EB2C : A Tool for Event-B to C Conversion Support

Dominique Méry, Neeraj Kumar Singh

EB2C : A Tool for Event-B to C Conversion Support

Dominique Méry
Université Henri Poincaré Nancy 1
LORIA, BP 239, 54506
Vandoeuvre lès Nancy, France
Email: mery@loria.fr

Neeraj Kumar Singh
Université Henri Poincaré Nancy 1
LORIA, BP 239, 54506
Vandoeuvre lès Nancy, France
Email: singhnne@loria.fr

***Abstract*—To use of formal model effectively in formal method based development process, it is highly desirable that the formal specification be converted to C code, a de facto standard in many industrial application domains, such as medical, avionics and automotive control. In this paper we present the design methodology of a tool that translates an Event-B formal specification to equivalent C code with proper correctness assurance.**

***Keywords*-C, Formal model, Event-B, Proof-based refinement**

I. Introduction

Recently Medical, Avionics and Automotive industries are leaning more and more towards formal method-based development of safety-critical software. In embedded system community, formal method-based development implies verification and validation, and generated proof obligation ensure proof of correctness of a system model. The auto code generators associated with formal development tools can generate software codes from the formal specifications, thus enabling model developer to generate source code automatically without knowing the target language syntax. Proof-based development methods [1] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the (closed) system under development. Details and design choices are introduced in an incremental way. The correctness between two levels is ensured by refinement proofs. It also maintains very well refinement techniques that can transform an abstract and non-deterministic specification into a concrete, deterministic system model, in several stages. When implementations are aimed, refinement leads to a last level which describes, in some way, the expected behavior. Refinement-based model development is very popular in many industrial application domains. It is considered as a de facto standard in very complex systems such as in medical, avionics and automotive domains. Medical, Avionics and Automotive industries are already in possession of large formal specifications developed in formal language, which are not able to generate source code in C language [2] automatically. Realizing the benefit of adhering to automatic source code generation in their formal development process, these industries are now interested in embracing automatic source code generation approaches in their new projects; thereby having source codes for all of their old formal specifications for the ease of reuse and maintenance.

In this paper, we describe an approach to build a tool which will be useful to translate Event-B [1] formal specification model to an equivalent C code function. The implementation of translation is described in the form of a *toolchain*. We also provide a mechanism to ensure the correctness of such a translation. From Classic-B [3] notation to C language translation tool has been developed by D. Bert, et al. [4], in which models are restated in an intermediate language "B0", and then converted to finally in C language. A pioneering work for automatically translate subset of Event-B formal notation of MIDAS [5] specification in C language is proposed by S. Wright [5]. The shortcoming of the work is that the tool shows only for simple translation of formal concrete machines. Moreover, in Wright's work, handling of constants, axioms, enumerated sets and functions are not covered in translation process. In our work, we provide complete translation for them. For large generated source code, optimization using events scheduling and translation correctness assurance are of prime importance. We show how, event scheduling is resolved by refinement approach. Proper correctness assurance of generated C code from Event-B modeling language is verified using the code verification tool.

II. Description of the Tool

The translation process consists in transforming the concrete part of an Event-B project into a semantically equivalent text written in C programming language. We propose an architecture for the Event-B translator. Figure-1 depicts the overall architecture of the tool. The tool is called EB2C. This tool has mainly four components: Pre-processing, Event-B to C translator, code optimization and code verification. The input of the translator tool is a Rodin project [6] files containing formal specification in Event-B modeling language. To generate C code for an Event-B model we use Eclipse development framework for developing a plug-in in the Java language. The Pre-processing takes an Event-B project and introduce C context file to provide deterministic range for all kind of data types and makes an Event-B model

Event-B type	Formal Range	C type
tl_int16	$-2^{15}..2^{15}-1$	int
tl_uint16	$0..2^{16}-1$	unsigned int
tl_int32	$-2^{31}..2^{31}-1$	long int
tl_uint32	$0..2^{32}-1$	unsigned long int

Table I
INTEGER BOUNDED DATA TYPE DECLARATION IN CONTEXT FILE

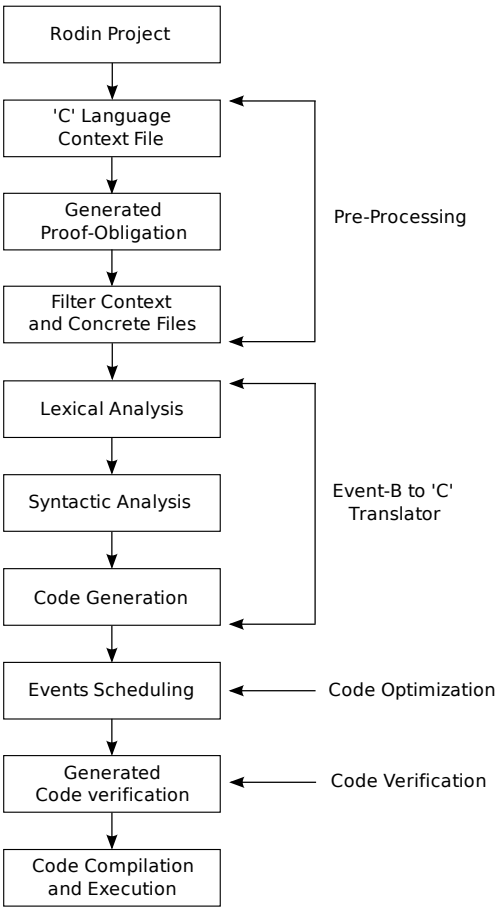

Figure 1. Architecture of Translation Tool

deterministic. Table-I shows bounded integer data types of C context.

The next two sub-steps of the Pre-processing is to prove all generated proof obligations and filter context files and concrete machines files from selected Rodin project [6]. An Event-B to C translator takes set of context and concrete filtered files. In this translation process our aim is to generate a C code file for an Event-B concrete file using Event-B grammar through syntax-directed translation. The translator generates separate functions for these Event-B events. We consider Event-B formal notations available at [1]. We capture Event-B grammar in a Abstract Syntax Tree (AST). For the production rules of the Event-B grammar we designed the appropriate algorithm with appropriate parameters to generate desired C codes in the targeted file with ".c" extension during the parsing process. The translator successfully translates an Event-B model with the following components: Constants, Enumerated sets, Functions, Variables, Arrays, Parameters, Events, Guards and Actions with arithmetic and logical operations. At present the translator cannot translate a sets, sets operations and relation over sets. Event-B has a rich set of formal model elements which cover formal modeling as abstraction, which is not applicable for code translation. We have closely examined the Event-B grammar, and the observational equivalence between Event-B and C programming language types is done as follows:

Event-B types	C Language types
Enumerated sets	Enumerated types
Basic integer sets	Predefined integer types
Event-B array types	C array types
Function	C function structure

Table II shows a set of Event-B syntax to an equivalent C programming language. This table shows a list of supported Event-B syntax. All constants defined in a model's context must be replaced with their literal values. This translation tool supports conditional, arithmetic and logical expression of formal model and translates into equivalent C language code.

An Event-B enumerated sets is equivalent to the C programming language enumerated types. It is very easy to translate into C programming language equivalent form due to equivalent semantical structure.

The links between Event-B and C programming language for integer values have been considered as crucial for the efficiency of the generated code and for the correctness of the translation. So, the solution is provided in first level of Pre-processing phase by introducing C programming language context and it is able to interface very tightly with Event-B integer types and C programming language integer types.

The links between Event-B arrays and C programming language arrays are not straightforward. In Event-B, arrays correspond to total functions whereas in C programming language, they correspond to a contiguous zone of memory (coded as the beginning address of the array and its size). However, it is easy to do a semantical correspondence between an array element *arr(i)* in Event-B and the value at the location *arr[i]* in C programming language.

The links between Event-B functions and C programming language functions are also very ambiguous. Translation tool only supports total functions of Event-B into equivalent corresponding C programming language functions. However, it is easy to do a semantical correspondence between a function passing parameters in a C programming language is equivalent to the elements of left side of total functions symbol ($\rightarrow$) and output of the C programming language function correspond to the right hand side of the

Event-B	C Language	Comment
n..m	int	Interger type
x ∈ Y	Y x;	Scaler declaration
x ∈ tl_int16	int x;	'C' Context declaration
x ∈ n..m → Y	Y x [m+1];	Array declaration
x :∈ Y	/* No Action */	Indeterminate initialization
x : | Y	/* No Action */	Indeterminate initialization
x = y	if(x==y) {	Conditional
x ≠ y	if(x!=y) {	Conditional
x < y	if(x<y) {	Conditional
x ≤ y	if(x<=y) {	Conditional
x > y	if(x>y) {	Conditional
x ≥ y	if(x>=y) {	Conditional
(x>y) ∧ (x≥z)	if ((x>y) && (x>=z) {	Conditional
(x>y) ∨ (x≥z)	if ((x>y) || (x>=z) {	Conditional
x := y + z	x = y + z;	Arithmetic assignment
x := y - z	x = y - z;	Arithmetic assignment
x := y * z	x = y * z;	Arithmetic assignment
x := y ÷ z	x = y / z;	Arithmetic assignment
x := F(y)	x = F(y);	Function assignment
a := F(x↦y)	a = F(x, y);	Function assignment
x := a(y)	x = a(y);	Array assignment
x := y	x = y;	Scalar action
a := a ⩤ {x↦y}	a(x) = y;	Array action
a := a ⩤ {x↦y} ⩤ {i↦j}	a(x)=y; a(i)=j;	Array action
X⇒Y	if(!X || Y){	Logical Implication
X⇔Y	if((!X || Y) && (!Y || X)){	Logical Equivalence
¬x<y	if(!(x<y)){	Logical not
x ∈ ℕ	unsigned long int x	Natural numbers
x ∈ ℤ	signed long int x	Integer numbers
∀	/* No Action */	Quantifier
∃	/* No Action */	Quantifier
Sets	/* No Action */	Sets operations

fun ∈ ℕ × ℕ → ℕ	long int fun(unsigned long int $arg1$, unsigned long int $arg2$)
	{
	//TODO: Add your Code
	return;
	}
		Function Definition

Table II
EVENT-B TO C TRANSLATION SYNTAX

total functions symbol (→) in Event-B. So, this step of function translation generates the function structure into C programming language. If body of the function is defined in Event-B as in form of predicate then translation tool translate equivalent predicate into function body of the C programming language.

An Event-B model is a collection of interdependent events. The translator takes each of these events one by one and converts them to an equivalent C function. So we have separate C functions for each events. This C code file contains appropriately generated constants, local and global variables, arrays, functions, and events which are generated from Event-B models using lexical and syntactic analysis.

Translation tool provides a recursive process to generate source code for each event of the Event-B specification into C programming language. Translation tool always checks for "null" event (i.e. guard of false condition), never generates the source code for that event and inserts suitable comment into the source code for traceability purpose. This automatic reduction is performed to avoid generation of unreachable run-time code.

In Event-B specification, there are two kind of variables: global variables and local variables. Global variables are derived directly from VARIABLES statements of the concrete machine and all these variables have global scope. Local variables are derived from the ANY statement of the particular event, and are entirely local to the corresponding function. Therefore no parameters are passed to C programming language function. The function returns a boolean value, signaling whether the event has been triggered to its calling environment at run-time. After generation of the function header, all local variable declarations are inserted at the beginning of the function, giving them scope across the whole function.

In Event-B guard handling is very ambiguous due to different meaning, such as local variables type definition, the assignment of a value to a local variable, condition statements using negation (¬), conjunction (∧), disjunction (∨), implication (⇒) and equivalence (⇔) operators. Therefore, for handling so much complex situation, we introduce a recursive algorithm for parsing complex guards and separate each element of guard for translation purpose. Thus each

guard must be automatically analyzed to resolve this ambiguity from context information. For example an implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$) operator, we rewrite the predicate in equivalent form using conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$) operators, the equals relation may signify an assignment or equality comparison, and the precise meaning (and hence the resulting translation) deduced from the type and scope of its operands. A further ambiguity that must be resolved is the meaning of a functional-image relation, which may be used to model a data array or an external function. Once the guards of the event have been classified, those conferring local variable type information are used for variable declarations in the function, and remaining guards are used to generate local assignment and conditional statements. Local variable type information is derived in a similar fashion as the global variables from the guard information instead of using INVARIANT.

All condition guards are placing in the function as nested statements, using directly translated conditional and local variables declared within nested scope ranges. After insertion of all conditional guards, we provide translation for remaining local assignments with a comment. The meaning of functional-image statements within the model is automatically resolved to an array if the mapping is a global variable, otherwise a call to an uninterpreted function is inserted.

The next sub-stage of event translation is the actions translation. In Event-B, actions are triggered in a concurrent manner and state that all state modification in the actions is only valid in the entire event post-condition. Therefore, dependency checks must be performed to ensure that any state variable used as an action assignee has not already been modified to its post-condition prior to use. A similar way of parsing is applied on Event-B action statement as a guard statement. An action translation supports assignments to scalar variables, override statements acting on array-type variables and arithmetic complex expression. The translation tool translates all Event-B actions into equivalent C programming language source code.

The code optimization is used to optimize the events scheduling for calling functions using refinement approach. An incremental refinement-based structure of events within an Event-B model exploit to recursively generate nesting calling functions corresponding to the abstract events. An abstract level guards are forming a group of concrete events. An event group is inserted for execution in place of multiple events, improving run-time performance.

We propose two techniques to trigger all translated events. First is a calling function *"Iterate"* implements a continuous iteration of translated C programming language functions of the Event-B model, in the same order, defined by their position in the Event-B model. Second technique is to optimize the calling order of the events. From the event scheduling, we have got the code structure that help to make a group of all concrete events. Each group of events are triggered by main *"Iterate"* function.

The code verification is very important to verify correctness of translated code. We use a methodology called code verification to validate a C code with respect to the formal specification from which it has been generated. Main objective to use code verification to test the C program with proper coverage criteria and to ensure its correctness. In this way, C codes for all the Event-B formal models are successfully generated.

III. Conclusion

Generating C code from Event-B formal specification is of prime importance to medical, avionics and automotive industries. This importance arises from lots of advantages that can be gained using formal methods based development process in the design, development and maintenance of embedded software. The medical, avionics and automotive industries are willing to convert all their formal specifications to equivalent C codes, and then leverage formal methods based development process for further work. The approach we discussed in this paper will be of great help towards this goal. We have implemented the translator covering a large subset of Event-B modeling language. We have also completely automated the test of translated code generation process for correctness assurance. The translator tool has been tested on a number of Event-B formal models of a cardiac pacemaker system with encouraging results. In near future, we have planed to extend this translation tool to handle *Sets*, *Relation* and other kind target programming language such as *ladder logic*, so that this translation tool can be used by all industrial areas, whereas formal verification and validation are primary techniques to develop a system.

Acknowledgement Work of Dominique Méry and Neeraj Kumar Singh is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche. Neeraj Kumar Singh is supported by grant awarded by the Ministry of University and Research.

References

[1] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] B. W. Kernighan and D. Ritchie, *C Programming Language*. Prentice Hall, 1988, iSBN-100131103628.

[3] J.-R. Abrial, *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996.

[4] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin, *Adaptable Translator of B Specifications to Embedded C Programs*. FME 2003, Springer, 2003, pp. 94–113.

[5] S. Wright, "Automatic generation of c from event-b," in *Workshop on Integration of Model-based Formal Methods and Tools*, February 2009. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000990.pdf

[6] Project RODIN, "Rigorous open development environment for complex systems," http://rodin-b-sharp.sourceforge.net/, 2004.

[7] Dominique Méry and Neeraj Kumar Singh, "Functional behavior of a cardiac pacing system (in Press)," *International Journal of Discrete Event Control Systems*, vol. 1, 2010.

IV. Appendix

We use our developed translation tool to generate source code in 'C' language from Event-B modeling language of a cardiac pacemaker specification. The cardiac pacemaker system is a medical device which uses electrical impulses, delivered by electrodes contacting the heart muscles, to regulate the beating of the heart. The primary purpose of a pacemaker is to maintain an adequate heart rate, either because the heart's native pacemaker is not fast enough, or there is a block in the heart's electrical conduction system [7]. This case study completes a formal development of the cardiac pacemaker using Event-B. We use our developed translation tool to generate source code in 'C' language from cardiac pacemaker Event-B specification. The generated code is 90% equivalent to the handwritten code. Remaining 10% coding is additional coding style which provided by the developer in the pacemaker system using some other language. So, we believe that our translation tool is correct. An example of an Event-B event and its 'C' translation is given as follows:

Actuator ON and OFF events of Event-B

EVENT Actuator_ON_V
WHEN
grd1 : $PM\_Actuator\_V = OFF$
grd2 : $(sp = Pace\_Int) \lor (sp < Pace\_Int \land AV\_Count > V\_Blank \land AV\_Count \geq FixedAV)$
grd3 : $sp \geq VRP \land sp \geq PVARP$
THEN
act1 : $PM\_Actuator\_V := ON$
act2 : $last\_sp := sp$
END

EVENT Actuator_OFF_V
WHEN
grd1 : $PM\_Actuator\_V = ON$
grd2 : $AV\_Count\_STATE = TRUE$
grd3 : $PM\_Sensor\_A = OFF$
grd4 : $PM\_Actuator\_A = OFF$
grd5 : $sp \geq VRP \land sp \geq PVARP \land sp \geq URI$
grd6 : $(sp = Pace\_Int) \lor (sp < Pace\_Int \land AV\_Count > V\_Blank \land AV\_Count \geq FixedAV)$
THEN
act1 : $PM\_Actuator\_V := OFF$
act2 : $AV\_Count\_STATE := FALSE$
act3 : $AV\_Count := 0$
act4 : $PM\_Sensor\_V := OFF$
act5 : $sp := 1$
act6 : $Thr\_A := 0$
act7 : $Thr\_V := 0$
act8 : $Thr\_A\_State := FALSE$
act9 : $Thr\_V\_State := FALSE$
END

Translated C code

```
BOOL Actuator_ON_V(void)
{
    /* Guards No. 1 */
    if(PM_Actuator_V == OFF){

    /* Guards No. 2*/
    if((sp == Pace_Int) ||
    ((sp < Pace_Int) &&
    (AV_Count > V_Blank) &&
    (AV_Count >= FixedAV))) {

    /* Guards No. 3 */
    if((sp >= VRP) &&
    (sp >= PVARP) {

    /* Actions */
    PM_Actuator_V = ON;
    last_sp = sp;

    return TRUE;
    }}}
    return FALSE;
}
```

```
BOOL Actuator_OFF_V(void)
{
    /* Guards No. 1 */
    if(PM_Actuator_V == ON){

    /* Guards No. 2*/
    if(AV_Count_STATE == TRUE){

    /* Guards No. 3*/
    if(PM_Sensor_A == OFF){

    /* Guards No. 4*/
    if(PM_Actuator_A == OFF){

    /* Guards No. 5*/
    if((sp >= VRP)&&(sp > PVARP)&&(sp >= URI)){

    /* Guards No. 6*/
    if((sp == Pace_Int) ||
    ((sp < Pace_Int) &&
    (AV_Count > V_Blank) &&
    (AV_Count >= FixedAV))) {

    /* Actions */
    PM_Actuator_V = OFF;
    AV_Count_STATE = FALSE;
    AV_Count = 0;
    PM_Sensor_V = OFF;
    sp = 1;
    Thr_A = 0;
    Thr_V = 0;
    Thr_V_State = FALSE;
    Thr_A_State = FALSE;

    return TRUE;
    }}}}}}
    return FALSE;
}
```