Event-B Code Generation: Type Extension with Theories

A. Edmunds[1], M. Butler, I. Maamria, R. Silva, and C. Lovell

[1]University of Southampton, UK
ae2@ecs.soton.ac.uk

Abstract. The Event-B method is a formal modelling approach; our interest is the final step, of generating code for concurrent programs, from Event-B. Our Tasking Event-B tool integrates Event-B to facilitate code generation. The theory plug-in allows mathematical extensions to be added to an Event-B development. When working at the implementation level we need to consider how to translate the newly added types and operators into code. In this paper, we augment the theory plug-in, by adding a *Translation Rules* section to the tool. This enables us to define translation rules that map Event-B formulas to code. We illustrate the approach using a small case study, where we add a theory of arrays, and specify translation rules for generating Ada code.

1 Introduction

Using Event-B [1] and Tasking Event-B, we have the ability to model and implement single and multi-tasking software systems, see [2]. It may be the case that we need some new mathematical type, and in many cases, the type will need to be implemented. In this paper we describe, using an example, how new types can be added to an Event-B Theory. We then describe the tool's new translation rule feature, which we use to define translation rules for generating Ada code. The rules describe the mapping from Event-B types, and mathematical notation, to implementable code fragments. This work has been undertaken as part of the EU DEPLOY [4] project.

The basic structural features of Event-B are contexts and machines. Contexts describe the static features of a system using sets and constants. Machines are used to describe the variable features of a system in the form of state variables and guarded events; system properties are specified using the invariants clause. Theories for Rodin are described in [3] where we introduce mathematical extensions; with this, we can add new types, operators, and rules. Rules, such as rewriting, compare a source against patterns defined in the theory rule-base. When a pattern matches with the source, the source is replaced by new elements, as determined by the pattern. In the work described in the paper, we add the ability to specify translation rules for code. This uses pattern matching, in a similar way; but, instead of initiating a substitution of new elements in place of old, we generate text for use in the main code generator. To facilitate the specification of new rules one can introduce Type Parameters, and specify Datatypes.

Users can also introduce new operators and theorems. Proof obligations are generated to verify the soundness of the rules, and the prover is augmented with the new rules, when the theory is *deployed*. In our work, we extend the theory with translator rules.

1.1 An Array Theory

The first step is to create a new theory of arrays, we introduce an array of type T. The array is a new operator, which takes a powerset of type T as an argument. The array has the following definition, using set-notation, where n is the length of the array.

$$array(s : \mathbb{P}(T)) \triangleq \{n, f \cdot n \in \mathbb{Z} \wedge f \in 0 \mathinner{..} (n-1) \rightarrow s | f\}$$

Since we have a low-level specification we consider implementation issues: generally, arrays are fixed-length implementations. We introduce *arrayN*, parametrized by n, which fixes the array length by stating $card(s) = n$.

$$arrayN(n : \mathbb{Z}, s : \mathbb{P}(T)) \triangleq \{a | a \in array(s) \wedge card(s) = n\}$$

The array constructor operator *newArray* has an integer parameter n, representing the array length; and, additionally a value x, of type T for initialising the array elements. The array construction operator has the following definition,

$$newArray(n : \mathbb{Z}, x : T) \triangleq (0 \mathinner{..} (n-1)) \times \{x\}$$

Additonally, *newArray* requires a well-definedness condition, $n \in \mathbb{N}$. For the array update, we have the definition,

$$update(a : array(T), i : \mathbb{Z}, x : T) \triangleq a \mathbin{\lhd\!\!\!-} \{i \mapsto x\}$$

update has the well-definedness condition $i \in 0 \mathinner{..} (card(a) - 1)$. We can see that array a is updated with value x at index i. The case study, which we use to illustrate the extension mechanism and the link to code, omits irrelevant detail.

2 An Event-B Model

In the following model, we make use of the array operator that we have just introduced. In the invariant, we type *cbuf* as an array of size *maxbuf* of integers. *maxbuf* is a constant defined in a seen context. The second parameter defines the element types, which in this case are integers. In the Initialisation event, we specify the size, and initial value for the array elements in the clause *act1*. In our model, we initially set *maxbuf* elements to be zero.

```
variables cbuf  a  b
invariants
 @inv1   cbuf ∈ arrayN(maxbuf, ℤ)
 @inv2   ...
initialisation
 @act1   cbuf := newArray(maxbuf, 0)
 @act2   ...
```

An example of the update to the array can be seen in the following *Put* event, which inserts an element into the array in action *act2*.

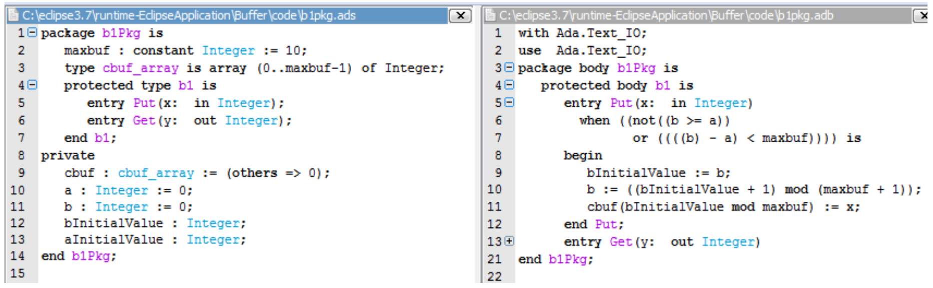
```
event  Put ≜
any x
where
 @grd1   x ∈ ℤ
 @grd2   b ≥ a ⇒ b − a < maxbuf
then
 @act1   b := (b + 1) mod (maxbuf + 1)
 @act2   cbuf := update(cbuf, b mod maxbuf, x)
end
```

3 Adding Translation Rules

The next step is to add translation rules to the theory that defines arrays. We add the Ada *Translator Target* section, shown below, and use this to define the translation of the newly introduced operators. Metavariables (variable patterns) are introduced to facilitate type inference, and pattern matching during translation. Using the rules defined in the *Translator Rules* section, we match the patterns in the following way to determine which translation is applicable. We specify the operator to be matched on the left side of the rule (left of the $\mapsto$ operator), and the translated text, on the right side. Since there is no formal link between the pattern on the left side and text output on the right side of the rule, we use visual inspection to verify that the rule is correct.

```
Translator Target: Ada
Metavariables
 s ∈ ℙ(T),   n ∈ ℤ,   a ∈ ℤ↔ T,   i ∈ ℤ,   x ∈ T
Translator Rules
  trns1: ...
  trns2: a = update(a, i, x) ⤇ a(i) := x
  trns3: newArray(n, x) ⤇ (others => x)
Type Rules
  typeTrns1: arrayN(n, s) ⤇array(0 .. n − 1) of s
```

In the example shown, the array *update* operator maps to the Ada array assignment $a(i) := x$. The construction operator *newArray* provides the initial values in parameter x, and maps to the Ada clause (*others* $=>$ x) which sets all elements (using *others*) of an array to x. In addition to translation rules, we can add type rules; these are used to map the type, as defined in the theory, to an implementable type for use in the generated code. In the type rule *typetrans1* we specify that the type $arrayN(n, s)$ should be mapped to the Ada type clause $array(0\,..\,(n-1))\ of\ s$.



C:\eclipse3.7\runtime-EclipseApplication\Buffer\code\b1pkg.ads

```
1  package b1Pkg is
2     maxbuf : constant Integer := 10;
3     type cbuf_array is array (0..maxbuf-1) of Integer;
4     protected type b1 is
5        entry Put(x:  in Integer);
6        entry Get(y:  out Integer);
7     end b1;
8  private
9     cbuf : cbuf_array := (others => 0);
10    a : Integer := 0;
11    b : Integer := 0;
12    bInitialValue : Integer;
13    aInitialValue : Integer;
14 end b1Pkg;
15
```

C:\eclipse3.7\runtime-EclipseApplication\Buffer\code\b1pkg.adb

```
1  with Ada.Text_IO;
2  use  Ada.Text_IO;
3  package body b1Pkg is
4     protected body b1 is
5        entry Put(x:  in Integer)
6          when ((not((b >= a))
7                 or ((((b) - a) < maxbuf)))) is
8        begin
9           bInitialValue := b;
10          b := ((bInitialValue + 1) mod (maxbuf + 1));
11          cbuf(bInitialValue mod maxbuf) := x;
12       end Put;
13       entry Get(y:  out Integer)
21 end b1Pkg;
22
```

Fig. 1. Generated Ada Code

4 Conclusion

We used the our tool [5] to generate the code shown in Fig. 1. The rules allow translation of operators, and type definitions, to implementation constructs. We can now see how the Event-B relates to the generated code. In line 3 of the code, on the left hand side of the figure, we see an Ada type declaration statement. This results from applying *typeTrns1* to the invariant *inv1*. The translation of the type s, i.e. the mapping of the Event-B $\mathbb{Z}$, to the Ada *Integer* type, is handled by the pre-defined Ada theory, and not shown here. In Ada, we must 'instantiate' the *cbuf_array* type, so in line 9 we declare *cbuf* to be of type *cbuf_array*, and initialise the values. The translator makes use of the translation rule *trns3*, matching the pattern $arrayN(n, s)$ with the assignment of the initialisation action *act1*. The update rule has also been translated in line 11 on the right side of Fig. 1, which uses *trns2* applied to *act2* of the *Put* event.

References

1. J. R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.
3. I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *ABZ2010*, February 2010.
4. The DEPLOY Project Team. Project Website. at http://www.deploy-project.eu/.
5. The Deploy Wiki Website - Code Generation Activity. at http://wiki.event-b.org/index.php/Code_Generation_Activity.