The Event-B Static Checker

Laurent Voisin
ETH Zürich

Stefan Hallerstede
ETH Zürich

August 31st, 2005

Contents

1 Introduction

All modelling items used in a formal B development are kept in the database kernel-component. This database is analysed by the static checker with respect to various properties the collection of modelling items must satisfy. When the static checker has accepted the database as being consistent, its items can be submitted to proof obligation generation and subsequent proof. In addition to marking modelling items as consistent the static checker computes auxiliary data structures to improve performance of all tasks that involve using items stored in the database.

Before we discuss the static checker in more detail we introduce some necessary terminology. Some of the definitions we make are left abstract here and refined in other places.

We refer to all entities that are contained in a formal development as *modelling item*. The following is a complete list of all modelling items.

simple modelling items:	
identifier,	*predicate*,
expression,	*substitution*
complex modelling items:	
elements:	
model,	*context*,
carrier set,	*constant*,
property,	*variable*,
invariant,	*variant*,
event,	*guard*,
local variable,	*action*,
witness,	*theorem*,
relations:	
model has abstraction,	*context has abstraction*,
model sees context,	*event has abstraction*,
context contains carrier set,	*context contains constant*,
context contains property,	*model contains variable*,
model contains invariant,	*model contains variant*,
model contains event,	*event contains guard*,
event contains local variable,	*event contains substitution*,
event contains witness,	*model contains theorem*,
context contains theorem,	
attributes:	
new event,	*model name*,
context name,	*variable name*,
carrier set name,	*constant name*,
property name,	*invariant name*,
event name,	*guard name*,
local variable name,	*theorem name*
property predicate,	*invariant predicate*,
guard predicate,	*theorem predicate*,
variant expression,	*action substitution*,
witness substitution	

Modelling items that are atomic and self-contained are called *simple modelling items*. Non-atomic modelling items whose structure is known to the database are called *complex modelling items*.

Among the simple modelling items *predicates*, *expressions*, and *substitutions* are also called *formulas*. The user enters simple items usually in textual form. Complex modelling items are entered by creating forms that need to be filled in subsequently. In the predicates and invariants are not the same thing: a predicate is a piece of unformatted text and an invariant is a database item that has a predicate and a name as attributes. In the full database complex modelling items are further distinguished into *elements*, *attributes*, and *relations*. The static checker is not aware of this distinction: it verifies all complex items in the same manner. Hence, we only use the generic term *item* in this text.

All modelling items must conform with the data structures used in the database. We call these the *minimal requirements* imposed on each modelling element. For instance, a *model* can only have one abstraction. Minimal require-

ments really define B developments as data-types, e.g. tree, or more generally hierarchical structures. A modelling item that satisfies the minimal requirements expressed in the meta-model is called *unchecked*. The term *unchecked* alludes to an item not yet having been verified by the static checker. The minimal requirements have a major impact on the GUI. The GUI must provide that the user can only enter items that satisfy the minimal requirements. If some items would not satisfy them, these items could not be stored in the database.

2 Architecture of the Static Checker

The static checker consists of three parts called *parser*, *graph-checker*, and *type-checker* (see Figure 1). The *parser* reads formulas that are given in textual form,

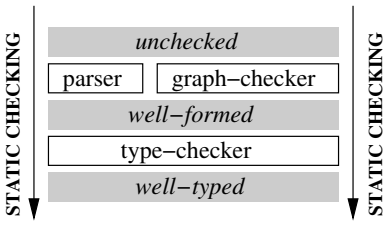

Figure 1: Layers of the static checker

and produces corresponding abstract syntax trees. The *graph-checker* analyses structural properties of, and relations of, modelling items such as *models* and *contexts*. The parser and the graph-checker are different components although they both check well-formedness of modelling items. Nonetheless, the partition into two components arose naturally: the parser only analyses formulas and does not have any knowledge of complex modelling items, and the graph-checker does not know anything about formulas or abstract syntax trees. The corresponding concepts are abstracted in the parts of the meta-model for the parser and the graph-checker respectively. The *type-checker* analyses and computes the types of all formulas that occur in the database. Modelling items that have passed static-checking are *well-formed* and *well-typed* and can be used by the proof obligation generator. Well-formedness is checked by the parser and the graph checker, and the type-checker checks whether modelling elements are well-typed. Initially all items are said to be *unchecked*. The following relationship must be maintained by the database between well-formed and well-typed modelling items: *All well-typed modelling items are well-formed*. Hence, the well-typed items are a subset of the well-formed items, and the well-formed items are a subset of the unchecked items.

2.1 Parts

We have specified the different parts of the static checker in formalisms that seemed best suited to express required properties, reason about them, and provide models that have an appropriate level for implementing the corresponding part of RODIN platform kernel-component.

The *parser* is modelled using the EBNF notation as is customary. There are standard implementation techniques and parser generators that can be used directly with EBNF notation. The parser also contains a feature to compute free and bound identifiers. This is expressed by means of an attributed grammar that can be used with the same standard tools as the EBNF notation.

The *graph-checker* is modelled in EventB. This has proven to be advantageous for expressing well-formedness properties and derivation of dependencies among the modelling items.

Scoping rules for identifiers are checked across the boundary between the parser and the graph-checker. The parser checks scoping rules within, say, a predicate, computing the sets of free and bound identifiers. The graph-checker uses the set of free identifiers computed by the parser to check if the corresponding item declarations are in the scope of the predicate.

The *type-checker* is also modelled by means of an attributed grammar. As opposed to the parser and the graph-checker, the type-checker uses all items of the database, i.e. simple modelling items and complex modelling items.

The static checker has two layers that group the three components *parser*, *graph-checker*, and *type-checker*. The boundaries of the layers describe the state of modelling items. Figure 1 shows the layers of the static checker. We identify the possible states of a modelling item with the boundaries in the layer schema of Figure 1. We say:

- An item that has not been parsed or graph-checked is in state *unchecked*;
- an item that has been parsed or graph-checked but not yet type-checked is in state *well-formed*;
- an item that has been type-checked is in state *well-typed*.

Hence, in the database each modelling item can be in one of three states: *unchecked*, *well-formed*, or *well-typed*. Remember, that being in state unchecked means, in fact, satisfying the minimal requirements. The boundaries shown in Figure 1 are only conceptual. It is possible (and intended) that different modelling items are in different states. However, each modelling item can only be in one state at a time. For this purpose modelling items are *tagged* with their state. Although different modelling items may carry different tags (but each item only one), the tags can not be attached arbitrarily marking progression of single modelling items through the three layers. The reason is that there are *structural dependencies* between modelling items. Structural dependencies are described in the minimal requirements, the well-formedness requirements, and well-typedness requirements of the meta-model.

2.2 Checked models

The file describing the checked model contains copies of invariants and theorems of abstractions and properties and theorems seen contexts and abstractions. In addition, it contains typing information produced by the type-checker together with the variables, carrier sets, and constants being typed.

2.3 Example

We give a simple example where modelling items are hierarchical, i.e. contain other modelling items. We use the following items: *predicates* "**P**", *invariants*

"**I**", *events* "**E**", and *guards* "**G**". The capital letters are used in the figure to represent items of the corresponding type. The state tags are represented by "u" for unchecked, "f" for well-formed, and "t" for well-typed. In the small example database of *predicates*, *invariants*, *events*, and *guards* we assume the following dependencies: *invariant* contains *predicate*, *event* contains *guard*, and *guard* contains a *predicate*. We read "contains" as "depends on".

> We require that a modelling item **X** may only pass from one state to the next state if all modelling items that **X** depends on have at least reached the next state.

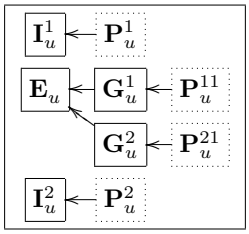

Figure 2: Items tagged *unchecked* and dependencies

In the database of items shown in Figure 2 the tags are shown as subscripts of the database items. We have given numbers to the items as superscripts in order to be able to distinguish them. The arrows signify "is contained in". Modelling items that are checked by the parser are enclosed by dotted boxes, and items that are checked by the graph-checker by solid boxes.

We present a sequence of valid states (leaving out some intermediate states) and comment on activities performed by the static checker. In state shown in

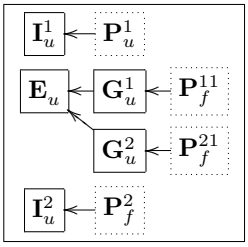

Figure 3: Predicates $\mathbf{P}^{11}$, $\mathbf{P}^{21}$, and $\mathbf{P}^2$ are well-formed

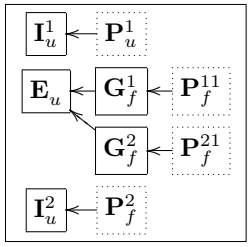

Figure 4: Guards $\mathbf{G}^1$ and $\mathbf{G}^2$ are well-formed

Figure 2 only the parser can be active because all *predicates* are *unchecked* and all other items depend on them directly or transitively. In Figure 3 the parser has succeeded checking *predicates* $\mathbf{P}^{11}$, $\mathbf{P}^{21}$, and $\mathbf{P}^2$. The parser also creates abstract syntax trees for these *predicates*. But this is not shown in the figure. We assume parsing $\mathbf{P}^1$ would fail. As a consequence the parser would produce a corresponding error message.

The graph-checker can now check the *guards* $\mathbf{G}^1$ and $\mathbf{G}^2$, and the *invariant* $\mathbf{I}^2$. We assume checking $\mathbf{I}^2$ would fail (and the graph-checker would produce an error message). Figure 4 shows the state of the database where checking $\mathbf{G}^1$ and $\mathbf{G}^2$ has succeeded.

In the next state (shown in Figure 5) we assume that graph-checking *event* **E** would have failed (and an error message would have been produced). Furthermore, we assume type-checking $\mathbf{P}^{21}$ and $\mathbf{P}^{2}$ would have succeeded, and type-checking $\mathbf{P}^{11}$ failed (and an error message produced). Finally the type-

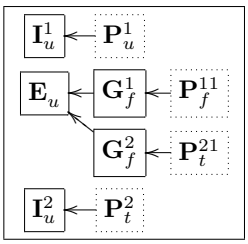

Figure 5: Predicates $\mathbf{P}^{21}$ and $\mathbf{P}^{2}$ are well-typed

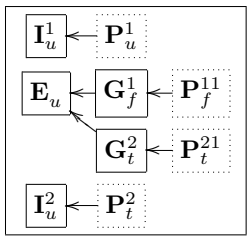

Figure 6: Guard $\mathbf{G}^2$ is well-typed

checker marks also the *guard* $\mathbf{G}^2$ as type-checked. This is all that is possible in this state because parsing *predicate* $\mathbf{P}^1$ has failed, graph-checking *event* **E** and *invariant* $\mathbf{I}^2$ has failed, and type-checking *predicate* $\mathbf{P}^{11}$ has failed. Figure 6 shows the final state. Note, that marking database items like *guards*, *events*, or even *models* as type-checked free us from searching through the database when this information is required. E.g. to find out whether all items in a *model* have passed type-checking, we need only look at the modelling item representing the *model*.

The proof obligation generator can only generate proof obligations for invariants and events that carry the subscript "t", i.e. none in the database shown in Figure 6.

3 Graph-Checker Specification

EventB developments, i.e. all items contained in it, form an acyclic graph-structure. This an properties related to the graph structure like use of variable names is verified by the graph-checker. The graph-checker takes into account formulas that have been parsed. Type-checking takes place after graph-checking has finished.

The graph-checker is specified in EventB. The graph structure is described in the invariant of the EventB model. The graph-checking is described by means of events *commit* that attempt to add items to a database DB_{wf} of well-formed elements, where items that satisfy the minimal requirements are kept in a database DB_{un} of unchecked items. We require that DB_{wf} is a subset of DB_{un}. That is, the graph-checker only works with items entered by the user; it does not add items or change the contents of DB_{un}. All failures to add an item to DB_{wf} result in error messages to the user. The error messages are described in the guards of the events *commit* that attempt to insert items into DB_{wf}. Dependencies between items in DB_{wf} are described by events *retract* that attempt to remove items from DB_{wf} maintaining well-formedness of DB_{wf}. The graph-checker works incrementally, i.e. it permits parts of a model to be unchecked while others are well-formed. The user interacts with DB_{un} being able to *add* or *remove* items to or from it.

3.1 Minimal Requirements

We state the minimal requirements for contexts and models as an example.

MIN 1 (CTX) *The contexts form a directed graph without self-loops where each node has exactly one outgoing edge.*

MIN 2 (MDL) *The models form a directed graph without self-loops where each node has exactly one outgoing edge.*

MIN 3 (MDL) *Models are related to contexts by the sees relationship. A model sees at most one context.*

⋮

3.2 Well-formedness Requirements

WFD 1 (CTX) *The contexts form a collection of disjoint trees.*

DEF 1 (CTX) *The child of a context C in a context tree is called a* refinement *of C. The parent of a context C in a context tree is called an* abstraction *of C.*

WFD 2 (MDL) *The models form a collection of disjoint trees.*

DEF 2 (MDL) *The child of a model M in a context tree is called a* refinement *of M. The parent of a model M in a context tree is called an* abstraction *of M.*

Contexts seen by models must be related to each other properly, i.e. have a similar the tree structure to the seeing models:

WFD 3 (MDL) *If a model sees some context C then the abstraction of the model must see the same context C or some abstraction of C.*

⋮

3.3 Implementation of Graph-Checking

The EventB model is used to implement the graph-checker. However, instead of manipulating a database shared by all models and contexts, it simply constructs a local copy of all necessary items (see Section 2.2) and inserts them into the well-formed database for the particular model or context. The advantage of this is that subsequent kernel components like the proof obligation generator can work concurrently on different models and contexts without interference.

4 Type-Checker Specification

Deliverable D3.2 specifies the notion of a well-typed formula [1, part VI, sec. 4.3] and states that a ill-typed formula is meaningless. As a consequence, we want that all the proof obligations that are generated from a model or a context are well-typed. Then, every generated proof obligation could be type-checked at generation time.

However, that would be very inefficient, as proof obligations usually share a lot of common sub-formulae. For instance, all proof obligations of a model contain the properties and theorems of seen contexts in hypothesis. Therefore, it seems wiser to check that the elements of a context (or model) satisfy some *sufficient conditions* to ensure that, later on, proof obligations generated therefrom are well-typed. It is the essence of the type-checker to check these sufficient conditions.

In the sequel, we first give these sufficient type-checking conditions, explaining from which proof obligation they are derived. We then expose the behavior of the type-checker when errors are encountered while type-checking.

4.1 Conditions to check

Firstly, as stated in the static-checker specification, type-checking is only attempted on well-formed models and contexts. That means that, when specifying type-checking conditions, one can rely on the model or context satisfying well-formedness conditions.

Secondly, as stated in [1], formula type-checking takes as input a typing environment (a function that maps identifiers to types). Its output is an indication of success or failure. In case of success, formula type-checking also produces a new typing environment which is a superset of the input typing environment. This output typing environment then contains type mapping for all identifiers that occur (free or bound) in the formula. In the sequel, we will call *resulting typing environment* the typing environment synthesized by formula type-checking, but with all bound variable types removed.

We will first define the type-checking conditions for a context. Then, we will examine models, looking first at global clauses (invariants, theorems and variant) and then at events.

4.1.1 Type-Checking a Context

Let's start with the simplest proof obligation (the one that contains the least number of predicates). This proof obligation is the well-definedness (WD) of the first property of a top-level context (labeled CTX_PRP_WD in the Proof Obligation Generator Specification). It contains no hypothesis and the goal is the WD lemma of the property.

As the property is well-formed, we know that the only identifiers that can occur free in it are carrier sets and constants declared in the same context. Then, in the worst case, its WD proof obligation contains the same free identifiers. Hence, to ensure that this proof obligation is well-typed, a sufficient condition is that the property is well-typed. That entails that the input typing environment for type-checking the property must contain the carrier sets defined in the context (each set is mapped to its powerset) and that the resulting typing environment contains the types for all constants that occur free in the property.

When examining the WD proof obligation of the second property, a similar reasoning leads us to use the previous resulting typing environment as input (because the first property appears in the hypothesis of the proof obligation) and then to apply formula type-checking to the second property, obtaining a maybe new typing environment as result. Applying the same scheme to successive properties of the context, we build incrementally larger typing environment.

Then, as the context is well-formed, we know that all constants occur free in some property. Hence, when the last property has been type-checked, our resulting typing environment will map all constants to a type.

As concerns theorems of a context, things are much simpler. In every proof obligation related to a theorem (CTX_THM_WD and CTX_THM), all properties occur in hypothesis. As a consequence, these properties define through type-checking the types of sets and constants, which are the only identifiers that can occur free in the theorem proof obligations. So, to ensure that the theorem proof obligations are well-typed, one just needs to type-check every theorem, using as input typing environment the one produced by the type-checking of the last properties.

Now, let's examine the case of a non top-level context, that is a context that refines another (abstract) context. Then, all properties and theorems of the abstract context (and its abstractions) will occur in hypothesis in our current context proof obligations. As a consequence, we will use for type-checking the full typing environment of the abstract context (that is the one obtained after type-checking all properties and theorems of the abstraction). This is the only change that we have to the above reasoning for specifying type-checking in a context.

We now have all what is needed to specify type-checking of a context, so let's formalize it. Assume we have a context C_n which refines a context C_{n-1}. For a top-level context, we denote it as C_1, assuming that it refines a dummy empty context C_0 to streamline things. Also, let's use $TE(C_n)$ to denote the typing environment obtained as the result of type-checking context C_n.

Finally, let's denote the objects of our C_n context as follows:

$$\begin{aligned} \text{Sets: } & S_1, S_2, \ldots, S_k \\ \text{Constants: } & c_1, c_2, \ldots, c_l \\ \text{Properties: } & P_1, P_2, \ldots, P_m \\ \text{Theorems: } & T_1, T_2, \ldots, T_p \end{aligned}$$

Then, type-checking of context C_n looks like the following:

$$\begin{aligned} \tau_0 &= TE(C_{n-1}) \\ \tau_1 &= \tau_0 \cup \{x \mapsto \mathbb{P}(x) \mid x : \{S_1, S_2, \ldots, S_k\}\} \\ \tau_2 &= \text{result of type-checking } P_1 \text{ with } \tau_1 \\ \tau_3 &= \text{result of type-checking } P_2 \text{ with } \tau_2 \\ &\vdots \\ TE(C_n) &= \text{result of type-checking } P_m \text{ with } \tau_m \\ &\quad \text{type-check } T_1 \text{ with } TE(C_n) \\ &\quad \text{type-check } T_2 \text{ with } TE(C_n) \\ &\quad \vdots \\ &\quad \text{type-check } T_p \text{ with } TE(C_n) \end{aligned}$$

The formulae above read as follows:

- First start with the typing environment of this context abstraction (τ_0).
- Then add the types for the carrier sets defined in this context, this gives τ_1.
- Then, type-check each property, collecting new types while proceeding ($\tau_2, \ldots, \tau_m$).
- The typing environment obtained after type-checking the last property is the typing environment of this context ($TE(C_n)$).
- Finally, type-check each theorem with the typing environment of this context.

4.1.2 Type-Checking Global Clauses of a Model

type-checking of global clauses (invariants, theorems, and variant) of a model is pretty similar to type-checking of properties and theorems of a context. The reasoning for proof obligations only related to global clauses is exactly the same. The proof obligations considered are:

MDL_INV_WD	MDL_THM_WD	MDL_THM	
REF_INV_WD	REF_THM_WD	REF_THM	REF_VAR_WD

We denote a model to type-check by M_m. It supposedly refines another model M_{m-1} (with the convention that an initial model is denoted M_1 and refines a dummy empty model M_0). Furthermore, model M_m sees a context C_n (with the convention that it sees C_0 in case of the absence of a SEES clause). Finally, we denote by TEM_m the typing environment obtained as the result of type-checking model M_m.

The contents of model M_m are:

$$\begin{aligned} \text{Variables: } & v_1, v_2, \ldots, v_k \\ \text{Invariants: } & I_1, I_2, \ldots, I_l \\ \text{Theorems: } & U_1, U_2, \ldots, U_p \\ \text{Variant: } & V \\ \text{Events: } & E_1, E_2, \ldots, E_q \end{aligned}$$

Then, the type-checking of global clauses of model M_m is formalized as follows:

$$
\begin{aligned}
\tau_0 &= TE(M_{m-1}) \\
\tau_1 &= \tau_0 \cup TE(C_n) \\
\tau_2 &= \text{result of type-checking } I_1 \text{ with } \tau_1 \\
\tau_3 &= \text{result of type-checking } I_2 \text{ with } \tau_2 \\
&\vdots \\
TE(M_m) &= \text{result of type-checking } I_l \text{ with } \tau_l \\
&\quad \text{type-check } U_1 \text{ with } TE(M_m) \\
&\quad \text{type-check } U_2 \text{ with } TE(M_m) \\
&\quad \vdots \\
&\quad \text{type-check } U_p \text{ with } TE(M_m) \\
&\quad \text{type-check } V \text{ with } TE(M_m)
\end{aligned}
$$

Note: Typing environment τ_1 is well-formed (i.e., a function) due to the well-formedness restrictions on the architectural links between models and contexts, and to the way typing environments are incrementally built for models and contexts.

4.1.3 Type-Checking Events of a Model

Events do not share common identifiers beyond those introduced at the model level (i.e., sets, constants and variables). As a consequence, every event can be type-checked in isolation. Also, the proof obligations for the model initialization are a subset of the proof obligations of regular events. Hence, in this section, we will consider the initialization to be a special kind of event and do not treat it specially.

The simplest proof obligations of an event concern well-definedness of guards (MDL_GRD_WD and REF_GRD_WD). Using the same reasoning as before, we induce that guards must be type-checked in their order of appearance and that, when all guards have been type-checked, the resulting typing environment shall contain the type of all local variables of the event (because the well-formedness of the event implies that every local variable appears in at least one guard).

Another kind of proof obligations that concerns guards is guard refinement (REF_GRD_REF). In these proof obligations, both the guards of the concrete event and those of the abstract event(s) appear. Moreover, the abstract and concrete events can declare local variables with the same name. In that case, these common variables are considered to represent the same data. As a consequence, they should have the same type. So, to ensure that relationship, the type-checking of concrete guards shall be started using the typing environment of the abstract event.

In the case of a merging of events (REF_GRD_MRG), well-formedness ensures that all abstract events declare the same local variables. However, one needs to check that the typing environment of the abstract events are compatible

(that is every local variable has the same type in all abstract events). Then, the common abstract typing environment is used when type-checking the concrete guards.

Then, to ensure that proof obligations about substitution well-definedness (MDL_EVT_WD and REF_EVT_WD) are well-typed, one needs to type-check every substitution using the typing environment obtained after type-checking all guards (as the latter appear in hypothesis of these proof obligations). Type-checking of a substitution consists in type-checking its before-after predicate. For that, one needs to add to the typing environment the type of the after variables (primed variables). Their type is the same as the type of their corresponding before variable (the unprimed one).

Finally, one needs also to type-check the witnesses provided within an event. Each witness is made of two parts. Its left-hand side contains the name of a local variable of the abstract event(s) or a double primed variable which corresponds to the after value of a variable in the abstract event(s). Its right-hand side is an expression the free identifiers of which are sets, constants, concrete global and local variables. So, to type-check witnesses one needs to build a typing environment made of the concrete and abstract events typing environment plus a typing environment that associates double primed variable to their type. Under this typing environment, a simple equality between the left-hand side and right-hand side is type-checked.

Now, let's formalize all that. Assume we have in model M_m an event F which refines abstract events E_1, E_2, ..., E_p. We denote by L the set which contains all the local variables of the abstract events E_i (well-formedness ensures that they all declare the same local variables) and by K the set of the local variables of the concrete event. The set of global variables of model M_i is denoted by V_i. The guards of event F are denoted as G_1, G_2, ..., G_g, its substitutions by S_1, S_2, ..., S_s and its witnesses have left-hand side l_1, l_2, ..., l_l and right-hand side r_1, r_2, ..., r_l. Finally, we denote by $TE(F)$ the typing environment of event F.

The type-checking of event F is formalized as:

$$
\begin{aligned}
\tau_0 &= TE(M_m) \\
&\quad \text{check that } \forall i, j \cdot TE(E_i) = TE(E_j) \\
\tau_1 &= \tau_0 \cup ((K \cap L) \lhd TE(E_1)) \\
\tau_2 &= \text{result of type-checking } G_1 \text{ with } \tau_1 \\
\tau_3 &= \text{result of type-checking } G_2 \text{ with } \tau_2 \\
&\quad \vdots \\
TE(F) &= \text{result of type-checking } G_g \text{ with } \tau_g \\
\theta &= TE(F) \cup (prime^{-1} ; (V_m \lhd TE(M_m))) \\
&\quad \text{type-check } BA(S_1) \text{ with } \theta \\
&\quad \text{type-check } BA(S_2) \text{ with } \theta \\
&\quad \vdots \\
&\quad \text{type-check } BA(S_s) \text{ with } \theta \\
\zeta &= TE(F) \cup (K \lhd TE(E_1)) \cup (dprime^{-1} ; (V_{m-1} \lhd TE(M_{m-1}))) \\
&\quad \text{type-check } l_1 = r_1 \text{ with } \zeta \\
&\quad \text{type-check } l_2 = r_2 \text{ with } \zeta \\
&\quad \vdots \\
&\quad \text{type-check } l_l = r_l \text{ with } \zeta
\end{aligned}
$$

where BA maps a substitution to its before-after predicate, *prime* (resp. *dprime*) is a relation that maps an unprimed identifier to its primed (resp. double primed) variant.

4.2 Error Recovery

In the previous sections, we described type-checking with the implicit assumptions that all elements were found correct. But, it can happen that some element produce a type-check error. We examine here the consequence of such an error.

When looking to the calls to the formula type-checker that appear above, one can easily see two kinds of calls. In one kind, type-checking produces an output typing environment which is used later on (e.g., type-checking of a context property). In the other kind, one only checks a formula but no new typing environment is produced (e.g., type-checking of a theorem). Let's first examine the second case, as it is the easier one to tackle with.

When no new typing environment is expected, the output of formula type checking is either success or failure. In case of success, the type-checked element is added to the type-checked database. In case of failure, the element is just ignored. It is thus not added to the type-checked database. This approach is sound, as there is no dependence on this element in proof obligations generated afterwards.

When a new typing environment is expected, the output of formula type-checking is twofold. Firstly, it can be either success or failure. Secondly, and only in case of success, an output typing environment is also produced. So, if

type-check succeeds, the checked element is added to the type-checked database and type-checking proceeds on with the new typing environment just obtained. In case of failure, the element is ignored and not added to the type-checked database.

However, if a single failure is encountered when type-checking a set of elements like the properties of a context, the final typing environment (the one obtained after type-checking the last property) must be checked for completeness (all constants must occur in it): Because of that failure, relying on the well-formedness of the context to ensure that all constants have a type doesn't work anymore, so an additional check is needed. In addition, the untyped constants are marked with an error flag and not added to the type-checked database. Subsequently, any element in which an erroneous constant occurs is considered to fail type-check. It will not be added to the type-checked database.

In fact, what was described for constants (whose typed are inferred from properties), applies as well to global variables (typed by invariants) and local variables (typed by guards). In the specification above, everywhere the *TE* operator is defined for some element, this typing environment must be checked for completeness and all data names which should occur in it but do not are flagged as erroneous.

This approach allows for generating as many well-typed proof obligations as possible, despite some type errors. Most of the times, these proof obligations will be incomplete, lacking some hypothesis, but, hopefully, they will contain enough information to be discharged by the prover.

References

[1] Rodin Deliverable D3.2 *Event-B Language*.