Software Component Design with the B Method - A Formalization in Isabelle/HOL

David Déharbe, Stephan Merz

Software component design with the B method — a formalization in Isabelle/HOL *

David Déharbe[1] and Stephan Merz[2]

[1] UFRN/DIMAp, Natal–RN, Brazil
david@dimap.ufrn.br
[2] Inria, Villers-lès-Nancy, F-54600, France
Stephan.Merz@loria.fr

Abstract. This paper presents a formal development of an Isabelle/HOL theory for the behavioral aspects of artifacts produced in the design of software components with the B method. We first provide a formalization of semantic objects such as labelled transition systems and notions of behavior and simulation. We define an interpretation of the B method using such concepts. We also address the issue of component composition in the B method.

Key words: B-method, formal semantics, Isabelle/HOL, simulation

1 Introduction

The B method is an effective, rigorous method to develop software components [1]. There exist tools that support its application, and it is used in industry to produce software components for safety-critical systems.

The B method advocates developing a series of artifacts, starting from an abstract, formal specification of the functional requirements, up to an implementation of this specification, through stepwise refinements. The specification is verified internally for consistency, and may be validated against informal requirements through animation. The specification and the successive refinements are produced manually, and are subject to *a posteriori* verification. In order to systematize the correctness proofs, the B method associates proof obligations with specifications and refinements, and these are automatically generated by the support tools and discharged automatically or interactively by the user. Finally, the implementation produced in the B method is translated to compilable source code (there are code generators for C and Ada, for instance).

Improvements in the application of the B method can be achieved by increasing the degree of automation in the construction of refinements and in the verification activities. The BART project provides a language and a library of refinement rules that may be applied automatically [2]. However these rules

* This work was partially supported by CNPq grants 308008/2012-0 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br), and STIC/Amsud–CAPES project MISMT.

have not been verified, and the resulting artifacts must be submitted to formal verification. Verification of proof obligations also benefits from advancement in automated theorem proving such as the use of SMT solvers [3, 4]. One missing step in the verification aspect of the B method is the code generation from the implementation artifacts. In practice, one apprach taken to mitigate risks of errors introduced in the code generators is redundancy: apply multiple code generators and execute the generated components in parallel.

Extending the scope of formal verification in the B method would benefit from having a machine-checkable formalization of the semantics of the B method. Such a formal semantic framework could be applied to prove the correctness of refinement rules, or at least derive proviso conditions that would be simpler to prove than the general-purpose proof obligations applied to refinements, as shown in [5]. Moreover, it could also be the employed to establish a library of verified refactoring rules, such as [6]. Finally, such a semantic framework could be used to demonstrate the correctness of the code generator, assuming that it is extended to include the constructions used in the target programming language.

In this paper, we present a formalization of the behavioral aspects of artefacts of the B method that may be taken as a starting point towards the construction of the formal semantic framework we envision. This formalization is carried out using the proof assistant Isabelle/HOL [7]. We represent the behavior of a (B) software component as a labeled transition system (LTS). We first provide a formalization of LTS, as well as the classic notion of simulation relation between LTSes. Next we formalize the concepts of B specification, refinement and project, based on LTSes as underlying behavioral model. We adapt the notion of simulation so that it matches the concept of refinement in the B method. Finally, we address the composition of components in the B method.

Outline: Section 2 presents the technical background of the paper, namely the B method and formalization in Isabelle/HOL. Section 3 contain a description of the formalization of labeled transition systems and associated concepts: simulation, traces, etc. Then, section 4 formalizes software development in B, relating each type of artifact to its semantic interpretation. Section 5 is devoted to the presentation and formalization of component composition in B. Section 6 concludes the paper, discussing related work and prospective extensions.

2 Background

2.1 The B method

In the B method [1], an individual project consists in deriving a software system that is consistent with a high-level specification. The derivation follows principled steps of formal system development: specifications are decomposed into (libraries of) modules from which executable programs are eventually obtained. Each module has a specification, called a machine, and its implementation is derived formally by a series of modules called refinements. Such modules may be used to specify additional requirements, to define how abstract data may be

encoded using concrete data types, or to define how operations may be implemented algorithmically. From a formal point of view, each module is simulated by the subsequent refinement modules, and this relationship is ensured by discharging specific proof obligations. A refinement is called an implementation when its data is scalar and behavior is described in a procedural style. Implementations may be translated into an imperative programming language such as C.

At every level of the refinement chain, a module describes a state transition system. A module contains variables and instances of other modules; the state space of a module is the compositon of the set of all possible valuations of its variables and the state spaces of its components. A module also contains an initialization clause that establishes the possible initial states. And a module has operations that describe how the system might transition from one state to another state. Each operation has a name, it may have input and output parameters, as well as a precondition, specifying the configurations of state and input parameters values in which the operation is available and guaranteed to terminate.

To illustrate this, figure 1 provides a simple B machine of a counter from zero to three. The state of the module is the value of variable counter. The invariant provides the type and the range of possible values for this variable. The initialisation specifies two possible initial values for c: either 0 or 3. The operations specify that the counter may always be reset to zero, that it may be incremented only when it has not reached its upper bound. The operation get is always available, does not change the state and has a single output parameter which holds the value of c.

Figure 2 contains the graph of the labelled transition system corresponding to the B machine counter3. Each node depicts one of the four possible states, and directed edges correspond to transitions between states. They are labelled with events that correspond to the operations active in each state.

```
MACHINE counter3
VARIABLES c
INVARIANT c : INTEGER & c : 0..3
INITIALISATION c :: {0, 3}
OPERATIONS
zero = c := 0;
inc = PRE c < 3 THEN c := c + 1 END;
out <-- get = out := c
END
```

Fig. 1. A simple B machine, with one state variable, named c, and three operations (zero, inc and get).

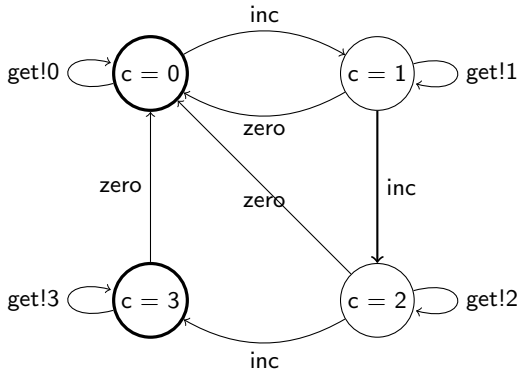

Fig. 2. Labelled transition system of the counter3 machine. Initial states are drawn with thick lines. Each transition is labelled with the corresponding operation, possibly decorated with parameter values.

2.2 Formalization in Isabelle/HOL

Isabelle is a logical framework for building deductive systems; we use here Isabelle/HOL, its instantiation for higher-order logic [7]. It provides a language combining functional programming and logic that is suitable to develop rigorous formalizations. Isabelle also comes with numerous standard packages and powerful tools for automating reasoning. In particular, Sledgehammer [8] provides access to external proof tools complete with proof reconstruction that is checked by the trusted Isabelle kernel. Formalizations in Isabelle are structured in so-called theories, each theory containing type and function definitions, as well as statements and proofs of properties about the defined entities. Theories and proofs are developed in the Isar [9] language.

Isabelle has an extensible polymorphic type system, similar to functional languages such as ML. Predefined types include booleans and natural numbers, as well as several polymorphic type constructors such as functions, sets, and lists. Type variables are indicated by a quote, as in $'ev$, and function types are written in curried form, as in $'a \Rightarrow 'b \Rightarrow 'c$. Terms can be built using conventional functional programming constructions such as conditionals, local binding and pattern matching. Formulas are built with the usual logic connectives, including quantifiers. Type inference for expressions is automatic, but expressions can be annotated by types for clarity. For example, $e :: E$ specifies that expression e should have type E, which must be an instance of the most general type inferred for e. In the following we introduce the constructions of the language that are used in the presentation of the theories developed in our formalization of the B method.

Our development uses natural numbers, pairs, lists, sets, and options:

– Type *nat* is defined inductively with the constructors 0 and *Suc* (the successor function).

- Pairs are denoted as (e_1, e_2), and the components may be accessed using the functions *fst* and *snd*. If e_1 and e_2 have types E_1 and E_2, then the pair (e_1, e_2) has type $E_1 \times E_2$.
- The type $'a\ list$ represents finite lists with elements of type $'a$; it is defined inductively from the empty list $[\,]$ and the cons operation # that prefixes a list by an element. The standard library contains operators such as *hd* and *tl* (head and tail of a list), @ (concatenation), ! (access by position), *length*, and *map* (constructing a list from a given list by applying a function to every element).
- The theory for sets defines type $'a\ set$ of sets with elements of type $'a$ and contains definition for all the standard operations on sets. Syntax is provided for conventional mathematical notation, e.g. $\{x\ .\ x \bmod 2 = 0\}$ denotes the set of even numbers. We use the generalized union operator, written *UNION*, of type $'a\ set\ set \Rightarrow 'a\ set$, that returns the union of its argument sets and the image operation ‘ $:: ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set$ that is the counterpart of the *map* operation for sets. Also, operator *Collect* yields the set characterized by a given predicate.
- In Isabelle/HOL, all functions must be total. The type $'a\ option$ is handy to formalize partial functions. It has constructors $None :: 'a\ option$ and $Some :: 'a \Rightarrow 'a\ option$ to represent either no or some value of type $'a$. Also, operator *the* accesses the value constructed with *Some*.

We also use Isabelle/HOL **record** types. A record is a possibly polymorphic, named type that consists of a series of fields, each field having a name and a type. The field name is also used as a getter function to access the corresponding field in a record value. Record values and patterns can be written as $(\!|\, fld_1 = val_1, fld_2 = val_2 \,|\!)$.

Our type definitions are either introduced via **type-synonym** (type abbreviations) or by **record**, in the case of record types. Values, including functional values, are defined either through **definition**, in the case of equational definitions, or **inductive-set**, in the case of inductively defined sets. Such commands, in addition to adding a new binding to the current context, also create theorems for use in subsequent proofs. For instance, an unconditional equational definition gives rise to a theorem expressing the equality between the defined value and the defining expression. (Definitions are not expanded by default in proofs.) Inductive definitions introduce introduction rules corresponding to each clause, as well as theorems for performing case distinction and induction. For notational convenience, a definition may also be accompanied by a syntax declaration, for example for introducing an infix mathematical symbol.

Properties of the entities thus defined are introduced in the form of **lemma**, **theorem** and **corollary** paragraphs (there is no formal distinction between these levels of theorems). Properties are written as $[\![\, H_1; H_2; \cdots H_n \,]\!] \Longrightarrow C$ where the H_i are hypotheses and C is the conclusion. An alternative syntax is

assumes H_1 **and** H_2 **and** $\ldots$ H_n
shows C

In an Isabelle theory, statements of properties are immediately followed by a proof script establishing their validity. In this paper, we have omitted all proof scripts.

3 Formalizing transition systems

The specification of a software component in the B formalism describes a labeled transition system (LTS). We therefore start with an encoding of LTSs in Isabelle, which may be of interest independently of the context of the B method.

3.1 Labeled transition systems and their runs

Our definitions are parameterized by types *'st* and *'ev* of states and events. We represent a transition as a record containing a source and a destination state, and an event labeling the transition. An LTS is modeled as a record containing a set of initial states and a set of transitions.

record *('st,'ev) Tr* =
  src :: *'st*
  dst :: *'st*
  lbl :: *'ev*

record *('st,'ev) LTS* =
  init :: *'st set*
  trans :: *('st,'ev) Tr set*

The auxiliary functions *outgoing* and *accepted-events* compute the set of transitions originating in a given state, and the set of their labels.

definition *outgoing* **where** *outgoing l s* ≡ {*t* ∈ *trans l* . *src t* = *s*}

definition *accepted-events* **where** *accepted-events l s* ≡ *lbl* ‘ (*outgoing l s*)

The set of reachable states of an LTS *l*, written *states l*, is defined inductively as the smallest set containing the initial states and the successors of reachable states.

inductive-set *states* **for** *l* :: *('st,'ev) LTS* **where**
  base : *s* ∈ *init l* ⟹ *s* ∈ *states l*
| *step* : ⟦ *s* ∈ *states l*; *t* ∈ *outgoing l s* ⟧ ⟹ *dst t* ∈ *states l*

The alphabet of an LTS is the set of all events that are accepted at some reachable state.

definition *alphabet* **where** *alphabet l* ≡ *UNION* (*states l*) (*accepted-events l*)

Runs. We formalize two notions of (finitary) behavior of LTSs. The internal behavior or *run* is an alternating sequence of states and events, starting and ending with a state. In particular, a run consists of at least one state, and the function *append-tr* extends a run by a transition, which is intended to originate at the final state of the run.

record $('st,'ev)\ Run =$
  $trns :: ('st \times 'ev)\ list$
  $fins :: 'st$

definition *append-tr* **where**
  $append\text{-}tr\ run\ t \equiv$
    $(\!|\ trns = (trns\ run) @ [(fins\ run,\ lbl\ t)],$
      $fins = dst\ t\ |\!)$

The set of runs of an LTS is defined inductively, starting from the initial states of the LTS, and extending runs by transitions originating at the final state.

inductive-set *runs* **for** $l :: ('st,'ev)\ LTS$ **where**
  $start : s \in init\ l \Longrightarrow (\!|\ trns = [], fins = s\ |\!) \in runs\ l$
| $step :\ [\![r \in runs\ l;\ t \in outgoing\ l\ (fins\ r)]\!] \Longrightarrow append\text{-}tr\ r\ t \in runs\ l$

We prove a few lemmas about runs. In particular, every run starts at an initial state of the LTS, and the steps recorded in the run correspond to transitions. Moreover, the reachable states of an LTS are precisely the final states of its runs. The proofs are straightforward by induction on the definition of runs.

lemma *runs-start-initial*:
  assumes $r \in runs\ l$
  shows $(if\ trns\ r = []\ then\ fins\ r\ else\ fst\ (hd\ (trns\ r))) \in init\ l$

lemma *run-steps*:
  assumes $r \in runs\ l$ **and** $i < length\ (trns\ r)$
  shows $(\!|\ src = fst\ (trns\ r\ !\ i),$
    $dst = (if\ Suc\ i < length\ (trns\ r)\ then\ fst\ (trns\ r\ !\ (Suc\ i))$
      $else\ fins\ r),$
    $lbl = snd\ (trns\ r\ !\ i)\ |\!) \in trans\ l$

lemma *states-runs*: $states\ l\ =\ fins\ `\ (runs\ l)$

Traces. The second, observable notion of behavior is obtained by recording only the successive events that appear in a run. We call this projection a *trace* of the LTS.

type-synonym $'ev\ Trace = 'ev\ list$

definition *trace-of* **where** $trace\text{-}of\ run \equiv map\ snd\ (trns\ run)$

definition *traces* **where** $traces\ l \equiv trace\text{-}of\ `\ (runs\ l)$

3.2 Simulations between labeled transition systems

Two transition systems l and l' are naturally related by a notion of simulation that ensures that every behavior of l can also be produced by l'. More formally, given a relation between the states of l and l', we say that l is simulated by l' if the two following conditions hold:

- Every initial state of l is related to some initial state of l'.
- Whenever two states s and s' are related and t is an outgoing transition of s then s' has an outgoing transition t' with the same label as t and such that the destination states are related.

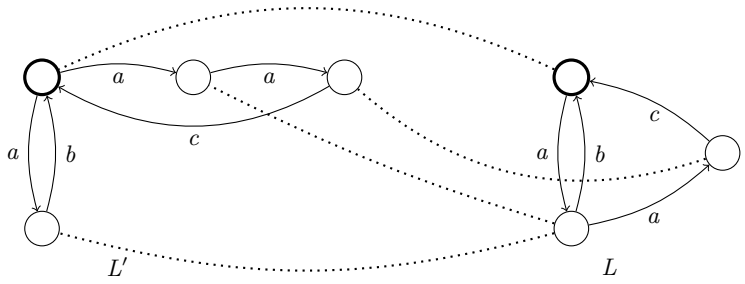

Fig. 3. Example of two LTSs L, L' such that $L' \preceq L$. Dotted lines depicts pairs of states in the simulation relation. Initial states are depicted with a thicker border.

The following definitions express these conditions in Isabelle: they lift a relation on states to a relation on transitions, respectively on LTSs. We write $l \preceq l'$ if l is simulated by l' for some relation r. We also sometimes refer to l as the *concrete* and to l' as the *abstract* LTS. Figure 3 illustrates the notion of simulation.

definition *sim-transition* **where**
$\quad$ *sim-transition* $r \equiv \{\, (t, t') \mid t\ t'\ .\ (src\ t, src\ t') \in r \wedge$
$\qquad\qquad\qquad\qquad\qquad\qquad (dst\ t, dst\ t') \in r \wedge lbl\ t = lbl\ t' \,\}$

definition *simulation* **where**
$\quad$ *simulation* $r \equiv$
$\quad\quad \{\, (l, l') \mid l\ l'\ .\ (\forall s \in init\ l.\, \exists s' \in init\ l'.\ (s, s') \in r) \wedge$
$\qquad\qquad\qquad (\forall (s, s') \in r.\, \forall t \in outgoing\ l\ s.$
$\qquad\qquad\qquad\qquad \exists t' \in outgoing\ l'\ s'.\ (t, t') \in sim\text{-}transition\ r) \,\}$

definition *simulated* (**infix** $\preceq$) **where**
$\quad l \preceq l' \ \equiv\ \exists r.\, (l, l') \in simulation\ r$

We prove some structural lemmas about simulation: every LTS is simulated by itself through the identity relation on states, and the composition of two simulations is a simulation. It follows that $\preceq$ is reflexive and transitive. All proofs are found automatically by Isabelle after expanding the corresponding definitions.

lemma *simulation-identity*:
$\quad (Id :: ('st, 'ev)\ LTS\ rel) \subseteq simulation\ (Id :: 'st\ rel)$

lema *simulation-composition*:
$\quad$ **assumes** $(l, l') \in simulation\ r$ **and** $(l', l'') \in simulation\ r'$
$\quad$ **shows** $\quad (l, l'') \in simulation\ (r\ O\ r')$

lemma *simulates-reflexive*: $l \preceq l$

lemma *simulates-transitive*: $[\![\, l \preceq l';\ l' \preceq l'' \,]\!] \Longrightarrow l \preceq l''$

We now prove that simulation between LTSs gives rise to similar behaviors: every behavior of the simulating LTS corresponds to a behavior of the simulated

one. In order to make this idea precise, we first lift a relation on states to a relation on runs.

definition *sim-run* **where**
sim-run $r \equiv \{\, (run, run') \mid run\ run'\ .$
$\quad length\,(trns\ run) = length\,(trns\ run') \wedge$
$\quad (\forall i < length\,(trns\ run).$
$\quad\quad (fst\,(trns\ run\ !\ i), fst\,(trns\ run'\ !\ i)) \in r \wedge$
$\quad\quad snd\,(trns\ run\ !\ i) = snd\,(trns\ run'\ !\ i)) \wedge$
$\quad (fins\ run, fins\ run') \in r\,\}$

By induction on the definition of runs, we now prove that whenever l is simulated by l' then every run of l gives rise to a simulating run of l'.

theorem *sim-run*:
assumes $(l, l') \in simulation\ r$ **and** $run \in runs\ l$
shows $\exists run' \in runs\ l'.\ (run, run') \in sim\text{-}run\ r$

Turning to external behavior, it immediately follows that any two similar runs give rise to the same trace. Using the preceding theorem, it follows that the traces of the concrete LTS are a subset of the traces of the abstract one.

lemma *sim-run-trace-eq*:
assumes $(run, run') \in sim\text{-}run\ r$
shows $trace\text{-}of\ run = trace\text{-}of\ run'$

theorem *sim-traces*:
assumes $(l, l') \in simulation\ r$ **and** $tr \in traces\ l$
shows $tr \in traces\ l'$

corollary *simulated-traces*: $l \preceq l' \implies traces\ l \subseteq traces\ l'$

3.3 A notion of simulation tailored for the B method

The notion of simulation considered so far requires that for any two related states, every transition of the concrete LTS can be matched by a transition with the same label of the abstract one. In particular, the concrete LTS accepts a subset of the events accepted by the abstract LTS. In the B method, it is required on the contrary that the concrete LTS accepts at least the events accepted by the abstract LTS. Concrete transitions labeled by events that are also accepted by the abstract system must still be matched, but nothing is required of concrete transitions for events that are not accepted by the abstract LTS. This idea is formalized by the following definition.

definition *simulation-B* **where**
simulation-B $r \equiv$
$\quad \{\, (l, l') \mid l\ l'\ .\ (\forall s \in init\ l.\ \exists s' \in init\ l'.\ (s, s') \in r) \wedge$
$\quad\quad (\forall (s, s') \in r.$
$\quad\quad\quad accepted\text{-}events\ l\ s \supseteq accepted\text{-}events\ l'\ s' \wedge$
$\quad\quad\quad (\forall t \in outgoing\ l\ s.\ lbl\ t \in accepted\text{-}events\ l'\ s' \longrightarrow$
$\quad\quad\quad\quad (\exists t' \in outgoing\ l'\ s'.\ (t, t') \in sim\text{-}transition\ r)))\,\}$

definition *simulated-B* (**infix** $\preceq_B$) **where**
$l \preceq_B l' \equiv \exists r.\, (l, l') \in \mathit{simulation\text{-}B}\ r$

The analogous structural lemmas are proved for this notion of simulation as for the previous one, implying that $\preceq_B$ is again reflexive and transitive. However, runs of the concrete LTS can in general no longer be simulated by runs of the abstract LTS because they may contain events that the abstract LTS does not accept at a given state. The following definition weakens the relation *sim-run*: the abstract run corresponds to a simulating execution of a maximal prefix of the concrete run. We show that whenever l is simulated by l' then a simulating run of l' in this weaker sense can be obtained for every run of l.

definition *sim-B-run* **where**
$\mathit{sim\text{-}B\text{-}run}\ r\ l' \equiv \{\, (\mathit{run}, \mathit{run}') \mid \mathit{run}\ \mathit{run}'\ .$
$\quad \mathit{length}\,(\mathit{trns}\ \mathit{run}') \leq \mathit{length}\,(\mathit{trns}\ \mathit{run}) \wedge$
$\quad (\forall i < \mathit{length}\,(\mathit{trns}\ \mathit{run}').$
$\quad\quad (\mathit{fst}\,(\mathit{trns}\ \mathit{run}\ !\ i), \mathit{fst}\,(\mathit{trns}\ \mathit{run}'\ !\ i)) \in r \wedge$
$\quad\quad \mathit{snd}\,(\mathit{trns}\ \mathit{run}\ !\ i) = \mathit{snd}\,(\mathit{trns}\ \mathit{run}'\ !\ i)) \wedge$
$\quad\quad (\mathit{if}\ \mathit{length}\,(\mathit{trns}\ \mathit{run}') = \mathit{length}\,(\mathit{trns}\ \mathit{run})$
$\quad\quad\ \mathit{then}\ (\mathit{fins}\ \mathit{run}, \mathit{fins}\ \mathit{run}') \in r$
$\quad\quad\ \mathit{else}\ \mathit{snd}\,(\mathit{trns}\ \mathit{run}\ !\ (\mathit{length}\,(\mathit{trns}\ \mathit{run}')))$
$\quad\quad\quad\quad \notin \mathit{accepted\text{-}events}\ l'\ (\mathit{fins}\ \mathit{run}'))\,\}$

theorem *sim-B-run*:
assumes $(l, l') \in \mathit{simulation\text{-}B}\ r$ **and** $\mathit{run} \in \mathit{runs}\ l$
shows $\exists \mathit{run}' \in \mathit{runs}\ l'.\, (\mathit{run}, \mathit{run}') \in \mathit{sim\text{-}B\text{-}run}\ r\ l'$

Turning to observable behavior, we introduce a refined notion of a trace that does not only record the events that occur in a run but also which events are accepted at the end of the run.

definition *traces-B* **where**
$\mathit{traces\text{-}B}\ l \equiv \{\, (\mathit{trace\text{-}of}\ r,\ \mathit{accepted\text{-}events}\ l\ (\mathit{fins}\ r)) \mid r\ .\ r \in \mathit{runs}\ l\,\}$

Theorem *sim-B-run* implies that whenever l is simulated by l' then for every (B) trace of l there exists a maximal similar traces of l'.

theorem *sim-traces-B*:
assumes $l \preceq_B l'$ **and** $(tr, acc) \in \mathit{traces\text{-}B}\ l$
shows $\exists (tr', acc') \in \mathit{traces\text{-}B}\ l'.$
$\quad \mathit{length}\ tr' \leq \mathit{length}\ tr\ \wedge\ (\forall i < \mathit{length}\ tr'.\ tr'\ !\ i = tr\ !\ i) \wedge$
$\quad (\mathit{if}\ \mathit{length}\ tr' = \mathit{length}\ tr\ \mathit{then}\ acc' \subseteq acc$
$\quad\ \mathit{else}\ tr\ !\ (\mathit{length}\ tr') \notin acc')$

4 Formalizing development in B

We now turn our attention to the artifacts produced by the application of the B method and associate them to the formal entities we have defined in the preceding section. We address successively B machines (i.e. specifications), refinements, and the development process.

4.1 Specification

The semantics of a B machine identifies it with a labelled transition system, together with an invariant, i.e. a predicate on the states of the LTS.

record $('st, 'ev)$ *B-machine* =
  lts :: $('st, 'ev)$ *LTS*
  invariant :: $'st \Rightarrow bool$

This definition of *B-machine* puts no restriction whatsoever on the invariant with respect to the LTS. In contrast, sound B machines are such that all the reachable states of the LTS satisfy the invariant. They are identified by the following predicate:

definition *sound-B-machine* **where**
  sound-B-machine $m \equiv \forall s \in states\,(lts\ m).\ invariant\ m\ s$

The following theorem states two sufficient conditions to establish that a machine is sound: all initial states must satisfy the invariant, and the transition relation must preserve the invariant. These conditions correspond to the standard proof obligations of the B method that express induction on the set of reachable states.

theorem *machine-po*:
  assumes $\bigwedge s.\ s \in init\,(lts\ m) \Longrightarrow invariant\ m\ s$
  and $\bigwedge t.\ [\![\, t \in trans\,(lts\ m);\ invariant\ m\ (src\ t) \,]\!]$
    $\Longrightarrow invariant\ m\ (dst\ t)$
  shows *sound-B-machine m*

4.2 Refinement

A B refinement is composed of an *abstract* and a *concrete* LTS related by a *gluing invariant*. The gluing invariant is a binary predicate over the states of the abstract LTS and the states of the concrete one; it corresponds to the relation on states considered in sections 3.2 and 3.3.

record $('st, 'ev)$ *B-refinement* =
  abstract :: $('st, 'ev)$ *LTS*
  concrete :: $('st, 'ev)$ *LTS*
  invariant :: $'st \times 'st \Rightarrow bool$

As in the previous definitions of simulation, we assume that the two LTSs are defined over the same types of states and events. In practice, we expect the type of states to be a mapping of variable names to values.

A refinement is considered *sound* if the invariant establishes a simulation (in the sense established in section 3.3) of the concrete component by the abstract component.

definition *sound-B-refinement* **where**
  sound-B-refinement $\equiv$
    $(concrete\ r, abstract\ r) \in simulation\text{-}B\ (Collect\,(invariant\ r))$

It then follows that every concrete execution corresponds to some execution of the abstract LTS, and that the former is simulated by the latter.

lemma *refinement-sim*
  assumes *sound-B-refinement r*
  shows *concrete r* $\preceq_B$ *abstract r*

We lift the structural properties of simulations to B refinements. First, the trivial refinement of an LTS by itself where the gluing invariant is the identity on states is a sound refinement.

definition *refinement-id* **where**
  refinement-id l $\equiv$ $(\!|$ *abstract* $= l$, *concrete* $= l$, *invariant* $= (\lambda(s,t).s = t)$ $|\!)$
lemma *refinement-id*: *sound-B-refinement* (*refinement-id l*)

Second, we define the composition of two refinements and show that the composition of two refinements is sound, provided that the concrete LTS of the first refinement is the abstract LTS of the second one. Moreover, the composition of refinements admits identity refinements as left and right neutral elements, and it is associative.

definition *refinement-compose* **where**
  refinement-compose r r' $\equiv$
    $(\!|$ *abstract* $=$ *abstract r*,
      concrete $=$ *concrete r'*,
      invariant $= \lambda p.\ p \in$ *Collect* (*invariant r'*) $\circ$ *Collect* (*invariant r*) $|\!)$
lemma *refinement-compose-sound*:
  assumes *sound-B-refinement r* **and** *sound-B-refinement r'*
  and *concrete r* $=$ *abstract r'*
  shows *sound-B-refinement* (*refinement-compose r r'*)
lemma *refinement-compose-neutral-left* :
  refinement-compose (*refinement-id* (*abstract r*)) *r* $=$ *r*
lemma *refinement-compose-neutral-right* :
  refinement-compose r (*refinement-id* (*concrete r*)) $=$ *r*
lemma *refinement-compose-associative* :
  refinement-compose (*refinement-compose r r'*) *r''* $=$
  refinement-compose r (*refinement-compose r' r''*)

4.3 B development

The development of software components in the B method proceeds by stepwise refinement. We represent this process in a so-called *B design* as a sequence of refinements. The idea is that the abstract LTS of the first refinement is gradually refined into the concrete LTS of the last refinement. Such a design is *sound* if every single refinement is sound and if the concrete LTS of each refinement is the abstract LTS of its successor. By repeated application of lemma *refinement-compose-sound*, it follows that the concrete LTS of the last refinement is simulated by the abstract LTS of the first refinement in the sequence.

type_synonym $('st, 'ev)$ *B-design* $=$ $('st, 'ev)$ *B-refinement list*

definition *sound-B-design* **where**
sound-B-design refs $\equiv$ $\forall i <$ *size refs*.
sound-B-refinement (*refs* ! *i*) $\wedge$
(*Suc i* $<$ *size refs* $\longrightarrow$ *concrete* (*refs* ! *i*) $=$ *abstract* (*refs* ! (*Suc i*)))

lemma *design-sim*:
assumes *sound-B-design refs* **and** *refs* $\neq$ [ ]
shows *concrete* (*last refs*) $\preceq_B$ *abstract* (*hd refs*)

Finally, we define a *B development* as consisting of a B machine and a B design. A sound B development consists of a sound B machine and a sound B design such that the abstract LTS of the first refinement in the design is the LTS of the B machine.

record $('st, 'ev)$ *B-development* $=$
spec :: $('st, 'ev)$ *B-machine*
design :: $('st, 'ev)$ *B-design*

definition *sound-B-development* **where**
sound-B-development dev $\equiv$
sound-B-machine (*spec dev*)
$\wedge$ *sound-B-design* (*design dev*)
$\wedge$ (*design dev* $\neq$ [ ] $\longrightarrow$ *lts* (*spec dev*) $=$ *abstract* (*hd* (*design dev*)))

It follows that in a sound B development, the concrete LTS of the final refinement simulates the initial specification.

theorem *development-sim*:
assumes *sound-B-development d* **and** *design d* $\neq$ [ ]
shows *concrete* (*last* (*design d*)) $\preceq_B$ *lts* (*spec d*)

5 Component composition in B

The language B has several mechanisms for composing components:

- The SEES construction allows one component access to definitions found in another component. This modularization mechanism aims at reusing definitions of some module across several components. It is not related to the behavioral aspects explored in this paper.
- The INCLUDES construction makes it possible to use instances of existing components to build a new specification, say machine M. The state of M is a tuple of the variables declared in M and the variables of the imported instances. The imported instances are initialized automatically upon initialization of M. The states of the imported instances are read-only in M, and each operation of M may call at most one operation of each such instance.

- B offers constructions named EXTENDS and PROMOTES that are essentially syntactic sugaring of the INCLUDES construction. A construction named USES provides read access between several instances that are included by the same machine, and provides additional flexibility to build a specification from combinations of INCLUDES components.
- The IMPORTS construction is to B implementations what the INCLUDES is for specifications: existing components may be used as bricks to build implementations. Since B implementations define the body of operations as sequential composition of atomic instructions, they may have multiple calls to operations of imported components.

In the following, we present a formalization of the INCLUDES construction. This formalization is carried out on the semantic objects associated to B components: LTSs and runs. The inclusion of a component C in a component A is represented by a record containing the LTS corresponding to C, a function to project states of A to states C, and a function to map each event of A to at most one event of C.

record $('st, 'ev)$ *Includes* $=$
  lts $::$ $('st, 'ev)$ *LTS*
  sync-st $::$ $'st \Rightarrow 'st$
  sync-ev $::$ $'ev \Rightarrow 'ev$ *option*

Next, we express the soundness conditions for such record. With respect to a given LTS A, an *Includes* record I, with LTS C, is sound when it satisfies two conditions. First, the state projection function π maps every initial state of A to an initial state of C. Next, if t is a transition of A whose event is not mapped to an event in C by the event mapping function σ, then π projects the end states of t to the same C state; if the mapping σ yields an event e of C, then C must contain a transition labeled by e relating the projections of the source and destination states of t.

definition *sound-includes* **where**
  sound-includes $A\ I \equiv$
    $(let\ (C, \pi, \sigma) = (lts\ I, sync\text{-}st\ I, sync\text{-}ev\ I)\ in$
    $\pi\ `\ (init\ A) \subseteq (init\ C) \quad \wedge$
    $(\forall t \in trans\ A\ .\ (case\ \sigma(lbl\ t)\ of$
        $None \Rightarrow \pi(src\ t) = \pi(dst\ t)$
      $\mid Some\ e \Rightarrow (\!|\ src = \pi(src\ t), dst = \pi(dst\ t), lbl = e\ |\!) \in trans\ C)))$

Sound inclusion ensures that the projection of every reachable state of the including LTS is a reachable state of the included LTS.

theorem *includes-states*:
  assumes $s \in states\ A$ **and** *sound-includes* $A\ I$
  shows $(sync\text{-}st\ I)\ s \in states\ (lts\ I)$

In order to obtain a similar result for runs, we need to define a relationship between runs of a LTS and behaviors of the included LTS. We first define an

auxiliary function *interaction-trns*: given an LTS A, an include record I with a LTS C, and a sequence of state and events corresponding to transitions of A, it filters out those pairs that do not correspond to events in C, and projects the result to states and events in C according to I. The function *interaction* uses *interaction-trns* in order to construct a run of the included LTS C from a run of A.

definition *interaction-trns* **where**
interaction-trns A I seq ≡
map (λ(*s*, *e*) . (*sync-st I s*, *the*(*sync-ev I e*)))
(*filter* (λ(*s*, *e*) . *sync-ev I e* ≠ *None*) *seq*)

definition *interaction* **where**
interaction A I run ≡
(| *trns* = *interaction-trns A I* (*trns run*),
fins = *sync-st I* (*fins run*) |)

The soundness result at the level of runs now states that the projection (in the sense produced by *interaction*) of any run of the including LTS is a run of the included LTS. It follows that the projection of any trace of the including LTS to those events that are mapped to an event of the included LTS is a trace of the included LTS.

theorem *interaction-runs*:
assumes *r* ∈ *runs A* **and** *sound-includes A I*
shows *interaction A I r* ∈ *runs* (*lts I*)

theorem *interaction-traces*:
assumes *tr* ∈ *traces A* **and** *sound-includes A I*
shows *map* (*the* ∘ *sync-ev I*) (*filter* (λ*e*. *sync-ev I e* ≠ *None*) *tr*)
∈ *traces* (*lts included*)

6 Conclusion

This paper presents a formalization of the design of software components using the formal method B. We focus on the concepts that are central to the behavioral semantics of B components, namely labelled transition systems, as well as their internal and external (observable) behavior. An important relation between such entities is that of simulation: we express the classical definition of simulation and we give a variation of simulation that corresponds to B's view of refinement properties between components. All concepts are formally defined in the Isabelle/HOL proof assistant, and related by formally proved theorems. The formalization also addresses B components and the B design process at an abstract level, relating these concepts to the semantic concepts and to simulation. Our formalization also addresses inclusion, i.e. the fundamental mechanism for component composition provided in the B specification language, and characterizes soundness for such component inclusion.

The B method has previously been subject of several formalization efforts addressing either the full B specification language or some aspects of this language.

Chartier [10] formalized the language of the B method, also with Isabelle/HOL, with the goal of formally verifying proof obligations and to produce a formally verified proof obligation generator. A similar effort was carried out by Bodeveix et al. [11], but using instead both Coq and PVS as formalization engines. It is noteworthy that their work provides a semantics for the language in terms of state transition systems, and is quite complementary to ours. Dunne [12] produced a mathematical formalization of the generalized substitution language, which was implemented in Isabelle/HOL by Dawson [13]. More recently, Jacquel et al. [14] have used Coq to formalize a proof system for B, therefore providing another rigorous framework to reason about the expression language of B.

In contrast to most previous work, our formalization focuses on the transition system semantics of B and is independent on B's concrete expression language. It would therefore be interesting to specialize the mapping of B artifacts to labelled transition systems as defined in this paper, based on the preexisting work. As a result of such a formalization, we would like to derive a library of sound refinement and refactoring rules for the B method.

References

1. Abrial, J.: The B-book - assigning programs to meanings. Cambridge University Press (2005)
2. Requet, A.: BART: A tool for automatic refinement. In Börger, E., Butler, M.J., Bowen, J.P., Boca, P., eds.: Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings. Volume 5238 of Lecture Notes in Computer Science., Springer (2008) 345
3. Mentré, D., Marché, C., Filliâtre, J., Asuka, M.: Discharging proof obligations from Atelier B using multiple automated provers. In Derrick, J., Fitzgerald, J.S., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E., eds.: Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings. Volume 7316 of Lecture Notes in Computer Science., Springer (2012) 238–251
4. Conchon, S., Iguernelala, M.: Tuning the Alt-Ergo SMT solver for B proof obligations. In Ameur, Y.A., Schewe, K., eds.: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings. Volume 8477 of Lecture Notes in Computer Science., Springer (2014) 294–297
5. Borba, P., Sampaio, A., Cornélio, M.: A refinement algebra for object-oriented programming. In Cardelli, L., ed.: ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings. Volume 2743 of Lecture Notes in Computer Science., Springer (2003) 457–482
6. Cornélio, M., Cavalcanti, A., Sampaio, A.: Sound refactorings. Sci. Comput. Program. **75**(3) (2010) 106–133
7. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Volume 2283 of Lecture Notes in Computer Science. Springer (2002)
8. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. J. Autom. Reasoning **51**(1) (2013) 109–128

9. Wenzel, M., Paulson, L.C.: Isabelle/isar. In Wiedijk, F., ed.: The Seventeen Provers of the World, Foreword by Dana S. Scott. Volume 3600 of Lecture Notes in Computer Science., Springer (2006) 41–49
10. Chartier, P.: Formalisation of B in Isabelle/HOL. In: Proc. B'98, Springer (1998)
11. Bodeveix, J.P., Filali, M., Muñoz, C.: A formalization of the B-method in Coq and PVS. In Springer, ed.: Electronic Proc. B-User Group Meeting FM 99. Volume 1709 of LNCS. (1999) 33–49
12. Dunne, S.: A theory of generalised substitutions. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings. Volume 2272 of Lecture Notes in Computer Science., Springer (2002) 270–290
13. Dawson, J.E.: Formalising generalised substitutions. In Schneider, K., Brandt, J., eds.: Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings. Volume 4732 of Lecture Notes in Computer Science., Springer (2007) 54–69
14. Jacquel, M., Berkani, K., Delahaye, D., Dubois, C.: Verifying B proof rules using deep embedding and automated theorem proving. Software and System Modeling **14**(1) (2015) 101–119