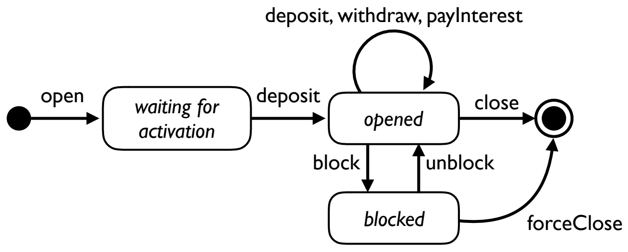


Fig. 1: UML Statechart visualization of an `Account`

is blocked, no transactions are allowed. Eventually an account can be closed either normally or by force. Figure 1 shows a visual representation of the above rules.

Listing 1 shows the REBEL2 specification of an account that complies with the rules stated above. A REBEL2 specification consists of four parts, *fields*, *events*, *states*, and *assumptions*. We explain the first three below. Assumptions are discussed further-on in the section.

Fields: Fields represent the internal state of a state machine. The `Account` specification declares three fields: `nr` , `balance` and `openedOn`. Fields can have primitive types, e.g., `balance` (of type `Integer`). But fields may also refer to other REBEL2 specifications as a type, as is the case with the `nr` field, which has type `AccountNumber`.

Events: Events define the business events and actions that may be triggered on a state machine. In the `Account` example there are eight. Events may have formal parameters (e.g., `amount` in `deposit` and `withdraw`), and are (optionally) guarded by preconditions. The effect of an event is specified in the form of a postcondition where the next value of a field is accessible by priming its name. For instance, the effect of `withdraw` is defined as `this.balance' = this.balance - amount`, effectively decrementing the account's balance.

States: The last section in Listing 1 defines the lifecycle of an account, by defining state transitions of the form "*from* `->` *to*: *via*,...", where *via* is a list of events declared earlier. The special marker `(*)` is used to indicate initial and final states. Events from the initial state need to be marked as initial (cf. `init event open`), and events to the final state have to be marked `final` (cf. `close` and `forceClose`).

Checking assertions: Now that we have specified our `Account` we can validate its behavior by checking the safety property that an account can not be overdrawn. For this, REBEL2 supports assertions. The above property is expressed as follows:

```
assert CantOverdrawAccount = forall ac:Account |
  always (ac is initialized => ac.balance >= 0);
```

Assertions are expressed using Linear Temporal Logic (LTL) expressions. REBEL2 supports the standard LTL operators **always**, **eventually**, **next** and **until**. The `CantOverdrawAccount` assertion can thus be read as follows: for all possible execution paths, all initialized accounts have a non-negative balance.

REBEL2 uses bounded model checking. To check the above property it is required to specify the bound in terms of i) the number of instances (e.g., `Account` objects) that take part and ii) the maximum search depth.

The number of instances is specified using the **`config`** directive:

```
config Basic = ac: Account is uninitialized,
               aNr: AccountNumber, dt: Date;
```

A configuration defines all the specification instances that can participate during model checking. This configuration specifies that there are three instances: an `Account`, an `AccountNumber` and a `Date`. All instances in a configuration are bound to a label (i.e. `Account` is bound to `ac`, `AccountNumber` to `aNr` and `Date` to `dt`). The **`config`** statement supports constraining the state and field values of an instance. In the example the required state of the `Account` instance `ac` is set to **`uninitialized`**. The state of the `AccountNumber` and `Date` instances are not specified and thus are left open for the underlying model checker to decide.

The maximum search-depth is specified when invoking the verifier through the **`check`** command:

```
check CantOverdrawAccount from Basic in max 10 steps;
```

This instructs the model checker to try and find a counter-example to the property `CantOverdrawAccount`, starting in the `Basic` configuration, with a maximum search depth of 10 consecutively triggered events.

Forget: Running the model checker on the above **`check`** command, results in a time-out, because the `AccountNumber` and `Date` specifications (not shown here) are complex state machines. As a result the state space that the checker must traverse is too large to find a counter example within the default 30 second time-out.

The **`forget`** modifier can be used to abstract from the `nr` and `openedOn` fields in the `Basic` configuration, as follows:

```
config Basic = ac: Account forget nr, openedOn
  is uninitialized;
```

The result is that the field definitions and all constraints referencing these fields are removed resulting in a smaller (but well-formed) specification. Since the fields that reference the `AccountNumber` and `Date` specifications have been removed, the instances of these specifications can also be removed from the configuration (cf. `aNr` and `dt`).

Running the model checker again now results in a trace, a counter example for our assertion. The trace shows an execution path for which the assertion does not hold. In other words, it is possible to overdraw the account according to the specification. The following execution trace is shown: [2]

```
Counter example found:
1 (INIT): ac (Account) is 'uninitialized' :
  --> Raised open() on ac (Account)
2: ac (Account) is 'activation' : balance = 0
  --> Raised deposit(amount = 1) on ac (Account)
3: ac (Account) is 'opened' : balance = 1
  --> Raised payInterest(rate = -101) on ac (Account)
4 (GOAL): ac (Account) is 'opened' : balance = -1
```

[2]Traces can be shown both textually and visually. In this paper they are listed in their textual notation.

```
spec Transaction
  frm: Account, to: Account, amount: Integer;

  init event create(frm: Account, to: Account, amt: Integer)
    pre:  frm != to, amt > 0;
    post: this.frm' = frm, this.to' = to,
          this.amount' = amt;

  final event book()
    pre:  this.frm.withdraw(this.amount),
          this.to.deposit(this.amount);

  states:
    (*) -> created: create;
    created -> (*): book;
```

Listing 2: Specification of a Money Transaction.

Examining the trace shows the root of the problem: the `rate` parameter of the `payInterest` action can be negative (see the third step in the trace). A way to prevent this is by adding the constraint `rate >= 0` to the precondition of the `payInterest` event:

```
event payInterest(rate: Integer)
  pre: rate >= 0;
  post: this.balance' =
    this.balance + ((this.balance * rate) / 100);
```

Rerunning the model checker after this fix yields the desired result: no counter example is found given the specified configuration and search depth.

Synchronization: To illustrate synchronization between state machines, Listing 2 shows the specification of a `Transaction` entity which captures a transfer of money between two accounts. This is modeled in the `book` event, which triggers the `withdraw` and `deposit` events on the `frm` and `to` accounts, respectively. Semantically, all three events happen as a single atomic step: either all three succeed, or none.

To check whether it is possible to perform such a booking a *simulation* is run. The difference between a check and a simulation is that the model checker is not instructed to look for a counter example, but to find a witness of the assertion of interest instead.

Here's the assertion of interest:

```
assert CanBookATransaction =
  exists t: Transaction | eventually book on t;
```

The assertion `CanBookATransaction` states that at some point in time there exists a `Transaction` on which the event `book` has been triggered.

Just like with **`check`**, a configuration specifies the elements participating:

```
config BasicTrans = t: Transaction is uninitialized,
  ac1,ac2: Account, an1,an2: AccountNumber,
  d1,d2: Date;
```

We use the **`run`** command to have the model checker find a witness:

```
run CanBookATransaction from BasicTrans in max 5 steps;
```

Executing this command causes a time-out because, like before, the state space is too large to check due to the inclusion of the detailed `AccountNumber` and `Date` specifications. Instead of

```
spec MockAccount
  balance: Integer;

  internal event withdraw(amount: Integer)
    pre: amount > 0;
    post: this.balance' = this.balance - amount;

  internal event deposit(amount: Integer)
    pre: amount > 0;
    post: this.balance' = this.balance + amount;

  assume PositiveBalance =
    always forall a:MockAccount | a.balance >= 0;

  states:
    opened -> opened: withdraw, deposit;
```

Listing 3: A mock specification of the `Account` of Listing 1

slicing out fields from the participating instances (using **`forget`**), we want to keep the interaction between the `Transaction` and the two accounts in place since this is the essence of a transaction. To realize this, REBEL2 offers the **`mock`** operator to substitute simpler entities for certain instances in a configuration.

Mocking: Similar to mock classes in object-oriented programming, mocking in REBEL2 entails writing a compatible specification that acts as a drop-in replacement for another specification. A potential mock specification of `Account` is shown in Listing 3.

This `MockAccount` contains two new concepts, **`internal`** events, and **`assume`**. The **`internal`** modifier signals to the model checker, that the event can not be triggered in isolation, but it can occur as part of a synchronizing event, like `book` in `Transaction`. Assumptions are invariants that the model checker assumes to always hold, expressed using the same LTL and FO formulas used in assertions. For instance, the `PositiveBalance` assumption in `MockAccount` allows the model checker to assume that balance is always non-negative, an assumption that we have checked earlier on actual accounts.

Mock specifications must be *compatible* with the mocked specification in that it needs to support the same events (including their signature) as the original, *restricted to* the events that are potentially triggered by the check. For instance, `MockAccount` is a valid mock specification for `Account` because it supports both events which can be triggered by the `book` event of `Transaction`, namely `withdraw` and `deposit`.

The mock specification can now be used in the definition of a configuration and **`run`** invocation:

```
config SimplifiedTrans =
  t: Transaction is uninitialized,
  ac1,ac2: MockAccount mocks Account;

run CanBookATransaction from SimplifiedTrans in max 5 steps;
```

Running the model checker returns the following witness showing a trace where a `Transaction` is booked:

```
Witness found:
1 (INIT): ac1 (Account) is `opened` : balance = 7
  ac2 (Account) is `opened` : balance = 1
  t (Transaction) is `uninitialized` :
  --> Raised create(from = ac1, to = ac2, amount = 3) on t
     (Transaction)
2: ac1 (Account) is `opened` : balance = 7
   ac2 (Account) is `opened` : balance = 1
   t (Transaction) is `created` : from = ac1, to = ac2,
     amount = 3
  --> Raised book() on t (Transaction) : affected instances
      {t,ac2,ac1}
3 (GOAL): ac1 (Account) is `opened` : balance = 4
  ac2 (Account) is `opened` : balance = 4
  t (Transaction) is `finalized`
```

Unsoundness: In this section we have introduced the formal modeling language REBEL2 and shown how the **`forget`** and **`mock`** constructs can be used to check and simulate properties of interest. Note, however, that both **`forget`** and **`mock`** are *unsound*: neither construct guarantees that the abstractions they create are equivalent with the original specification. This is by design. However, just like the "small scope" assumption used in tools such as Alloy, we conjecture that, nevertheless, it is possible and convenient to check non-trivial, useful properties, in limited amounts of (solving) time. This gives the user additional flexibility and freedom in defining checks and simulations, without immediately running into time-outs. As such, **`forget`** and **`mock`** introduce a pragmatic middle-ground between full formal verification and traditional testing as is practiced in many organizations.

III. FORMALIZATION

To define the semantics of **`forget`** and **`mock`** we need to define the semantics of REBEL2 as a framework. For this we will use the logic proposed in *State / Event Linear Temporal Logic* (SE-LTL) [12]. This logic contains both the notion of states and events and operates over a *Labeled Kripke Structure* (LKS). An LKS is a 7-tuple $(S, Init, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ where S is a finite set of states, $Init \subseteq S$ is the set of *initial states*, P is a finite set of *atomic propositions*, $\mathcal{L} : S \to 2^P$ is a *state labeling function* from states to atomic propositions, $T \subseteq S \times S$ is the transition relation, Σ a finite set (or *alphabet*) of *events* and $\mathcal{E} : T \to \left(2^\Sigma \setminus \{\emptyset\}\right)$ the *transition labeling function*. We will write $s \xrightarrow{E} s'$ to denote a transition where $(s, s') \in T$ an $E \subseteq \mathcal{E}(s, s')$. If E is a singleton set we will just write $s \xrightarrow{e} s'$. The transition relation T is assumed to be *total* meaning that every state has a successor (no *deadlock* can occur).

A *path* $\pi = \langle s_1, e_1, s_2, e_2, s_3, \dots \rangle$ is an infinite, alternating sequence of states and events in which for each $i \geq 1, s_i$ and $s_{i+1} \in S, e_i \in \Sigma$ and $s_i \xrightarrow{e_i} s_{i+1} \in \mathcal{E}$. All paths together make up for the *language* of an LKS written as $L(M)$.

A. REBEL2 *Specification to LKS*

To map REBEL2 specifications to an LKS we use the following translation. The set of atomic propositions P contains all fields and possible values of a REBEL2 specification $\mathcal{R}$. For the sake of our formalization we will restrict the `Integer` and `String` domains to a bounded set of values where the bounds are arbitrarily chosen. This way we restrict the set of states S and the set of atomic propositions P to be finite. The `state` as defined in a REBEL2 specification is also considered part of the set of atomic propositions and should not be confused with the state set S of the LKS. Also the parameters of the defined events in $\mathcal{R}$ are considered as being part of the set of atomic propositions.

The state labeling function $\mathcal{L}$ maps values to fields for each possible $s \in S$. Σ contains all event labels as defined in $\mathcal{R}$. We derive $\mathcal{E}$ by calculating for each possible $s, s' \in S$ and event $e \in \Sigma$ the enabled events by checking whether the preconditions of e hold in $\mathcal{L}(s)$ and whether the postconditions hold in $\mathcal{L}(s')$.

B. Semantics of the Forget Operator

Our formalization of `forget` maps to the formalization of *abstraction* of an LKS as defined by Chaki et al. [12]. We will recall the notion of abstraction from [12] to show the meaning of `forget`.

Let $M = (S_M, Init_M, P_M, \mathcal{L}_M, T_M, \Sigma_M, \mathcal{E}_M)$ and $A = (S_A, Init_A, P_A, \mathcal{L}_A, T_A, \Sigma_A, \mathcal{E}_A)$. A is considered an abstraction of M (written $A \sqsubseteq M$) iff:

1) $P_A \subseteq P_M$.
2) $\Sigma_A = \Sigma_M$.
3) For every path $\pi = \langle s_1, e_1, s_2, e_2, \dots \rangle \in L(M)$ there exists a path $\pi' = \langle s'_1, e'_1, s'_2, \dots \rangle \in L(A)$ such that, for each $i \geq 1, e'_i = e_i$ and $\mathcal{L}_A(s'_i) = \mathcal{L}(s_i) \cap P_A$.

This is also known as *variable hiding* since an abstraction A contains a subset of the propositional variables of M while still accepting the original language of M.

This is also the essence of the `forget` operator. It will hide all atomic propositions bound to the field that it is instructed to forget (e.g., every reference that maps a value to the field is removed from the set P). All constraints in the pre- and postconditions of the event referencing the forgotten field can be considered to evaluate to *true* and thus can be removed.

C. Semantics of the Mock Operator

To define the meaning of the `mock` operator we must first define the notion of *parallel composition* as defined by Chaki et al. [12]. We recall it here for clarity.

Parallel composition is defined via *shared events*. It is not allowed to share variables between two LKSs. This facilitates the possibility to perform compositional reasoning.

Let $M_1 = (S_1, Init_1, P_1, \mathcal{L}_1, T_1, \Sigma_1, \mathcal{E}_1)$ and $M_2 = (S_2, Init_2, P_2, \mathcal{L}_2, T_2, \Sigma_2, \mathcal{E}_2)$ then two LKSs are considered compatible if (1) they do not share any variables: $S_1 \cap S_2 = P_1 \cap P_2 = \emptyset$, and (2) their parallel composition yields a total relation (so that no deadlock can occur). Thus, the parallel composition can be defined as: $M_1 \parallel M_2 = (S_1 \times S_2, Init_1 \times Init_2, P_1 \cup P_2, \mathcal{L}_1 \cup \mathcal{L}_2, T, \Sigma_1 \cup \Sigma_2, \mathcal{E})$ where $(\mathcal{L}_1 \cup \mathcal{L}_2)(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$ and T and $\mathcal{E}$ are such that $(s_1, s_2) \xrightarrow{E} (s'_1, s'_2)$ iff $E \neq \emptyset$ and one of the following holds:

1) $E \in \Sigma_1 \setminus \Sigma_2$ and $s_1 \xrightarrow{E} s'_1$ and $s_2 = s'_2$
2) $E \in \Sigma_2 \setminus \Sigma_1$ and $s_2 \xrightarrow{E} s'_2$ and $s_1 = s'_1$
3) $E \in \Sigma_1 \cap \Sigma_2$ and $s_1 \xrightarrow{E} s'_1$ and $s_2 \xrightarrow{E} s'_2$

To put it on other words, LKSs synchronize on shared events while proceeding independently from each other.

Communication between REBEL2 machines operates in the same manner. The difference is that REBEL2 allows users to define which events must synchronize instead of relying on the mechanism of shared labels. Shared event parameters become part of the local variables of both machines to maintain the separation of variables and allow for compositional reasoning. State and field queries (between two specifications) are also mapped to shared events.

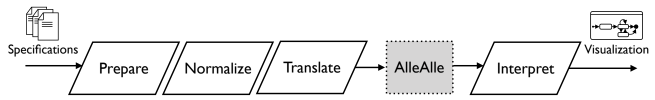


Fig. 2: Overview of the model checking processing pipeline for REBEL2 specifications. The white parallelograms are part of REBEL2. The gray box is an external process.

We also use this notion to define the `mock` operator. Assume we have two LKSs, M_1 and M_2, having the same signature as described earlier. Furthermore assume that M_1 and M_2 have shared events: $\Sigma_1 \cap \Sigma_2 \neq \emptyset$. An LKS $M_3 = (S_3, Init_3, P_3, \mathcal{L}_3, T, \Sigma_3, \mathcal{E})$ `mock` M_2 iff (1) the parallel composition of M_1 and M_3 is valid (no shared variables, composition yields a total relation) and (2) M_3 has the same shared events as M_1 and M_2: $\Sigma_1 \cap \Sigma_3 = \Sigma_1 \cap \Sigma_2$. This means that all events that were synchronized in the original composition, $M_1 \parallel M_2$ will synchronize in the abstracted composition $M_1 \parallel M_3$.

D. On the Logic of SE-LTL

Using the earlier definitions of LKS we use the definition of State/Event Linear Temporal Logic (SE-LTL) as defined by Chaki et al. [12]. This logic operates on both states and events and has the following syntax: $\phi = p \mid e \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\ \phi \mid \mathbf{G}\ \phi \mid \mathbf{F}\ \phi \mid \phi\ \mathbf{U}\ \phi$ where $p \in P$ and $e \in \Sigma$. The operators **X** (next), **G** (always), **F** (eventually) and **U** (until) have their usual semantics,

Besides propositional constraints REBEL2 also allows for first-order constraints like quantification (`forall`, `exists`) and relational operators like membership (`in`). Given that REBEL2 only operates in a bounded setting (i.e. a bounded number of machines and states) these operators can be translated to propositional logic via known translations (i.e. [9], [13], [14]).

IV. IMPLEMENTATION

Performing a REBEL2 `check` (or `run`) follows a pipeline described in Figure 2. This pipeline consists of four steps: *prepare*, *normalize*, *translate* and *interpret*. The general scheme is that we translate REBEL2 specifications to the relational algebra of ALLEALLE [9] which in turn translates its relational algebra to an SMT formula and calls the Z3 SMT solver [10] to check whether the formula is satisfiable. If it is satisfiable the result is interpreted back into the domain of REBEL2, via ALLEALLE, where it is presented as an interactive visualization or textual trace to the user. REBEL2[3], the language and the transformations needed for model checking, are all implemented using the Rascal Language Workbench [15]. We will discuss each step in more detail.

[3]See https://github.com/cwi-swat/rebel2/releases/tag/jan-2021.

A. Step 1: Preparation

The first step of the pipeline collects only those specifications which are needed to check (or run) an assertion. As an example we will use the **check** that was formulated earlier which we recall here for readability:

```
check CantOverdrawAccount from Basic in max 10 steps;
```

This **check** references the `CantOverdrawAccount` assertion and the `Basic` configuration. The `Account` specification references the `AccountNumber` and `Date` specifications. To be able to perform checks on a account we must therefore at least also include the `AccountNumber` and `Date` specifications. Therefore the referenced `Basic` configuration is declared as:

```
config Basic = ac: Account is uninitialized,
               aNr: AccountNumber, dt: Date;
```

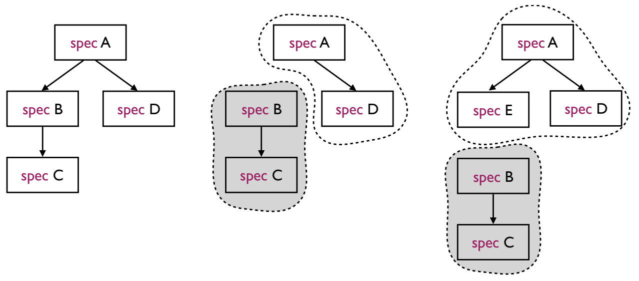
For each check or run that is performed a specialized module is created during preparation. This module, using the syntax of REBEL2 (meaning that it is itself a valid REBEL2 module), contains exactly those specifications that are referenced in the **config** and **spec**s reachable from the check or run command to be performed. This newly created module is *self contained* meaning that it does not need any external dependencies (i.e. imports) to be checked.

To be able to create a module that only contains those specifications that are referenced a specification dependency graph is built, which in turn is used to find all reachable specifications. The reachability algorithm checks which specifications are reachable from those specifications referenced in the **config** statement. The found set of specifications contains the minimally needed (transitive) dependencies. This set of specifications, together with the **check** (or **run**) command and referenced **assert** and **config** statements make up the specialized check-module.

Applying Forget and Mock: During preparation the execution of the **forget** and **mock** operators are also performed. Both **forget** and **mock** manipulate the specification dependency graph by removing those dependencies that are not needed any more after applying **forget** and **mock** (see Figure 3). Applying **forget** results in a subgraph of the original dependency graph with edges removed (see Figure 3b). Applying **mock** results in a graph that overlaps the original graph but also holds the newly inserted mock specification (see Figure 3c). In both cases the reachability analysis is performed again after the alteration of the graph resulting in the minimal set of specifications that is needed to perform the **check** or **run** command.

Next to altering the dependency graph both operators also rewrite parts of the specification as well. *Forget* slices out the field that is to be forgotten from the specification. It removes both the field definition as well as all constraints with references to the field in the pre- and postconditions of the events. This is done via a standard data-flow analysis in which the use-definition relation is traversed for the fields that need to be forgotten.

Mock performs a *rename* on the specification that is configured as an mock. The mock is renamed to the original (the specification of which it is a mock). This renaming is necessary to make sure the new combination of specifications is well-formed again since the unaltered specifications still reference the original, non-mocked specification by name.



(a) Original (b) **forget** B (c) E **mock** B

Fig. 3: Example of an specification dependency graph (a) and the consequence of applying **forget** (b) and **mock** (c) operators. The specifications in the grayed subgraphs are removed.

B. Step 2: Normalization

Normalization of a module entails three parts: (1) Inlining the state definitions as local fields, (2) Adding frame conditions to the events and (3) Adding a frame event. The result of normalizing a REBEL2 specification is again a valid REBEL2 specification. Normalization consists of the application of purely local transformations. Figure 4 shows the effect of normalizing the (slightly altered) `Transaction` specification. We will discuss every part separately.

Inlining State Definitions: As mentioned in the formalization section III the **states** definition in a REBEL2 specification becomes part of the local fields. A simple transformation is performed on each specification that introduces a new field `state` with a new type representing the states as defined in the original specification. A new specification is added to represent the new state type. Listing 4b contains an example of such a new 'state specification' (line 18). The definition of such a specification also introduces *constant instances* (`CREATED` and `BOOKED` in our example). These constant instances are instances of the `TState` specification that are implicitly part of each **config** statement. They can be referenced as constants by other specifications (e.g., see line 14 or 16 in Listing 4b). Constant instances serve a similar purpose as enumeration types do in other languages.

Next to the addition of this new 'State'-type, the pre- and postconditions of the events are strengthened with constraints on the newly added `state` field. These constraints simulate the **states** definition of the original specification. Lines 14 and 16 contain the constraints to simulate the **states** definition of the defined `Transaction`.

Adding Frame Conditions: A known problem when modeling behavioral systems declaratively is that a user must not only state what changes, but also what *does not* change otherwise the system might be under-constrained and thus start to behave in an unexpected manner. To relieve the user of this extra burden REBEL2 offers a simple heuristic; those fields whose next value is not referenced in a postcondition will be

```
spec Transaction
  frm: Account,
  to: Account,
  amt: Integer;

  init event create(...) ...
  final event archive() ...

  event book()
  pre:
    this.frm.withdraw(this.amt),
    this.to.deposit(this.amt);

  states:
    (*) -> created: create;
    created -> booked: book;
    booked -> (*): archive;
```

(a) Original specification.

```
spec Transaction
  frm: Account,
  to: Account,
  amt: Integer,
  state: TState;

  init event create(...) ...
  final event archive() ...

  event book()
  pre:
    this.frm.withdraw(this.amt),
    this.to.deposit(this.amt),
    this.state = TState[CREATED];
  post:
    this.state' = TState[BOOKED];

spec TState[CREATED,BOOKED];
```

(b) Inlining the `states` definition.

```
spec Transaction
  frm: Account,
  to: Account,
  amt: Integer,
  state: TState;

  init event create(...) ...
  final event archive() ...

  event book()
  pre:
    this.frm.withdraw(this.amt),
    this.to.deposit(this.amt),
    this.state = TState[CREATED];
  post:
    this.state' = TState[BOOKED],
    this.amt' = this.amt,
    this.frm' = this.frm,
    this.to' = this.to;

spec TState[CREATED,BOOKED];
```

(c) Adding frame conditions.

```
spec Transaction
  frm: Account,
  to: Account,
  amt: Integer,
  state: TState;

  init event create(...) ...
  final event archive() ...

  event book()
  pre:
    this.frm.withdraw(this.amt),
    this.to.deposit(this.amt),
    this.state = TState[CREATED];
  post:
    this.state' = TState[BOOKED],
    this.frm' = this.frm,
    this.to' = this.to;

  event frame()
  post: this.amt' = this.amt,
        this.frm' = this.frm,
        this.to' = this.to,
        this.state' = this.state;

spec TState[CREATED,BOOKED];
```

(d) Adding framing event.

Fig. 4: Result after each normalization step of the `Transaction` specification.

automatically added as frame condition. Listing 4c shows an example of this. In the `book` event the `frm`, `to` and `amt` fields are not referenced in the the postcondition. During normalization constraints are added to frame the value of these fields in the next state.

This is also where the **`init`** and **`final`** modifiers come into play. Frame conditions are not added in events flagged with these modifiers since this would lead to unsatisfiable constraints (a variable can not have a value if the machine is not in an initialized state).

Please note that there are cases where this heuristic fails to add all needed frame conditions. For instance when a fields next value is referenced in a postcondition but the formulated constraint is under-constrained (resulting in multiple possible values in the next step). Considering that mistakes are often easily spotted during model checking and the fact that a user can always add a custom frame condition to the postcondition instead of relying on the automatic addition we feel that this is less of a problem.

Adding Frame Event: Rebel2 allows checking multiple instances of specifications at the same time. In each step however, only one instance may make a step (or multiple if the step entails synchronization of events). The other instances must by definition keep their current values (see Formalization, Section III). To facilitate this a local `frame` event is added to every specification. This event is raised whenever the instance is not (part of) the instance that makes a step. The `frame` event frames all field values in the next state to the values of the current state (see Listing 4d).

C. Step 3: Translation

After normalization the resulting specification(s) are translated to an AlleAlle problem [9]. AlleAlle is a *relational model finder* which searches for satisfying relational instances of a given relational problem.

Short introduction of AlleAlle*:* AlleAlle's logic combines *relational algebra*, *first order logic* and *transitive closure*. AlleAlle is similar to the relational model finder Kodkod [16]. The difference is that AlleAlle utilizes an SMT solver instead of the SAT solver used by Kodkod. This means that AlleAlle can utilize a direct encoding of constraints over integers and strings without needing a specialized boolean encoding. AlleAlle's underlying logic is based on Codd's relational algebra [17] so that constraints on attributes (integer, string, etc.) can be directly expressed using the *selection* operator.

An AlleAlle problem contains two parts: i) Relational definitions and ii) Constraints on these relations. The definition of AlleAlle relations come from the relational model [18]. This definition prescribes that a relation contains a *header* and a *body*. The header defines the attribute names and domains, the body contains the (potential) tuples. AlleAlle is bounded: a relation cannot hold more or different tuples than described in its upper bound and never less than those defined in its lower bound.

The constraints are formulated using a combination of relational algebra, first order logic and transitive closure. An AlleAlle problem is satisfiable if there is a valuation of each relation such that both the tuple bounds and the constraints hold. There can be multiple satisfying instances. Finding these instances is performed by the AlleAlle model finder.

Encoding Rebel2 *specifications as* AlleAlle *problems:* AlleAlle, like Kodkod, is a general purpose model finder. It does not offer built-in support for encoding transition systems. To encode the transition system we use a similar encoding as described by Cunha [19]. In essence it means that every value that can change between each step in the transition system is encoded as a ternary relation. For instance, the `balance` of an `Account` can change in each step of the transition system therefore the AlleAlle relation representing this value is defined as the ternary relation: `Configuration x Account x Integer`. We use the term `Configuration` to describe the state of the LKS as a whole since the term 'State' is highly ambiguous. The `Account` relation holds all the instances of the `Account` that are defined in the `config` statement.

Next to the `Configuration` relation there is the binary `Step` relation which encodes the order of `Configuration`. The maximal number of `Configurations` and `Steps` depend on the `max steps` defined in the `check` or `run` command.

Field and event parameter cardinalities (optional, set or scalar) are encoded as additional ALLEALLE constraints. All the constraints of the pre- and postconditions of an event definition are conjuncted to a single ALLEALLE formula per event. The `Step` relation is used to encode the fact that there can be only one event that is raised in every step (e.g., "for all steps it must hold that there is only one raised event"). The translation of assumptions and assertions also make heavily use of the `Step` relation and the transitive closure of this relation to encode the path reachability properties of the LTL expressions.

After the translation the translated problem is given to the ALLEALLE model finder together with respective minimization criteria to find a solution in the least number of steps. This means that in the case that a counter example exists (or witness, depending on the executed command) the model finder will return a shortest path.[4]

D. Step 4: Interpretation of the Result

The last step of the model checking pipeline is the interpretation of the result. If the ALLEALLE problem is not satisfiable the user is prompted with the message that ALLEALLE can not find a satisfying model. If this is the outcome of running a `check` command it might mean that the checked assertion holds but since the used model finding technique is bounded this is not guaranteed.[5]

If the relational constraints of the generated ALLEALLE problem are satisfiable, the results are interpreted back into the domain of REBEL2 and presented to the user as a textual or interactive trace. The interactive trace allows the users to step through the found (counter) example.

V. EVALUATION

We evaluate both the expressiveness of REBEL2 and the effectiveness of mocking for model checking by implementing two case studies, one from the automotive domain, and one from the financial domain.

A. Case Study – Exterior Lighting System

As part of the ABZ conference of 2020 the real-world case study "Adaptive Exterior Light and Speed Control System" (ELS and SCS respectively) was presented [20]. We have implemented a part of the case study, consisting of a model of the direction indicators and hazard warning lights system, to compare the expressiveness, conciseness, and overall usability of REBEL2 with others state-based implementations.[6]

[4]This is a usability feature. Shorter paths are often easier to explore and comprehend when a bug is found.

[5]There still could potentially be a counterexample if the bounds would be extended.

[6]See https://github.com/cwi-swat/rebel2/releases/tag/jan-2021 in folder examples/paper/els for the encoding of the ELS case study in REBEL2.

TABLE I: SLOC comparison between different methods.

Method	SLOC	Included files
ASMeta [21]	361	CarSystem001
Event-B [22]	455	M2 (not plain text, only lighting related lines)
Classical-B [23]	363	Sensors, PitmanController_v6, PitmanController_TIME_v4, GenericTimers, BlinkLamps_v3
Electrum [24]	155	AdaptiveExteriorLight_EU (only lighting related lines)
REBEL2	244	Actuators, Input, Sensors, Timer

The system is split into three different subsystems: 1) Input 2) Sensors and 3) Actuators. The case also prescribes a timing component: the direction lights must blink 60 times per minute. This means that, when blinking, a full cycle (from bright to dark) must be completed every second. REBEL2 does not support continuous time but it is possible to model a `Timer` machine to simulate time at every step. This `Timer` has a single invariant: time always flows forward with each successive behavioral step of the system. The `Timer` can be used to (partially) model the timing requirements stated in the requirements.

Table I contains an overview of the different implementations in terms of Source Lines of Code (SLOC), restricted to the part that we have implemented. As can be seen in Table I the REBEL2 specification is comparable with the other implementations in terms of size, sitting between the Electrum and ASM / Classical-B implementations.

Out of the 13 stated requirements for the direction indicators and hazard warning lights the REBEL2 specification covers 11. The two missing or impartial requirements (ELS-4 and ELS-6) are concerned with modeling variants of the lighting system for different markets (e.g., EU versus USA) and the timing of the blink cycle when switching from tip-blinking to continuous blinking. Table II provides an overview of the fulfillment of the requirements of each implementation, as extracted from cited papers and available code.

We specified 17 assertions to check relevant properties of direction indicators and the hazard warning light. The initial decomposition of the system into four separate specifications (`Actuators`, `Sensors`, `Input`, `Timer`) facilitated checking local properties of each specification. The full sensor values contain more information than is needed to check the behavior of the direction and warning lights, so they could be mocked out. The mock state machine of `Sensor` only needed to specify single value (whether or not the key was in the ignition on position) to fully support the assertions, which resulted in a model checking speedup of approximately 1.17x.

The case description also documented scenarios describing the input and expected output values of all subsystems at a given time. For instance, the direction indicator scenario contains 26 steps. This scenario was represented as a dedicated (linear) state machine with 26 states, where each transition encoded

TABLE II: Fulfillment of direction blinking and hazard warning lights requirements.

Method	D.B.[1]	H.W.L.[2]	Remarks
ASMeta [21]	◕	●	No time management, simulated via events. Blinking frequency is missing.
Event-B [22]	◕	●	Blinking frequency is missing.
Classical-B [23]	●	●	Presented solution only addresses directional blinking and hazard warning lights.
Electrum [24]	◓	●	No time management. All integer values are replace by enumerations.
REBEL2	◕	●	Variants (EU-USA) not modelled. Time management partially implemented.

1) Direction Blinking (ELS 1–7) 2) Hazard Warning Lights (ELS 8–13)

● = Fully implemented

◕ = Fully implemented with minor omissions

◓ = Implemented but with omissions.

a step of the scenario. The events corresponding to the steps capture the given sensor values and inputs as preconditions and the expected output values as postconditions.

B. Case Study – Debit Card Lifecycle

This case study stems from the direct collaboration between the authors and a large bank, and describes the lifecycle of debit cards for payments or ATM withdrawals.[7] The case study involves three key REBEL2 specifications: DebitCard (97 SLOC), Limit (weekly withdraw limits; 45 SLOC), and Date (full date specification, including leap years, day of the year, and day of the week; 102 SLOC).

The specified assertions either check for desired behavior (e.g. "Can a debit card be produced and activated?") or check a safety property (e.g. "A customer should not be able to use a debit card after three failed PIN code attempts"). Per specification the model checker was executed on a configuration without mocked specifications and one with mocked specifications, for a total of 9 different assertions. The benchmark was run on a MacBook Pro (late 2015 model) with an Intel i5 processor and 8GB of RAM using Java version 11 (AdoptOpenJDK, build 2018-09-25), Rascal version 0.18.2 and Z3 version 4.8.8.

Table III shows the results of the experiment. All the checks with mocked specifications complete faster than their counterparts without mocked specifications, with an overall speedup factor in the range of 2x–5x. Most phases of the checking pipeline are faster with mocked specifications, except the preparation phase. The phase that mostly benefits of mocking is the solving phase. The speedup factor for this phase is between 2x to 234x (not taking the checks that timed out into consideration). Four checks ("CanAddOverrideAndCheck", "CanOverdrawLimit", "DebitCardCanBeProduced" and "CardCanExpire") could not be checked with the configuration without mocked specifications due to time-outs. Their counterparts with mocked specifications could however be checked within reasonable time.

As a proxy of the size of the explored state space, we report on the number of declared SMT variables and SMT clauses. Table III shows a decrease of the number of SMT variables and clauses in the case of mocked specifications, which in turn all have faster solving times. This data suggest that solving time is related to the number of variables and clauses. This is, however, not always the case since the "BlockedAfterThreeAttempts" and "Max3WrongPinAttempts" assertions can still be solved for the specifications without mocking while requiring more SMT variables and clauses than "DebitCardCanBeProduced" and "CardCanExpire", which result in a time-out.

C. Discussion

We found that REBEL2 is expressive enough to implement both case studies with the exception of the continuous time aspect of the automotive case. This problem is not unique to the REBEL2 language since other formalisms in this case study exhibited the same constraint.

Next to the expressiveness of the language we evaluated the effectiveness of mocking for model checking. We found that, especially in the financial case study, mocking allowed for the checking of properties that resulted in time-outs without mocked specifications. This however comes at an expense: using `forget` and `mock` makes model checking inherently unsound. In other words, it is possible to validate a property of interest in isolation, which will not hold when the system is considered as a whole.

The unsoundness of `forget` and `mock` may seem undesirable from the point of formal correctness, but lack of soundness is accepted in many other areas of validation and verification. For instance, the "small scope"-hypothesis (most bugs are found in a small scope) is at the heart of light-weight formal methods as, for instance, promoted by Alloy. Similarly, in bug-finding and static analysis, it is well-known that many analyses are inherently unsound [25], but that does not diminish their usefulness. Finally, mocked specifications are similar to mocked objects used for testing in object-oriented software development [6]. Mocked objects often have different behaviors than the actual objects they substitute for, but the benefit of using them is widely acknowledged [26].

The `forget` and `mock` operators in REBEL2 are designed to support a more flexible, conversational style of checking properties, much like unit testing or property-based testing in software development [27]. One way of stating this is: REBEL2 favors timely feedback over logical soundness.

Another added benefit of `mock` and `forget` is that having language constructs specific for abstraction helps making the applied abstractions needed for model checking explicit where they would otherwise remain implicit for the unsuspecting reader. In other words, a specification language without these constructs expects the user to create abstract specifications in

[7]See https://github.com/cwi-swat/rebel2/releases/tag/jan-2021 in folder examples/paper/debitcard for the REBEL2 encoding of the Debit Card case.

TABLE III: Comparison between model checking with and without mocking for the Debit Card case. Reported times are the found median in seconds after 10 runs.

	Without mocking							With mocking						
	Prep. (sec.)	Norm. (sec.)	Trans. (sec.)	Solve (sec.)	Total (sec.)	#vars (SMT)	#clauses (SMT)	Prep. (sec.)	Norm. (sec.)	Trans. (sec.)	Solve (sec.)	Total (sec.)	#vars (SMT)	#clauses (SMT)
CanInitializeLimit	4.4	4.5	9.2	23.4	41.5	314	715 708	1.8	1.8	5.5	0.1	9.1	222	348 399
CanAddOverrideAndCheck	4.2	4.8	23.1	t/o	t/o	732	2 144 348	1.8	1.7	13.8	0.7	17.9	504	1 045 782
CantOverdrawLimit	4.6	4.8	40.9	t/o	t/o	1150	3 587 839	1.8	1.7	22.9	2.7	29.1	786	1 758 738
LimitIsAlwaysPositive	4.2	5.1	33.3	2.1	44.6	941	2 858 968	1.8	1.7	18.4	1.2	23.1	645	1 394 935
AlwaysInSameCurrency	4.1	4.8	33.8	1.7	44.5	941	2 858 936	1.8	1.7	18.4	1.0	22.9	645	1 394 903
DebitCardCanBeProduced	6.4	10.3	68.3	t/o	t/o	1471	8 557 656	8.5	3.3	23.9	0.6	36.3	883	641 575
CardCanExpire	7.0	10.3	120.2	t/o	t/o	2337	14 258 727	8.6	3.2	40.9	1.9	54.7	1405	1 069 529
BlockedAfter3Attempts	5.8	7.7	259.2	33.9	306.6	4502	28 510 441	9.6	3.4	88.1	3.8	104.9	2710	2 137 854
Max3WrongPinAttempts	5.9	7.9	264.8	36.5	315.2	4502	28 510 471	9.3	3.4	89.7	3.9	106.3	2710	2 137 884

t/o = Timed out after 10 minutes.

the first place, rendering it difficult for readers to see which properties were abstracted from.

VI. Related Work

Alloy is a popular lightweight formal specification language based on relational logic with transitive closure [7], [14]. Alloy allows for bounded model finding by translating specifications to SAT formulas and utilizing an external SAT solver [16]. Like Rebel2, the user specifies the bounds of a problem, which are used during model finding. Because of Alloy's generality specifying behavioral problems (which require some sort of transition system) requires an encoding in the relational logic of Alloy.

Electrum extends Alloy with temporal operators [13] to make such encodings more direct. The temporal operators can be used to express safety and liveness properties and operate over so called *variable* relations, relations whose contents can change over time. DynAlloy [28], [29] is a dynamic logic-based extension of Alloy, supporting partial correctness reasoning via actions and action composition. The addition of actions in DynAlloy obviates the need of an explicit encoding of the transition system, but it does not offer support for LTL formulas, which makes expressing liveness properties hard.

Both Electrum and DynAlloy can be used to model structural and behavioral problems, but differ from Rebel2 in a number of ways. First, since Alloy translates specifications to SAT formulas it is hard to reason about non-relational data, such as integers, reals, or strings. Since Rebel2 translates it specifications to AlleAlle which in turn utilizes an SMT solver, Rebel2 offers native reasoning support in the theories supported by the solver. Second, Alloy, Electrum and DynAlloy support modularizing specifications using modules and inheritance but lack the forget and mock mechanisms of Rebel2. As a result, feasibility of checking properties is relative to the full specification, rather than the property of interest. We do believe however that both constructs could be implemented in these formalisms.

Abstraction is a key mechanism to control for complexity. In the context of formal specification this applies to both complexity reduction for humans, as well as potential reduction in the search space for automated proofs and model checking. For instance, the specification language mCRL2 [3] offers the primitives *internal action* or τ-step and the *abstraction operator* (τ_I) for this purpose. Another approach, which lies at the heart of formalisms such as Event-B [4] and ASM [30], is the concept of refinement. The specifier starts with a high-level specification of the system which is then gradually refined into more detailed specifications. Each refinement step must be proven to be correct via proof obligations which can have to be discharged, either using automatic tool support, or manually by providing a proof. Rebel2's mock and forget can be seen as similar operators to eliminate detail from a specification, in order to make model checking more feasible.

Mocking is a compositional reasoning technique. These techniques have a rich research history with seminal work of Owicki and Gries [31] and Lamport in the 1970s [32]. Assume-guarantee reasoning is a compositional reasoning technique with resemblance to our described mocking technique [33]. With assume-guarantee reasoning components are checked in isolation by assuming properties that must be guaranteed by the rest of the system (the environment). Cobleigh et al. introduced a method to automatically learn the assumptions that should be guaranteed by the environment when verifying a property on a single component [34]. With this technique it is possible to automate a large part of assume-guarantee reasoning. With mocking we aim for a similar goal: to be able to check a component in isolation by supplying the model checker with smaller, drop-in replacements of the interacting components. These replacement components (i.e. *mocks*) encode the assumptions that are needed to preform the check in isolation. Like the earlier described abstraction and refinement techniques, assume-guarantee reasoning is fully sound. Mocking on the other hand is not. Next to that, assume-guarantee reasoning can be mostly automated while mocking requires manual effort by the specifier. However, like discussed earlier, the flexible and unsound approach of mocking does allow for a more pragmatic and conversational style of interaction between the user and the model checker, even allowing partial assumptions.

VII. Conclusion

In this paper we have introduced the specification language REBEL2 which contains two language constructs, `forget` and `mock`, which offer the possibility to apply mocking during model checking. These two constructs allow the user to reduce the state space that needs to be traversed by the model checker. They can be used to redefine (parts of) the specification during model checking without having to change the original specifications. Users can specify the problem at hand without worrying about the impact on model checking at design time, but rather defer such concerns until actually checking a property of interest. We conjecture that this makes REBEL2 suitable for specifying industry-scale systems, such as those found in large enterprises, while still being able to verify (parts of) a system using model checking.

We have evaluated REBEL2's expressiveness and the effectiveness of model checking with mocking by implementing two industrial case studies. In the first case study – originating from automotive domain – we compared REBEL2 with existing solutions in alternative frameworks (ASMeta, Electrum, Classical-B and Event-B). The results showed that REBEL2 can be used to specify such problems in roughly the same number of lines code, and that applying the `mock` construct to parts of the specifications sped up model checking by a factor of roughly 1.17x. In the second case study we investigated the effectiveness of model checking with and without mocking. This case stemmed from the financial industry and was conducted together with employees from a large bank and showed that applying mocks while model checking improved the overall checking times up to 5 times. In some cases it made checking possible where performing the model checking without mocking resulted in time-outs of the underlying solver.

There are many opportunities for further research including 1) extend REBEL2 to support deadlock detection, since this is now an impediment to model checking; 2) implement different slicing algorithms (e.g., [35]) and assess the impact in terms of performance and soundness; 3) experiment with learning mocks by incorporating the assumption learning techniques of Cobleigh et al. used in assume-guarantee reasoning [34] and 4) provide empirical corroboration of the mocking hypothesis.

To summarize, REBEL2 is a formal specification language aimed at large industrial enterprise settings, which brings the concept of mocking to the world of formal methods, providing faster model checking feedback when checking behavioral properties of interest.

Acknowledgement

We would like to express our gratitude to Rene Niekel of ING for his collaboration and feedback.

References

[1] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018, vol. 10.

[2] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 1999.

[3] J. F. Groote and M. R. Mousavi, *Modeling and analysis of communicating systems*. MIT press, 2014.

[4] J.-R. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models: Application to event-b," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 1–28, 2007.

[5] G. D. Dennis, "A relational framework for bounded program verification," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[6] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, pp. 287–301, 2000.

[7] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[8] D. Jackson and J. Wing, "Lightweight formal methods," *FME 2001: Formal Methods for Increasing Software Productivity*, p. 1, 1996.

[9] J. Stoel, T. van der Storm, and J. J. Vinju, "Allealle: bounded relational model finding with unbounded data," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 46–61.

[10] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[11] J. Stoel, T. v. d. Storm, J. Vinju, and J. Bosman, "Solving the bank with rebel: on the design of the rebel specification language and its application inside a bank," in *Proceedings of the 1st Industry Track on Software Language Engineering*, 2016, pp. 13–20.

[12] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha, "State/event-based software model checking," in *International Conference on Integrated Formal Methods*. Springer, 2004, pp. 128–147.

[13] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 373–383.

[14] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[15] P. Klint, T. van der Storm, and J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *SCAM*. IEEE, 2009, pp. 168–177.

[16] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[17] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 26, no. 1, pp. 64–69, 1983.

[18] C. Date, *An Introduction to Database Systems*, 6th ed. Reading, MA, Addison-Wesley, 1994.

[19] A. Cunha, "Bounded model checking of temporal formulas with alloy," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 303–308.

[20] F. Houdek and A. Raschke, "Adaptive exterior light and speed control system," in *International Conference on Rigorous State-Based Methods*. Springer, 2020, pp. 281–301.

[21] P. Arcaini, S. Bonfanti, A. Gargantini, E. Riccobene, and P. Scandurra, "Modelling an automotive software-intensive system with adaptive features using asmeta," in *International Conference on Rigorous State-Based Methods*. Springer, 2020, pp. 302–317.

[22] A. Mammar, M. Frappier, and R. Laleau, "An event-b model of an automotive adaptive exterior light system," in *International Conference on Rigorous State-Based Methods*. Springer, 2020, pp. 351–366.

[23] M. Leuschel, M. Mutz, and M. Werth, "Modelling and validating an automotive system in classical b and event-b," in *International Conference on Rigorous State-Based Methods*. Springer, 2020, pp. 335–350.

[24] A. Cunha, N. Macedo, and C. Liu, "Validating multiple variants of an automotive light system with electrum," in *International Conference on Rigorous State-Based Methods*. Springer, 2020, pp. 318–334.

[25] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[26] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 402–412.

[27] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[28] M. F. Frias, J. P. Galeotti, C. G. López Pombo, and N. M. Aguirre, "Dynalloy: upgrading alloy with actions," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 442–451.

[29] G. Regis, C. Cornejo, S. Gutiérrez Brida, M. Politano, F. Raverta, P. Ponzio, N. Aguirre, J. P. Galeotti, and M. Frias, "Dynalloy analyzer: A tool for the specification and analysis of alloy models with dynamic behaviour," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 969–973.

[30] E. Börger, "High level system design and analysis using abstract state machines," in *International Workshop on Current Trends in Applied Formal Methods*. Springer, 1998, pp. 1–43.

[31] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Communications of the ACM*, vol. 19, no. 5, pp. 279–285, 1976.

[32] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977.

[33] O. Grumberg and D. E. Long, "Model checking and modular verification," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 843–871, 1994.

[34] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 331–346.

[35] R. Eilers, J. Hage, W. Prasetya, and J. Bosman, "Fine-grained model slicing for rebel," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 235–244.