From Event-B to Verified C via HLL

Ning Ge, Arnaud Dieumegard, Eric Jenn, Laurent Voisin

To cite this version:

Ning Ge, Arnaud Dieumegard, Eric Jenn, Laurent Voisin. From Event-B to Verified C via HLL. 2016. hal-01387137

HAL Id: hal-01387137
https://hal.science/hal-01387137

Preprint submitted on 25 Oct 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

From Event-B to Verified C via HLL

Ning Ge[1,a], Arnaud Dieumegard[1], Eric Jenn[1,b], and Laurent Voisin[2]

[1] IRT-Saint Exupéry, Toulouse, France
ning.ge|arnaud.dieumegard|eric.jenn@irt-saintexupery.fr
[2] Systerel, Aix-en-provence, France
laurent.voisin@systerel.fr

Abstract. This work addresses the correct translation of an Event-B model to C code via an intermediate formal language, HLL. The proof of correctness follows two main steps. First, the final refinement of the Event-B model, including invariants, is translated to HLL. At that point, additional properties (e.g., deadlock-freeness, liveness properties, etc.) are added to the HLL model. The proof of the invariants and additional properties at the HLL level guarantees the correctness of the translation. Second, the C code is automatically generated from the HLL model for most of the system functions and manually for the remaining ones; in this case, the HLL model provides formal contracts to the software developer. An equivalence proof between the C code and the HLL model guarantees the correctness of the code.

Keywords: Event-B, Code generation, C, HLL, S3, Property proof, Equivalence proof

1 Introduction

Event-B [1] is a formal notation and method for the *correct-by-construction* development of systems. In this method, a system is developed through a sequence of refinements, the consistency of which is formally proved. Event-B is based on first-order logic, typed set theory and integer arithmetic. As an integrated design environment for Event-B, the Rodin platform [14] provides support for refinement and mathematical proof of the consistency between refinements and of system-specific properties such as safety properties. In this work, we consider that the last refinement developed and proved with Event-B is the starting point for software development: the Event-B model is used as a specification for the implementation.

Translating a correct specification to a correct implementation is challenging. Existing works [30,9,26,20,25,10,21] have identified the following main issues for this translation, and have proposed partial solutions. (1) It is necessary to restrict the Event-B model to a well-defined subset in order to generate code for a particular programming language. (2) When multiple events are enabled, then an event is chosen non-deterministically to be executed. A particular schedule needs to be defined to remove such non-determinism. (3) The correctness of translation needs to be guaranteed, which

[a] Seconded from Systerel, Toulouse, France. ning.ge@systerel.fr
[b] Seconded from Thales Avionics, Toulouse, France. eric.jenn@fr.thalesgroup.com

implies that the translation must preserve the safety properties expressed in the Event-B model. (4) Event-B does not come with constructs close to programming languages, as the B method [2] does with B0. Even though such constructs may be "emulated" by Event-B, the most refined event-B model usually only specifies contracts for software functions that need to be developed or provided outside Event-B (such as the implementation of the set theory). This leads to a verification gap between the Event-B contracts and the external or third-party implementation. (5) At the time of writing, it is still difficult to verify the deadlock-freeness and liveness properties using Rodin due to the difficulties of integrating loop variant reasoning in the Event-B model. These properties may be easier to verify using an intermediate verification language before the code is implemented. Among the above challenges, (1) and (2) have been addressed by all the existing works, while (3), (4) and (5) are still open. In this work, our focus is placed on the three open issues. A detailed discussion about the related works is provided in Section 2.

Unlike other Event-B to code translation approaches, ours relies on an intermediate verification language, namely HLL (High Level Language). HLL is a synchronous declarative language similar to Lustre [23]. It is the modeling language used by the formal verification toolset S3 (Systerel Smart Solver)[3][13], built around a SAT-based model checker (S3-core). More details about the HLL language and the S3 toolset are presented in Section 3.2.

The overall translation process from Event-B to C via HLL is depicted in Figure 1. On this figure, activities performed by existing tools are tagged by a letter while our contributions are tagged by a number. Activities are briefly explained hereafter.

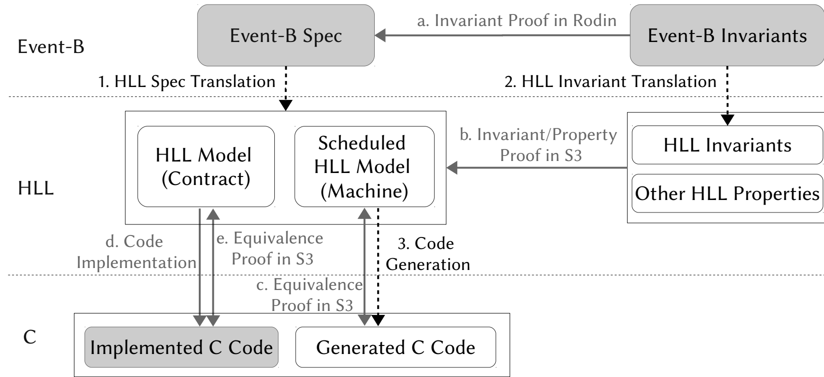

Fig. 1: Process and Activities from Event-B to C via HLL

– (a) When the Event-B refinement process is complete, all invariants in the model are proven in Rodin.
– (1) The Event-B model is translated to HLL models that contain both a scheduled machine and a set of function con tracts. Thanks to the function contracts, all in-

[3] S3 is maintained, developed and distributed by Systerel (http://www.systerel.fr/).

variants and additional properties are verifiable at the HLL level. This contribution will be detailed in Section 4.

- (2) The invariants in Event-B are translated to HLL invariants. Some properties that are not verified in the Event-B model (e.g., deadlock-freeness and liveness ones) are expressed in HLL. This contribution will be detailed in Section 4.
- (b) Compliance of the scheduled HLL model with the invariants and properties is proved using S3. If all invariants and properties are proven, the correctness of the HLL model is guaranteed. This activity will be detailed in Section 4.
- (3) + (c) The C code of most system functions is generated from the scheduled HLL model. An equivalence proof is performed between the HLL model and the code to guarantee the correctness of the generated code. This contribution will be detailed in Section 5.
- (d) + (e) Some functions of the C code are manually implemented from the function contracts in HLL. An equivalence proof is performed between the HLL contracts and the code to prove the correctness of the manually implemented code. This activity will be detailed in Section 5.

The organization of the paper is as follows: Section 2 discusses the related works; Section 3 describes the technological background, including the Event-B modeling language, the HLL modeling language and the S3 toolset, and presents our running example; Section 4 illustrates the translation from the Event-B model to the HLL model; Section 5 exposes the translation from the HLL model to the C code; Section 6 gives the experimental results on a significant use case: an automatic protection system for a robot; Section 7 gives some concluding remarks and discusses perspectives.

2 Related Works

As identified in Section 1, there are five main challenges for the translation from the Event-B model to the code. The issues of model restriction and prevention of non-determinism by scheduling have been solved in the existing works [30,9,26,20,25,10,21]. The correctness of the translation has been addressed in [30,9,26,10]. The author of [26] intended to verify the generated and implemented code using meta-proof and software model checking tools such as BLAST [4], which can check temporal safety properties of C program. As the approach was not yet experimented in their work and the details on the meta-proof and the checking of properties were not provided, it is difficult to evaluate the limits of this proposal. The work [25] generates verified C# code in a static program verification environment, namely the Spec# programming system [3], that is based on deductive verification of function contracts. For C code, the same idea can be applied using the Frama-C framework [16]. However, the verification based on function contracts is restricted to local properties. It is not easy to express and verify the safety properties concerning the global behavior of a system, as well as the deadlock-freeness and liveness properties. The work [21] generates correct C code relying on reasoning about well-definedness, assertions and refinement. It relies on a set of well-definedness restriction rules for the Event-B model to prevent the occurrence of runtime errors such as arithmetic overflows. The correctness of the generated C code is informally guaranteed by using a refinement step. To verify that the control flow is the same as the

restricted Event-B model, this work adds new variables that represents the program counter in the scheduled model and adds events that simulate the update of the counter. This approach is restricted to the verification of control flow invariants. In addition, it does not yet cover deadlock-freeness and liveness properties.

3 Technological Background

3.1 The Event-B Modeling Language and the Rodin Platform

Event-B is a modeling formalism and method for the development of systems relying on a correct-by-construction approach. At the time of writing, Event-B focuses on discrete transition systems. Centered around the notion of *events*, it is structured around *contexts* and *machines*. A *context* defines the type of data and models the static properties of the system including the function contracts. A *machine* models the dynamic behavior by means of variables, the values of which are initially determined by the *initialization* event and changed by actions of *events*. Construction *invariants* and safety properties are expressed in the machines. An *event* waits for a set of guard conditions to be verified to trigger a sequence of *actions*. A *parameter* is constrained by appropriate *guards* and its chosen value is used to update the machine *variables*. A *witness* maps a concrete parameter with a more abstract one defined in a less refined machine. Details about the Event-B language and method can be found in [1].

3.2 The HLL Modeling Language and the S3 Toolset

HLL is a synchronous dataflow language used to model a system, its environmental constraints as well as its properties. To give an overview of the language constructs, Figure 2 shows the HLL model of a saturated counter and its property about the range of output value (respectively in the *namespaces* `Counter` and `Counter_Verif`). The

```
Namespaces:
Counter {
Types:          enum {INC, DEC, RESET} Command;
Inputs:         Command in;
Constants:      int C_MIN := 0; int C_MAX := 100;
Declarations:   int unsigned 32 cnt;
Outputs:        cnt;
Declarations:   int unsigned 32 dec_input;
Blocks:         Fun_dec(int pre_v) -> (int v){v:= dec_input;}
Constraints:
  ALL f: Fun_dec (f.pre_v = C_MIN -> d.v = C_MIN);
  ALL f: Fun_dec (f.pre_v > C_MIN -> d.v = f.pre_v-1);
Definitions:
  I(cnt) := 0;
  X(cnt) := if    in==RESET then C_MIN
            elif  in==DEC   then Fun_dec(cnt)
            elif  in==INC   then
               if cnt==C_MAX then C_MAX else cnt + 1
            else  cnt;
}
```

```
Namespaces:
Counter_Verif
{
Proof Obligations:
  Counter::cnt>=Counter::C_MIN;
  Counter::cnt<=Counter::C_MAX;
}
```

Fig. 2: An Example of HLL Model

counter reacts to the input command (modelled as an HLL enumeration): incrementation (`INC`), decrementation (`DEC`) or reset (`RESET`). The saturation range is defined by

HLL *constants*. The behavior of the counter is initialized by $\mathbf{I}(\texttt{cnt})$ and periodically updated by $\mathbf{X}(\texttt{cnt})$. The effect of INC and RESET are directly defined in the schedule, while the effect of DEC is defined as a function contract by using HLL *constraints* and an intermediate variable dec_input of the HLL *block* Fun_dec(), without any concrete implementation.

HLL is used as the modeling/verification language of the S3 toolset. Design models specified in SCADE [12]/Lustre [23] or code in C/Ada can be translated to HLL thanks to translators provided by the toolset. The proof engine of S3 is a SAT-based [7] model checker, S3-core, that implements Bounded Model Checking (BMC) [6] and k-induction [29,8] techniques. The input of S3-core is a bit-level Low Level Language (LLL) that only contains boolean streams and is restricted to three bitwise operators: negation, implication and equivalence. The toolset provides "expanders" to translate HLL models into LLL models. More details about HLL and S3 can be found in [22].

S3 supports different activities of a software development process: property proof, equivalence proof, automatic test case generation, simulation, and provides necessary elements to comply with the software certification processes. It has been used for the formal verification of railway signalling systems for years by various industrial companies in this field.

3.3 A Running Example: A Traffic Light Controller

To make this paper more readable and understandable, we use the case of the traffic light controller proposed in the Rodin User's Handbook[4] as a running example. In order to illustrate the role of function contracts, we have slightly modified the original Event-B model by abstracting an event action into a set of function interface and the contracts of the function. The requirements of this traffic light controller are the follows:

- REQ-1: A traffic light controller shall be used to control both the pedestrian traffic lights and car traffic lights of a pedestrian crossing.
- REQ-2: Both traffic lights shall not be *green* at the same time, to ensure that the pedestrian crossing is in a safe state.

This example covers most Event-B constructs. The model is provided in the Appendix A. It is composed of an abstract machine (mac0) without context and a refined machine (mac1) with a context(ctx1). The specified invariants are related to the data type, the usage of gluing variables, and the safety property (REQ-2). The contract of the function that sets the pedestrian light to red color is specified in the context ctx1.

4 Translating Event-B to Verified HLL

4.1 Architecture of the HLL Model

Figure 3 depicts the static architecture of the HLL model. A complete model is composed of three main parts: (i) the specification of the scheduled system (the white

[4] https://www3.hhu.de/stups/handbook/rodin/current/html/

blocks), (ii) the specification of the invariants/properties and other verification related artefacts (the dark grey blocks), and (iii) the function contracts (the light grey blocks). Code generation requires part (i); the implementation of contracted functions requires part (iii); and the verification of the HLL model requires all parts. The HLL model of the system consists of the *inputs* of system *commands*, the *outputs* of the Event-B *variables* of the machine, the function *contracts* for the parameters, external functions and third-party library (e.g., the implementation of the set theory and HLL quantifiers). The verification architecture consists of the invariants/properties, the gluing invariants/witnesses in the less refined machine, and the related gluing variables/parameters in the abstract events.

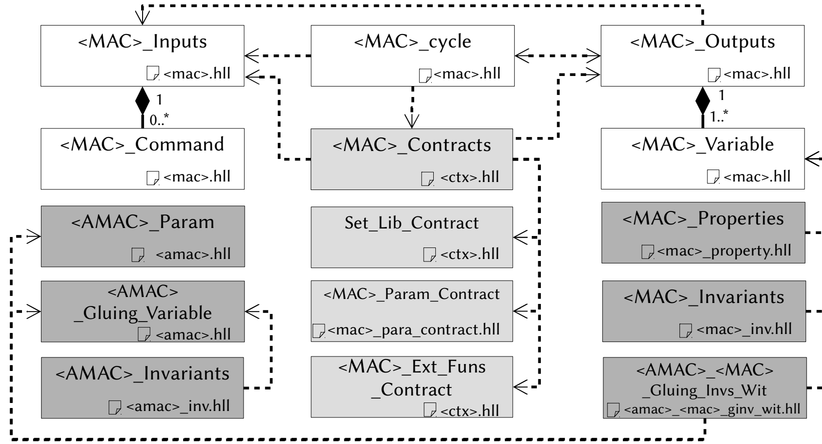

Fig. 3: Static Architecture of the HLL Model

Concerning the dynamic architecture of the model, the main question concerns the scheduling of events. When multiple events are enabled, then an event is chosen non-deterministically to be executed. However, to implement such model with a sequential programming language, and to ensure the deterministic behavior of the software, a specific scheduling of events must be chosen. The problem of event scheduling has been addressed in many existing works. We are not aimed to enumerate the translation of every possible scheduling algorithms. As an example, we use a simple algorithm where events are processed in their definition order in the Event-B model.

4.2 Translation Rules from Event-B to HLL

We present the translation rules according to the presentation of the Event-B constructs in [28], including arithmetic operations, predicates, sets, and relations.

Translating Arithmetic and Predicates The translation rules of arithmetic and predicates are defined in Table 1. These translations are straightforward, thanks to the quan-

tifiers in HLL language. Note that the quantifiers are executable at the HLL level, but they need to be implemented at the code level.

Table 1: Event-B Arithmetic and Predicates to HLL Model

Arithmetic	Event-B	HLL
Integers	INT	int signed N
Natural Number	NAT	int unsigned N
Interval	$x = n..m$	$\mathtt{int}[n, m]\ x;$
Addition	$x := a + b$	$x := a + b;$
Subtraction	$x := a - b$	$x := a - b;$
Multiplication	$x := a * b$	$x := a * b;$
Division	$x := a \div b$	$x := a\ /\ b;$
Modulo	$x := a \bmod b$	$x := a\ \%\ b;$
Exponentiation	$x := a\hat{}b$	$x := a\hat{}b;$
Minimum	$\min(S)$	$\$\min i : [0, N-1](S(i));$
Maximum	$\max(S)$	$\$\max i : [0, N-1](S(i));$

Predicate	Event-B	HLL
Negation	$\neg x$	$\sim x$
True	$\top$	true
False	$\bot$	false
Equality	$a = b$	$a == b$
Inequality	$a \neq b$	$a\ != b$
Less	$a < b$	$a < b$
Less or equal	$a \leq b$	$a \leq b$
Greater	$a > b$	$a > b$
Greater or equal	$a \geq b$	$a \geq b$
Conjunction	$x \wedge y$	$x\ \&\ y$
Disjunction	$x \vee y$	$x\ \#\ y$
Implication	$x \Rightarrow y$	$x \rightarrow y$
Equivalence	$x \Leftrightarrow y$	$x \leftrightarrow y$
Universal quantification	$\forall$	ALL
Existential quantification	$\exists$	SOME

Translating Set Theory The translation rules of sets and set predicates are defined in Table 2 and Table 3. They are implemented using the characteristic functions in HLL. In the HLL model, a set S of finite size N with elements $e_i (i = 0..N-1)$ is implemented as a boolean array A of size N such that $S[i]$ is true if and only if element e_i is in S. This implementation has strong restrictions. In particular, it imposes that the cardinality of S is bounded (and "reasonnable"). Many other implementations would have been possible, nevertheless, we chose this one to leverage the optimized operations on boolean vectors implemented in S3. Note that the code implementation of the HLL set constraints need to be provided by a set theory library.

Translating Relations and Functions A relation is a set of ordered pairs. The translation rules of relations are defined in Table 4. As a set in Event-B is implemented as a boolean array in HLL, a relation r of two sets S of size N and T of size M is implemented as a 2-dimensional boolean array $R[N][M]$. The HLL pre-conditions of a relation r between elements s_i of S and t_j of T is (i) s_i are in S ($S[i]$ is true), (ii) t_j is in T ($T[j]$ is true), (iii) r exists between s_i and t_j ($R[i][j]$ is true).

In Event-B, a special case of relations are functions, with the restriction that each element of the domain is related to a unique element in the range. The translation rules of functions are defined in Table 5.

Table 2: Event-B Sets to HLL Model

Sets	**Event-B**	**HLL**
Set	$S := \{e_1, ..., e_N\}$	bool S[N];
Empty set	$S := \varnothing$	S[i] := false;
Set comprehension	$E := \{z \cdot P \mid F\}$	E[z] := F[z] & P(z);
Union	$U := S \cup T$	U[i] := S[i] # T[i];
Intersection	$U := S \cap T$	U[i] := S[i] & T[i];
Difference	$U := S \setminus T$	U[i] := S[i] & ~T[i];
Cartesian product	$E := S \times T$	E[i][j] := S[i] & T[j];
Powerset	$\mathbb{P}(S)$	Declarations : bool P[N]; Constraints : ALL i : [0, N−1] (~S[i] → ~P[i]);
Non-empty subsets	$\mathbb{P}_1(S)$	Declarations : bool P[N]; Constraints : ALL i : [0, N−1] (~S[i] → ~P[i]); SOME i : [0, N−1] (S[i] → ~P[i]);
Cardinality	card(S)	card := population(S);
Generalized union	union(S)	U[i] := SOME j : [0, N−1] (S[i][j]);
Generalized intersection	inter(S)	U[i] := ALL j : [0, N−1] (S[i][j]);
Quantified union	UNION $z \cdot P \mid S$	U[i] := SOME j : [0, N−1] (P(S[i][j]));
Quantified intersection	INTER $z \cdot P \mid S$	U[i] := ALL j : [0, N−1] (P(S[i][j]));

Table 3: Event-B Set Predicates to HLL Model

Set Predicates	**Event-B**	**HLL**
Set membership	s : S	S(s) == true;
Set non-membership	s /: S	S(s) == false;
Subset	$S \subseteq T$	ALL i : [0, N−1] (S[i] → T[i]);
Proper subset	$S \subset T$	ALL i : [0, N−1] (S[i] → T[i]) & SOME i : [0, N−1] (T[i] → ~S[i]);
Finite set	finite(S)	(All sets in HLL are finite.)
Partition	$\text{partition}(S, s_1, ..., s_n)$	$S[i] := s_1[i]$ # ... # $s_n[i]$;

Translating Event-B Constructs Table 6 presents the mapping between Event-B constructs and the HLL elements.

Part of HLL model of the running example is provided in Figure 4, including one context (ctx1), one machine (mac1), and the invariants in the machine mac1 (mac1_INV). In the HLL model of machine mac1, for the reason of space limitations, only the variable peds_colour is defined to explain the schedule. The complete HLL model of the running example is provided in the Appendix B. As explained in Section 4.1, a sequential schedule is applied in the HLL model. Firstly, the guards (i.e. GRD_set_peds_green, GRD_set_peds_red and GRD_set_cars_colours) are defined. Then each variable is initialized by the definition I(vars), such as I(peds_colour). The update of values with respect to the schedule of events is defined in the step X(var), such as X(peds_colour).

Table 4: Event-B Relations to HLL Model

Relations	**Event-B**	**HLL**
Relations	$S \leftrightarrow T$	$Declarations : bool\ r[N][M];$ $Constraints :$ $ALL\ i : [0,N-1], j : [0,M-1]\Big((\sim S[i] \# \sim T[j]) \rightarrow \sim r[i][j]\Big);$
Domain Range	$dom(r)$ $ran(r)$	$dom[i] := SOME\ j : [0,N-1]\ (r[i][j]);$ $ran[j] := SOME\ i : [0,N-1]\ (r[i][j]);$
Total relation	S ⟷ T (total)	$Declarations : bool\ r[N][M];$ $Constraints :$ $ALL\ i : [0,N-1]\ \big(S[i] \rightarrow SOME\ j : [0,M-1]\ (T[j]\ \&\ r[i][j])\big);$ $ALL\ i : [0,N-1]\ \big(\sim S[i] \rightarrow ALL\ j : [0,M-1]\ (\sim r[i][j])\big);$
Surjective Relation	S ⟷ T (surjective)	$Declarations : bool\ r[N][M];$ $Constraints :$ $ALL\ j : [0,M-1]\ \big(T[i] \rightarrow SOME\ i : [0,N-1]\ (S[i]\ \&\ r[i][j])\big);$ $ALL\ i : [0,M-1]\ \big(\sim T[j] \rightarrow ALL\ i : [0,N-1]\ (\sim r[i][j])\big);$
Total surjective relation	S ⟷ T (total surjective)	$Declarations : bool\ r[N][M];$ $Constraints :$ $ALL\ i : [0,N-1]\ \big(S[i] \rightarrow SOME\ j : [0,M-1]\ (T[j]\ \&\ r[i][j])\big);$ $ALL\ j : [0,M-1]\ \big(T[i] \rightarrow SOME\ i : [0,N-1]\ (S[i]\ \&\ r[i][j])\big);$ $ALL\ i : [0,N-1]\ \big(\sim S[i] \rightarrow ALL\ j : [0,M-1]\ (\sim r[i][j])\big);$ $ALL\ i : [0,M-1]\ \big(\sim T[j] \rightarrow ALL\ i : [0,N-1]\ (\sim r[i][j])\big);$

Table 5: Event-B Functions to HLL Model

Functions	**Event-B**	**HLL**
Partial function	$f \in S \nrightarrow T$	$ALL\ s : S, t_1, t_2 : T\ \big(t_1 = f(s)\ \&\ t_2 = f(s) \rightarrow t_1 = t_2\big);$
Total function	$f \in S \rightarrow T$	$ALL\ s : S, t_1, t_2 : T\ \big(t_1 = f(s)\ \&\ t_2 = f(s) \rightarrow t_1 = t_2\big);$ $ALL\ s : S\ \big(SOME\ t : T\ (f(s) = t)\big);$
Partial injection	$f \in S$ ⤔ T	$ALL\ s : S,\ t_1, t_2 : T\ \big(t_1 = f(s)\ \&\ t_2 = f(s) \rightarrow t_1 = t_2\big);$ $ALL\ s_1, s_2 : S,\ t : S\ \big(t = f(s_1)\ \&\ t = f(s_2)\ \rightarrow s_1 = s_2\big);$
Total injection	$f \in S \rightarrowtail T$	$ALL\ s : S,\ t_1, t_2 : T\ \big(t_1 = f(s)\ \&\ t_2 = f(s) \rightarrow t_1 = t_2\big);$ $ALL\ s_1, s_2 : S,\ t : S\ \big(t = f(s_1)\ \&\ t = f(s_2)\ \rightarrow s_1 = s_2\big);$ $ALL\ s : S\ \big(SOME\ t : T\ (f(s) = t)\big);$
Partial surjection	$f \in S$ ⤀ T	$ALL\ s : S, t_1, t_2 : T\ \big(t_1 = f(s)\ \&\ t_2 = f(s) \rightarrow t_1 = t_2\big);$ $ALL\ t : T\ \big(SOME\ s : S\ (f(s) = t)\big);$
Total surjection	$f \in S \twoheadrightarrow T$	$ALL\ s : S,\ t_1, t_2 : T\ \big(t_1 = f(s)\ \&\ t_2 = f(s) \rightarrow t_1 = t_2\big);$ $ALL\ s : S\ \big(SOME\ t : T\ (f(s) = t)\big);$ $ALL\ t : T\ \big(SOME\ s : S\ (f(s) = t)\big);$
Bijection	$f \in S$ ⤖ T	$ALL\ s : S,\ t_1, t_2 : T\ \big(t_1 = f(s)\ \&\ t_2 = f(s) \rightarrow t_1 = t_2\big);$ $ALL\ s_1, s_2 : S,\ t : S\ \big(t = f(s_1)\ \&\ t = f(s_2)\ \rightarrow s_1 = s_2\big);$ $ALL\ s : S\ \big(SOME\ t : T\ (f(s) = t)\big);$ $ALL\ t : T\ \big(SOME\ s : S\ (f(s) = t)\big);$

4.3 Proving HLL Invariants and Properties in S3

The invariants in the Event-B models are translated to *HLL Proof Obligations* (see example `Mac1_INV` in Figure 4). The translation rules are the same as that defined in the

Table 6: Event-B Constructs to HLL Model

Event-B Construct	HLL Construct	Dependence
CONTEXT	**ctx.hll**	
EXTENDS	Abstract context HLL	abs_ctx.hll
SETS	Enum, Array	
CONSTANTS	Type	
AXIOMS	Block, Constraint	
MACHINE	**mac.hll**	
REFINES	Abstract machine HLL	abs_mac.hll
SEES	Abstract context HLL	abs_ctx.hll
VARIABLES	Inputs, Struct	
INVARIANT	Types, Proof obligations	mac.hll, (abs_)mac_inv.hll
Gluing INVARIANT	Constraints	mac_absmac_ginv.hll
EVENT	**mac.hll**	
REFINES	Abstract machine HLL	abx_mac.hll
ANY	Declarations	mac_para_contract.hll
WHERE	Predicate(vars)	mac_para_contract.hll
WITH	Constraints	mac_absmac_ginv.hll
THEN	Definitions, I(vars), X(vars)	

```
Namespaces:
Ctx1
{             // Context ctx1
Types:        // Variable types in ctx1
  enum {red, yellow, green} COLOURS_ELT;
  bool COLOURS[3];// Ordered table: red, yellow, green
Types:
  Struct {    // Variables types for function contracts
    Contract_input_fun_set_red : COLOURS_ELT
  } Fun_Contract_Vars;
Declarations:// Variables of function contract
    Fun_Contract_Vars fun_vars;

Blocks:       // mac1/set_peds_red/fun_set_red()
fun_set_red() -> (COLOURS_ELT colour)
{
    colour := fun_vars.Contract_input_fun_set_red;
}
Constraints: // Function contract for set_peds_red()
  ALL f: fun_set_red
        (fun_vars.colour = red);
}
```

```
Namespaces:
Mac1_INV
{
Proof Obligations:
  // INVARIANTS mac1/inv1
  Mac1::vars.peds_colour = Ctx1::red #
  Mac1::vars.peds_colour = Ctx1::green;
}
```

```
Namespaces:
Mac1 {
Types:
  struct {    // VARIABLES in mac1
     peds_colour : Ctx1::COLOURS_ELT,
     cars_colours: Ctx1::COLOURS
  } VARS;
  Struct {    // Local PARAMETERS
     set_cars_colours_new_value_colours: Ctx1::COLOURS
  } Param;
Inputs: Param param;
Declarations:
  VARS vars; // VARIABLES in mac1
  Ctx1::COLOURS_ELT peds_colour;
  Ctx1::COLOURS     cars_colours;
  bool GRD_set_peds_green; // GUARDS in mac1
  bool GRD_set_peds_red;
  bool GRD_set_cars_colours;
Definitions:
  GRD_set_peds_green  := (~vars.cars_colours[2]);
  GRD_set_peds_red    := TRUE;
  GRD_set_cars_colours:= TRUE;
  I(peds_colour) := Ctx1::red;   // Initialisation
  X(peds_colour) :=               // Steps
    if   GRD_set_peds_green   then Ctx1::green
    elif GRD_set_peds_red     then Ctx1::fun_set_red()
    elif GRD_set_cars_colours then peds_colour
    else                           peds_colour;
  vars := {peds_colour, cars_colours};
Outputs: vars; }
```

Fig. 4: Partial HLL Models of the Running Example

previous sections. Besides the invariants of refinement, usually the other invariants in Event-B represent safety properties. The deadlock-freeness and liveness properties will be directly expressed in HLL. The liveness property is handled by using lasso [5,24]. The deadlock-freeness [31] property is expressed using the guards of events. The HLL expression of the deadlock-freeness property is provided here. Note that $\texttt{a} := \texttt{true}, \texttt{b};$ is another written form for $\texttt{I(a)} := \texttt{true};\ \ \texttt{X(a)} := \texttt{b};$.

$$\texttt{PROP_DLF} := \ \texttt{true},\ \texttt{mac} :: \texttt{GRD_E}_1\ \#\ \texttt{mac} :: \texttt{GRD_E}_2\ \#\ \ldots\ \#\ \texttt{mac} :: \texttt{GRD_E}_n;$$

Figure 5 presents the process of property proof using S3. The HLL model, combined with properties expressed in HLL as well, are expanded to a LLL model that is fed to the S3-core. If a property is falsifiable, a generated counterexample can be simulated at

the HLL level to help the debug. S3 also provides automatic analysis tools to help the search of lemmas used by the proof.

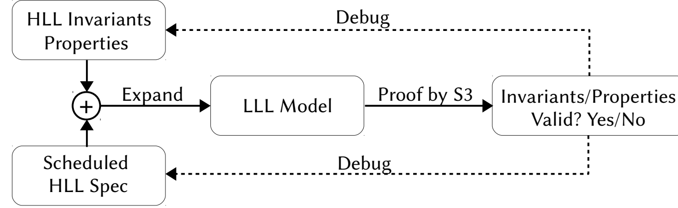

Fig. 5: Process of Property Proof using S3

5 Translating HLL to HLL-Equivalent C Code

Once the Event-B model is translated into a HLL model, the correctness of which is proven, the next phase consists of translating the HLL model into C code and proving that the code is correct. Translation of HLL to C involves two activities: an automatic one where the C code is generated directly from the HLL model, and a manual one where some functions in the C code are implemented manually from contracts expressed in HLL. In this section, we define the architecture of generated C code in Section 5.1. The code generation/implementation rules from the HLL model to the C code are respectively defined in Section 5.2 and 5.3. The approach of equivalence proof is briefly addressed in Section 5.4. Note that, instead of equivalence proof, the correctness of the code can also be guaranteed by proving the same properties (already expressed in HLL) in HLL. In order to re-prove the same properties, the C code is translated into an HLL model, where the HLL properties are combined and proved.

5.1 Architecture of the Generated C Code

The static architecture of the C code is given on Figure 6. It reflects directly the architecture of the HLL model shown in Fig. 3 for the parts related to implementation (the white blocks). The dynamic architecture is described as follows:

- The system is first initialized by the function ⟨MAC⟩`_init` that calls the function ⟨MAC⟩`_Initialisation`,
- Then the system calls periodically the scheduler ⟨MAC⟩`_Schedule`. The scheduler calls each event processing functions ⟨MAC⟩`_Events` according to the schedule order. Each event processing function evaluates its guards ⟨MAC⟩`_Guards` and, if all guards are enabled and the event is triggered, call the action realization function ⟨MAC⟩`_Actions`. The guards and actions depend on the inputs ⟨MAC⟩`_Inputs`, outputs ⟨MAC⟩`_Outputs` and the functions implemented from the contracts.

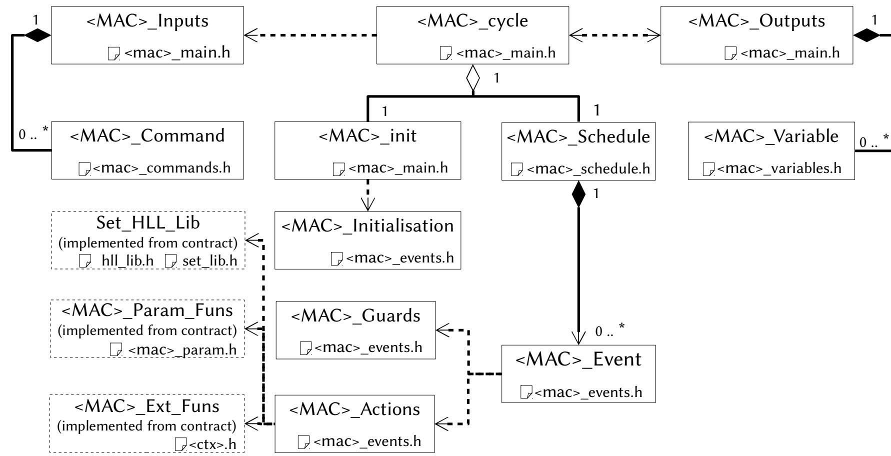

Fig. 6: Static Architecture of Generated C Code

5.2 Translation of HLL Model to C code

The C code can be generated either from the HLL model or from the LLL model that is the expanding result of the HLL model. The alternative is presented and discussed hereafter.

Code Generation from HLL Generating C code from HLL is in a way similar to that of the translating Event-B to HLL. Figure 7 shows the generated C code `mac1_main.c` and `mac1_schedule.c` of the running example. They are generated from the HLL model `Mac1` in Fig. 4. The schedule defined in the `Mac1` HLL model is extracted in the code `mac1_schedule.c`. The main file `mac_main.c` conforms to the cyclic execution in the HLL model. A set of global variables is defined as the set of system states and initialized. They are cyclically updated according to the order defined by the event scheduler. The guards and actions are defined as functions in the `mac1_events.c`. Because of space limitations, the complete C code of the running example is provided in the Appendix C.

Code Generation from LLL The C code may also be generated directly from the intermediate LLL model, which is the expanded translation of the HLL model. This is achieved by the LLL to C translator in S3. We present an example in Figure 8. In the model `ex.lll`, all variables are boolean, and only three bitwise operators are used, therefore the translation is direct. The C code `ex.c` contains a `init` function and a `cycle` function. The translation is automatic, and the cost of the conversion from LLL to C is very low. However, as all operations - including arithmetic operations - are encoded using bit-level C operators, execution performance degrades gradually as the number of arithmetic operations increases. This solution is interesting for applications that perform mainly logical operations.

```c
#include "mac1_main.h"
static int init = 1;
static Mac1_VARS global_vars;
void mac1_init( Mac1_VARS* vars ) {
    Initialisation( vars );
}
void mac1_step( Mac1_VARS* vars, Mac1_Param param ) {
    mac1_schedule( vars, param ); // Schedule of events
}
Mac1_VARS mac1_cycle( Mac1_Param params ) {
    Mac1_VARS* vars = &global_vars;
    Mac1_VARS out_vars;
    if ( init == 1 ) {      // Initialisation
       mac1_init( vars );
       init = 0;
    }
    else {                  // schedule of events
       mac1_schedule( vars, params );
    }
    out_vars.peds_colour = vars->peds_colour;
    for ( int i = 0; i < 3: i++ ) {
       out_vars.cars_colours[i] = vars->cars_colours[i];
    }
    return out_vars;
}
```
mac1_main.c

```c
#include "mac1_schedule.h"

bool mac1_schedule( Mac1_VARS* vars,
       Mac1_Local_Param param ) {

  // Event set_peds_green
  if ( GRD_set_peds_green( *vars ) ) {
    ACT_set_peds_green( vars );
    return true;
  }

  // Event set_peds_red
  if ( GRD_set_peds_red( *vars ) ) {
    ACT_set_peds_red( vars );
    return true;
  }

  // Event set_cars_colours
  if ( GRD_set_cars_colours( *vars ) ) {
    ACT_set_cars_colours( vars, param );
    return true;
  }

  return false;
}
```
mac1_schedule.c

Fig. 7: Generated C Code of the Running Example

```
__init__ := TRUE, ~TRUE;
s5 := s4 -> s3;
s6 := s4 -> ~s5;
s7 := s6 -> ~s3;
s8 := s6 -> ~s4;
s16 := s13 -> s12;
s17 := s13 -> ~s16;
```
ex.lll

```c
unsigned char s874;
void init(){
    __init__ = TRUE;
    s598 = !TRUE;
void cycle(){
    __init__ = !TRUE;
    s5 = !s4 || s3;
```
ex.c

Fig. 8: Generated C Code from the LLL Model

5.3 Code Implementation from HLL Contracts

The HLL function contracts require a developer to translate it to C. This is the usual practice, even though in our case, the software specification is formal. We show a function implemented under the HLL contract used in the running example in Figure 9. The function `mac1_fun_set_red`() is implemented from the contract expressed as constraint on the block `fun_set_red`() in the namespace `ctx1` in Fig. 4, shown as follows.

$$\texttt{ALL f} : \texttt{fun_set_red}\ (\texttt{fun_vars.Contract_input_fun_set_red} = \texttt{red});$$

```
#include "bool.h"
#ifndef CTX1_H
#define CTX1_H
enum COLOURS_ELT {red, yellow, green};
typedef bool COLOURS[3];
enum COLOURS_ELT mac1_fun_set_red();
#endif
```
ctx1.h

```
#include "ctx1.h"
enum COLOURS_ELT mac1_fun_set_red()
{
  return red;
}
```
ctx1.c

Fig. 9: Example of Implementation from HLL Contracts

5.4 Proving Equivalence between HLL Model and C Code

Proving the equivalence of two models (or one program and one model) consists in proving that the two systems behave identically (in particular, provide the same outputs) for any input in the input domain. In our case, we rely on the equivalence proof to guarantee the correctness of the code. The equivalence proof is concerned with two problems: the equivalence between HLL model and the generated C code, and the equivalence between HLL contracts and the implemented C code. Figure 10 presents the process of proving the equivalence between the HLL model and the generated/implemented C code. The C code is translated into another HLL model. Both HLL models are then respectively expanded to two LLL models using diversified expanders[5]. To prove the equivalence of two HLL models, the equivalence models are constructed and proved at the LLL level.

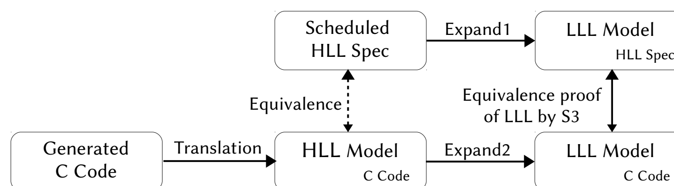

Fig. 10: Process of Equivalence Proof

The approach is sound for two reasons. First, the correctness of the HLL model of the system is proven, as shown in Section 4. Second, the verification means (i.e., the S3 toolset) is developed so as to facilitate qualification against certification standards (in particular, DO-178C [18] and DO-330 [19]). Towards this goal, S3 is organized in a set of small, independent components, from which the most critical ones - an equivalence model constructor, and a tool to verify the validity of the proof - are developed according to the highest integrity levels.

In practice, it shall be noted that some of the Event-B events actually model the modifications of monitored and commanded variables, i.e., variables that are not modified by the function under design [27,11]. The actions triggered by these events describe

[5] The diversified expanders have been designed and implemented by different teams using different programming languages.

the expected effects of the external environment on these variables. As they do not represent actions of the system, they shall not be translated to software code. Thus, they are implemented by simple interface functions performing acquisition of external variables. As a result, no equivalence checking can be done for those parts of the generated C code, their verification shall thus be achieved using other means (formal or informal).

6 The Case Study: Automatic Rover Protection

TwIRTee is the small three-wheeled robot (or "rover") used as the demonstrator of the INGEQUIP project[6]. It is used to evaluate new methods and tools in the domain of hardware/software co-design, virtual integration, and application of formal methods for the development of equipments. TwIRTee's architecture, software, and hardware components are representative of a significant family of aeronautical, spatial and automotive systems [15]. A rover performs a sequence of missions (❶ on Figure 11). A mission is defined by a start time and an ordered set of waypoints to be passed-by. Missions are planned off-line and transmitted to the rover by a supervision station (❷). To go from the first waypoint to the last, the rover moves on a track that is materialized by a grey line on the ground. In a more abstract way, a complete mission can be modelled by a path in a graph where nodes represent waypoints, and edges represent parts of the track joining two waypoints. A rover shares the tracks with several identical rovers. In order to prevent collisions, each of them embeds a protection function (or ARP) which purpose is to maintain some specified spatial (❸) and temporal separation (❹) between them. On Figure 1, temporal separations are represented by light green and light red areas superimposed on the map: basically, $rover_2$ (resp. $rover_1$) shall never enter the light green (resp. light red). In the current implementation, the ARP essentially acts by reducing the rover speed and, in some specific cases, by performing a simple avoidance trajectory. To take the appropriate action, the ARP exploit the following pieces of information: the map, the position of all other rovers transmitted by a centralized supervision station (❺), and its own position.

For this paper, we rely on a specific model of the ARP function where some elements have been simplified. We thus only consider specification elements such as rover position, speed, deceleration, and others as being discrete values (no use of Real or Floating Point data). The statistics of the Event-B model, translated HLL models, and generated/implemented C code are provided in Table 7.

7 Conclusion and Perspective

7.1 Conclusion

This work addresses the translation of Event-B into C and the demonstration of the correctness of translation using formal methods. Our approach relies on an intermediate modeling/verification language HLL. The correctness proof follows two steps.

[6] The INGEQUIP project is conducted at the Institut de Recherche Technologique of Toulouse (IRT-Saint Exupéry)

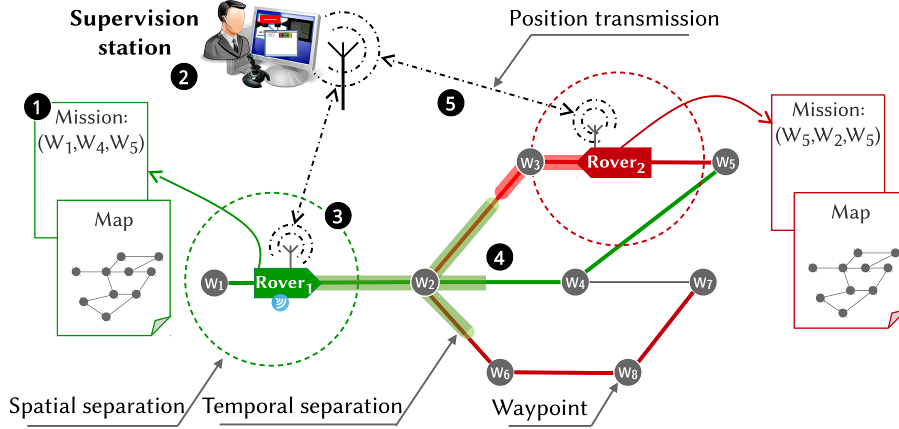

Fig. 11: ARP System Overview

Table 7: Statistics of the ARP Models (Event-B and HLL) and Code

Event-B model	Events / Actions / Guards	18 / 70 / 74
	Axioms / Theorems / Constants / Variables / Invariants	40 / 11 / 27 / 15 / 79
	PO (Total / Automatic / Manuel)	634 / 626 / 8
	LOC of Event-B model of the final refinement	243
HLL models	Properties (safety / liveness/ deadlock-freeness)	42 / 4 / 1
	Time for property proof (invariants + properties)	4s
	Contracts (external function / parameter)	5 / 12
	LOC of HLL models (system / verification)	800 / 500
C code	LOC of C code(generated / implemented)	1200 / 600
	Time for equivalence proof	3s

First, the final refinement of the Event-B model, including invariants from all refined machines and elements from all extended contexts, are translated to the HLL model. Additional properties are expressed in HLL. The invariants/properties are proved at the HLL level to guarantee the correctness of the translation. Second, the C code is automatically generated from the HLL model for most of the system functions and manually for implemented from the contracts for the remaining ones. The equivalence between the code and the HLL model is proved to guarantee the correctness of the code. In this paper, we define the translation rules, and show the experimented results on a significant use case. More details about the refinement and design can be found in [17].

Compared to the existing approaches to proving the direct translation from the Event-B model to generated code, our contribution is to address the verification of additional properties (i.e., deadlock-freeness and liveness properties) in the HLL model. For the direct translation approaches [26,25,21], the analysis of these properties needs to be performed in the final C code. Considering the complexity of analysis, the authors of these works all mentioned the limits of their approaches and considered it as an open issue.

Another advantage of our approach is the support of property proof and equivalence proof of Floating-Point Arithmetic (FPA) in both the HLL model and the code. The S3 toolset provides an HLL library of FPA based on bit-blasting[7] [13] that conforms to the FPA standard *IEEE std 2008-754* [32].

7.2 Perspectives

The development of the translation tool is ongoing. As the specification of the HLL language will be published in a near future, it will soon become possible to integrate the translator as a plug-in in the Rodin platform.

The antecedent of Event-B, the B method, supports formal refinement until the programming language B0, that is translated later to C/ADA code. This method requires more refinements, but the resulting B0 model is sufficiently close to software code that code generation becomes straightforward. The correctness of final code still needs to be proved. As the verification gap between B0 and C code is smaller than the one between Event-B and C code, the results of this work could be adapted to the translation from B0 to C via HLL.

Acknowledgments

This work has been funded by the INGEQUIP project. The authors would like to thank the members in the project, and the colleagues of the Systerel S.A.S company Nicolas Breton, Mathieu Clabaut and Yoann Fonteneau for their good cooperation. The author Ning Ge would like to thank Hongyu Liu for his help on this work.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.
2. Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.
3. Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.
4. Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.
5. Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.
6. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.
7. Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[7] Bit-blasting is a classic method that translates bit-vector formulas into propositional logic expressions.

8. Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, pages 409–426. Springer, 2000.
9. Pontus Boström. Creating sequential programs from event-b models. In *International Conference on Integrated Formal Methods*, pages 74–88. Springer, 2010.
10. Pontus Boström, Fredrik Degerlund, Kaisa Sere, and Marina Waldén. Derivation of concurrent programs by stepwise scheduling of event-b models. *Formal Aspects of Computing*, 26(2):281–303, 2014.
11. Michael Butler. Towards a cookbook for modelling and refinement of control problems. 2009.
12. Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *ACM Sigplan Notices*, volume 38, pages 153–162. ACM, 2003.
13. Mathieu Clabaut, Ning Ge, Nicolas Breton, Eric Jenn, Rémi Delmas, and Yoann Fonteneau. Industrial grade model checking - use cases, constraints, tools and applications. In *International Conference on Embedded Real Time Software and Systems*, 2016.
14. Joey Coleman, Cliff Jones, Ian Oliver, Alexander Romanovsky, and Elena Troubitsyna. Rodin (rigorous open development environment for complex systems). In *Fifth European Dependable Computing Conference: EDCC-5 supplementary volume*, pages 23–26, 2005.
15. Philippe Cuenot, Eric Jenn, Eric Faure, Nicolas Broueilh, and Emilie Rouland. An experiment on exploiting virtual platforms for the development of embedded equipments. In *International Conference on Embedded Real Time Software and Systems*, 2016.
16. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.
17. Arnaud Dieumegard, Ning GE, and Eric Jenn. An Experiment Report on a Process Combining Formal Refinement and Formal Software verification. working paper or preprint, October 2016.
18. RTCA DO. 178c. *Software considerations in airborne systems and equipment certification*, 2011.
19. RTCA DO. 330. *Software Tool Qualification Considerations*, 2011.
20. Andrew Edmunds and Michael Butler. Tasking event-b: An extension to event-b for generating concurrent code. In *PLACES 2011*, February 2011.
21. Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code generation for event-b. In *International Conference on Integrated Formal Methods*, pages 323–338. Springer, 2014.
22. Ning Ge, Eric Jenn, Nicolas Breton, and Yoann Fonteneau. Formal verification of a rover anti-collision system. In *International Workshop on Formal Methods for Industrial Critical Systems and Automated Verification of Critical Systems*, volume 9933, pages 171–188, 2016.
23. Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
24. Thai Son Hoang and Jean-Raymond Abrial. Reasoning about liveness properties in event-b. In *International Conference on Formal Engineering Methods*, pages 456–471. Springer, 2011.
25. Dominique Méry and Rosemary Monahan. Transforming event B models into verified c# implementations. In *First International Workshop on Verification and Program Transformation, VPT 2013, Saint Petersburg, Russia, July 12-13, 2013*, pages 57–73, 2013.
26. Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-b models. In *Proceedings of the second symposium on information and communication technology*, pages 179–188. ACM, 2011.

27. David Lorge Parnas and Jan Madey. *Functional Documentation for Computer Systems Engineering: Version 2*. McMaster University, Faculty of Engineering, Communications Research Laboratory, 1991.
28. Ken Robinson. A concise summary of the event b mathematical toolkit. `http://wiki.event-b.org/images/EventB-Summary.pdf`, 2009.
29. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal methods in computer-aided design*, pages 127–144. Springer, 2000.
30. Steve Wright. Automatic generation of c from event-b. In *Workshop on integration of model-based formal methods and tools*, page 14. Citeseer, 2009.
31. Faqing Yang and Jean-Pierre Jacquot. An event-b plug-in for creating deadlock-freeness theorems. In *14th Brazilian Symposium on Formal Methods*, 2011.
32. Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

Appendix A: Event-B Model of Traffic Light Control

CONTEXT ctx1
SETS
    COLOURS
CONSTANTS
    red
    yellow
    green
    fun_set_red
AXIOMS
    axm1: $partition(COLOURS, \{red\}, \{yellow\}, \{green\})$
    axm2: $fun_set_red \in COLOURS$
    axm3: $fun_set_red = red$
END

MACHINE mac0
VARIABLES
    cars_go
    peds_go
INVARIANTS
    inv1: $cars_go \in BOOL$
    inv2: $peds_go \in BOOL$
    inv3: $\neg(cars_go = TRUE \land peds_go = TRUE)$
EVENTS
Initialisation
    begin
        act1: $cars_go := FALSE$
        act2: $peds_go := FALSE$
    end
Event set_peds_go ⟨ordinary⟩ $\widehat{=}$
    when
        grd1: $cars_go = FALSE$
    then
        act1: $peds_go := TRUE$
    end
Event set_peds_stop ⟨ordinary⟩ $\widehat{=}$
    begin
        act1: $peds_go := FALSE$
    end
Event set_cars ⟨ordinary⟩ $\widehat{=}$
    any
        new_value
    where
        grd1: $new_value \in BOOL$
        grd2: $new_value = TRUE \Rightarrow peds_go = FALSE$
    then
        act1: $cars_go := new_value$
    end
END

MACHINE mac1
REFINES mac0
SEES ctx1
VARIABLES

peds_colour
cars_colours

INVARIANTS

inv4: $peds_colour \in \{red, green\}$
inv5: $cars_colours \subseteq COLOURS$
gluing_peds: $peds_go = TRUE \Leftrightarrow peds_colour = green$
gluing_cars: $cars_go = TRUE \Leftrightarrow green \in cars_colours$

EVENTS

Initialisation

begin
act1: $peds_colour := red$
act2: $cars_colours := \{red\}$
end

Event set_peds_green ⟨ordinary⟩ $\widehat{=}$
refines set_peds_go
when
grd1: $green \notin cars_colours$
then
act1: $peds_colour := green$
end

Event set_peds_red ⟨ordinary⟩ $\widehat{=}$
refines set_peds_stop
begin
act1: $peds_colour := fun_set_red$
end

Event set_cars_colours ⟨ordinary⟩ $\widehat{=}$
refines set_cars
any
new_value_colours
where
grd1: $new_value_colours \subseteq COLOURS$
grd2: $green \in new_value_colours \Rightarrow peds_colour = red$
grd3: $cars_colours = \{yellow\} \Rightarrow new_value_colours = \{red\}$
grd4: $cars_colours = \{red\} \Rightarrow new_value_colours = \{red, yellow\}$
grd5: $cars_colours = \{red, yellow\} \Rightarrow new_value_colours = \{green\}$
grd6: $cars_colours = \{green\} \Rightarrow new_value_colours = \{yellow\}$
with
new_value: $new_value = TRUE \Leftrightarrow green \in new_value_colours$
then
act1: $cars_colours := new_value_colours$
end

END

Appendix B: HLL Models of Traffic Light Control

```
Namespaces:
Ctx1 {
Types:
  enum {red, yellow, green} COLOURS_ELT;
  bool COLOURS[3];  /* Ordered array: 0-red, 1-yellow, 2-green */
Types:
  struct {        /* Variables for specifying function contracts */
    Contract_input_fun_set_red : COLOURS_ELT
  } Fun_Contract_Vars;
Declarations:        /* Variables of function contract */
  Fun_Contract_Vars fun_vars;
Blocks:              /* mac1/set_peds_red/fun_set_red() */
fun_set_red() -> (COLOURS_ELT colour) {
    colour := fun_vars.Contract_input_fun_set_red;
}
Constraints:         /* Contract of function set_peds_red() */
  ALL f: fun_set_red (f.colour = red);
}
```

```
Mac0 {
Types:
  struct {  /* PARAMETERS */
    set_cars_new_value : bool
  } Param;
  struct {  /* Gluing VARIABLES in mac0 */
    cars_go : bool,
    peds_go : bool
  } VARS;
Inputs:     /* VARIABLES in mac0 */
  VARS vars;
  Param param;
}
```

```
Mac0_INV {
Proof Obligations:
  /* INVARIANTS in mac0: mac0/inv3  */
  ~(Mac0::vars.peds_go & Mac0::vars.cars_go);
}
```

```
Mac1_Mac0_Gluing_INV {
Constraints:
  /* Gluing invariants */
  Mac0::vars.peds_go <-> Mac1::peds_colour = Ctx1::green;
  Mac0::vars.cars_go <-> Mac1::cars_colours[2];

  /* Witness in mac1/set_cars_colours*/
  Mac0::param.set_cars_new_value <-> Mac1::param.set_cars_colours_new_value_colours[2];
}
```

```
Mac1 {
Types:
  struct {        /* VARIABLES in mac1 */
    peds_colour : Ctx1::COLOURS_ELT,
    cars_colours: Ctx1::COLOURS
  } VARS;
  struct {        /* PARAMETERS */
    set_cars_colours_new_value_colours: Ctx1::COLOURS
  } Param;
Inputs:
  Param param;     /* Parameters */
Declarations:
  /* VARIABLES in mac1 */
  VARS vars;
  Ctx1::COLOURS_ELT peds_colour;
  Ctx1::COLOURS      cars_colours;
```

```
  /* GUARDS in mac1 */
  bool GRD_set_peds_green;
  bool GRD_set_peds_red;
  bool GRD_set_cars_colours;
Definitions:
  /* Define guards of events */
  GRD_set_peds_green   := (~vars.cars_colours[2]);
  GRD_set_peds_red     := TRUE;
  GRD_set_cars_colours:= TRUE;

  /* Define actions of events */
  /* Initialisation */
  I(peds_colour) := Ctx1::red;
  I(cars_colours):= {TRUE, FALSE, FALSE};

  /* Schedule */
  X(peds_colour) := if   GRD_set_peds_green   then Ctx1::green
                    elif GRD_set_peds_red     then Ctx1::fun_set_red()
                    elif GRD_set_cars_colours then peds_colour
                    else                           peds_colour;
  X(cars_colours):= if   GRD_set_peds_green   then cars_colours
                    elif GRD_set_peds_red     then cars_colours
                    elif GRD_set_cars_colours then param.set_cars_colours_new_value_colours
                    else                           cars_colours;
  vars := {peds_colour, cars_colours};
Outputs: vars;
}
```

```
Mac1_INV {
Proof Obligations:
 /* INVARIANTS in mac1 : mac1/inv1      */
 Mac1::vars.peds_colour = Ctx1::red # Mac1::vars.peds_colour = Ctx1::green;
}

Mac1_Para_Contract {
Constraints: /* Local parameter contracts */
 /* Event set_cars_colours: grd2 */
 Mac1::param.set_cars_colours_new_value_colours[2] -> Mac1::peds_colour = Ctx1::red;
 /* Event set_cars_colours: grd3 */
 (~Mac1::cars_colours[0] & Mac1::cars_colours[1] & ~Mac1::cars_colours[2])
    -> (    Mac1::param.set_cars_colours_new_value_colours[0]
        & ~Mac1::param.set_cars_colours_new_value_colours[1]
        & ~Mac1::param.set_cars_colours_new_value_colours[2]);
 /* Event set_cars_colours: grd4 */
 (Mac1::cars_colours[0] & ~Mac1::cars_colours[1] & ~Mac1::cars_colours[2])
    -> (   Mac1::param.set_cars_colours_new_value_colours[0]
        & Mac1::param.set_cars_colours_new_value_colours[1]
        & ~Mac1::param.set_cars_colours_new_value_colours[2]);
 /* Event set_cars_colours: grd5 */
 (Mac1::cars_colours[0] & Mac1::cars_colours[1] & ~Mac1::cars_colours[2])
    -> (   ~Mac1::param.set_cars_colours_new_value_colours[0]
        & ~Mac1::param.set_cars_colours_new_value_colours[1]
        & Mac1::param.set_cars_colours_new_value_colours[2]);
 /* Event set_cars_colours: grd6 */
 (~Mac1::cars_colours[0] & ~Mac1::cars_colours[1] & Mac1::cars_colours[2])
    -> ( ~Mac1::param.set_cars_colours_new_value_colours[0]
        & Mac1::param.set_cars_colours_new_value_colours[1]
        & ~Mac1::param.set_cars_colours_new_value_colours[2]);
}
```

```
Mac1_PROPS {
Declarations:       /* Property of deadlock freeness */
  bool PROP_DLF;
Definitions:        /* Compute property of deadlock freeness */
  PROP_DLF := TRUE,   Mac1::GRD_set_peds_green
                    # Mac1::GRD_set_peds_red
                    # Mac1::GRD_set_cars_colours;
Proof Obligations:  /* Property of deadlock freeness */
  PROP_DLF;
}
```

Appendix C: C Code of Traffic Light Control

ctx1.h

```
#include "bool.h"

#ifndef CTX1_H
#define CTX1_H
enum COLOURS_ELT {
    red,
    yellow,
    green
};
typedef bool COLOURS[3];
enum COLOURS_ELT mac1_fun_set_red();
#endif
```

---

ctx1.c

```
#include "ctx1.h"
enum COLOURS_ELT mac1_fun_set_red()
{
        return red;
}
```

---

mac1_events.h

```
#include "ctx1.h"
#include "mac1_param.h"
#include "mac1_variables.h"

#ifndef MAC1_EVENTS_H
#define MAC1_EVENTS_H

/* Guards in mac1 */
bool GRD_set_peds_green( Mac1_VARS );
bool GRD_set_peds_red( Mac1_VARS );
bool GRD_set_cars_colours( Mac1_VARS );

/* Actions in mac1 */
void Initialisation( Mac1_VARS* );
void ACT_set_peds_green( Mac1_VARS* );
void ACT_set_peds_red( Mac1_VARS* );
void ACT_set_cars_colours( Mac1_VARS* ,Mac1_Param );
#endif
```

---

mac1_events.c

```
#include "mac1_events.h"
/* Guards in mac1 */
bool GRD_set_peds_green( Mac1_VARS vars ) {
    return ( vars.cars_colours[2] == false );
}

bool GRD_set_peds_red( Mac1_VARS vars ) {
    return true;
}

bool GRD_set_cars_colours( Mac1_VARS vars ) {
    return true;
}

/* Actions in mac1 */
void Initialisation( Mac1_VARS* vars ) {
    vars->peds_colour = red;
    vars->cars_colours[0] = true;
    vars->cars_colours[1] = false;
    vars->cars_colours[2] = false;
}

void ACT_set_peds_green( Mac1_VARS* vars ) {
    Mac1_VARS pre_vars = *vars;
    vars->peds_colour =  green;
```

```
    vars->cars_colours[0] = pre_vars.cars_colours[0];
    vars->cars_colours[1] = pre_vars.cars_colours[1];
    vars->cars_colours[2] = pre_vars.cars_colours[2];
}

void ACT_set_peds_red( Mac1_VARS* vars ) {
    Mac1_VARS pre_vars = *vars;
    vars->peds_colour = mac1_fun_set_red();
    vars->cars_colours[0] = pre_vars.cars_colours[0];
    vars->cars_colours[1] = pre_vars.cars_colours[1];
    vars->cars_colours[2] = pre_vars.cars_colours[2];
}

void ACT_set_cars_colours( Mac1_VARS* vars, Mac1_Param param ) {
    Mac1_VARS pre_vars = *vars;
    vars->peds_colour = pre_vars.peds_colour;
    vars->cars_colours[0] = param.set_cars_colours_new_value_colours[0];
    vars->cars_colours[1] = param.set_cars_colours_new_value_colours[1];
    vars->cars_colours[2] = param.set_cars_colours_new_value_colours[2];
}
```

---

mac1_param.h

```
#include "ctx1.h"

#ifndef MAC1_PARAM_H
#define MAC1_PARAM_H
typedef struct {
    COLOURS set_cars_colours_new_value_colours;
} Mac1_Param;
#endif
```

---

mac1_variables.h

```
#include "ctx1.h"

#ifndef MAC1_VARIABLES_H
#define MAC1_VARIABLES_H
/* VARIABLES in mac1 */
typedef struct {
    enum COLOURS_ELT peds_colour;
    COLOURS cars_colours;
} Mac1_VARS;
#endif
```

---

mac1_schedule.h

```
#include "mac1_events.h"
#include "mac1_param.h"
#include "mac1_variables.h"

#ifndef MAC1_SCHEDULE_H
#define MAC1_SCHEDULE_H
bool mac1_schedule( Mac1_VARS*, Mac1_Param );
#endif
```

---

mac1_schedule.c

```
#include "mac1_schedule.h"
bool mac1_schedule( Mac1_VARS* vars, Mac1_Param param ) {
        // Event set_peds_green
        if ( GRD_set_peds_green( *vars ) )
        {
                ACT_set_peds_green( vars );
                return true;
        }

        // Event set_peds_red
        if ( GRD_set_peds_red( *vars ) )
        {
                ACT_set_peds_red( vars );
                return true;
```

```
        }

        // Event set_cars_colours
        if ( GRD_set_cars_colours( *vars ) )
        {
                ACT_set_cars_colours( vars, param );
                return true;
        }

        return false;
}
```

---

mac1_main.h

```
#include "ctx1.h"
#include "mac1_param.h"
#include "mac1_variables.h"
#include "mac1_events.h"
#include "mac1_schedule.h"

#ifndef MAC1_MAIN_H
#define MAC1_MAIN_H
typedef struct {
    Mac1_Param mac1_param;
} Mac1_Inputs;

typedef struct {
    Mac1_VARS mac1_vars;
} Mac1_Outputs;

void mac1_init( Mac1_VARS* );
void mac1_step( Mac1_VARS*, Mac1_Param );
Mac1_VARS mac1_cycle( Mac1_Param );
#endif
```

---

mac1_main.c

```
#include "mac1_main.h"
static int init = 1;
static Mac1_VARS global_vars;

void mac1_init( Mac1_VARS* vars ) {
    Initialisation( vars );
}

void mac1_step( Mac1_VARS* vars, Mac1_Param param ) {
    mac1_schedule( vars, param );
}

Mac1_VARS mac1_cycle( Mac1_Param params ) {
    Mac1_VARS* vars = &global_vars;
    Mac1_VARS out_vars;

    if ( init == 1 ) {
            mac1_init( vars );
            init = 0;
    }
    else {
            mac1_step( vars, params );
        }

    out_vars.peds_colour = vars->peds_colour;
    for ( int i = 0; i < 3: i++ ) {
            out_vars.cars_colours[i] = vars->cars_colours[i];
    }

    return out_vars;
}
```