Integration of polychrony and QGen model compiler

Christophe Junke, Thierry Gautier, Jean-Pierre Talpin, Loic Besnard

Integration of Polychrony and QGen Model Compiler

Christophe Junke, Thierry Gautier, Jean-Pierre Talpin
INRIA, Rennes-Bretagne-Atlantique Research Centre, France
firstname.lastname@inria.fr

Loïc Besnard
CNRS, IRISA, Rennes, France
loic.besnard@irisa.fr

Abstract **We present the development of a model transformation tool between the synchronous Signal language and QGen model compiler's intermediate language as well as an alternative block sequencing implementation for QGen which supports strict partial and total orders. We discuss our contributions and their possible applications in the field of reactive system design as well as some experiments**

1 Introduction

The work presented in this paper is about providing a semantic bridge between the data-flow language Signal and a model compiler named QGen developed by Adacore. QGen is the result of a collaborative work made in a project funded by the French FUI funding framework[1], named P, which was initiated to continue the work made in the Geneauto project [15]. Signal [3, 1] is a synchronous data-flow language for designing reactive systems. The language manipulates clocks and dependencies to perform a static scheduling of computations and data exchanges. Our objective in the P project was to use the Polychrony toolset of the Signal language to compute fined-grained static scheduling of computations and communications for P models based on architectural properties, as demonstrated in a previous work [19]. In this paper, we show a translation scheme between the formalism used by QGen, named P, and the Signal metamodel, SSME[2].

1.1 P project and QGen

The goal of the P project was to develop a qualifiable model compiler in the context of critical systems, which would group several existing heterogenous modelling formalisms under a single language, referred to as P language. The set of input languages envisioned for P was ranging from architecture description languages, such as AADL [11], SysML [13], Marte [21], to control and command languages such as Matlab Simulink [8] and SCADE [10]. The motivation behind the P project and in particular its goals regarding qualification is explained in details in [4, 5].

QGen is the compiler resulting from this combined effort and its first version was released February, 2015. It can import a subset of discrete Simulink models (models with a fixed time-step) and produce MISRA C [20] or Ada Ravenscar [6] code along with traceability information and formal annotations. The P language is layered as multiple languages describing different views of a same system. (i) *Code Model* is a model representation of imperative statements and functions. This language is an intermediate target before code optimizations and the actual code generation to C and Ada. (ii) *System Model* is inspired by Simulink and other block diagram formalisms, where systems are organized as data-flow graphs, connecting blocks by their data and control ports through signals. (iii) *Block library Model* is a configuration language describing block types. Block types are identified by names (e.g. "Gain", "Product") and can contain additional parameters. For example, the Sum block has different meaning depending on the number and type of its arguments (scalars, vectors, matrices). The QGen compiler is responsible for translating block instances as Code Model elements.

Typically, a model is first designed as a System Model and is gradually refined upto a point where imperative Code Model elements are generated and optimized. The QGen compiler is organized as multiple optional steps of transformations. The first one is a preprocessing step which removes organizational and visual features: for example, from/goto blocks, used for simplifying data-flow routing in a model, can be replaced by actual signals; also, artificial boundaries defined by virtual subsystems are removed, so that blocks previously grouped in a virtual system are moved as direct elements of the removed system's parent. Another interesting step is block sequencing. As a hierarchical data-flow graph, the System Model defines a partial order among blocks, which must be respected at execution time. The purpose of the block sequencer is to compute a sequential execution order for all blocks in a system according to data-flow dependencies and other attributes. Finally, a compilation step transforms System Model elements into Code Model elements. Without getting into details, each system is implemented as a function manipulating global variables, whereas blocks of a system are run in accordance to the total execution order computed at the previous step. Nested subsystems are implemented as function calls in the enclosing system's associated body. At any stage of the processing, the current model can be exported as XMI. Various options modify the behavior of the whole compilation process.

1.2 Signal and Polychrony

Signal is a synchronous data-flow language for designing reactive systems, based on infinite discrete flows of values and events called *signals*. A signal x represents an unbounded series $(x_t)_{t\in\mathbb{N}}$ of values at each logical instant t. At any instant, a signal may be present, at which point it holds a value, or absent, which is denoted by $\bot$. Signal values can be constructed from previously known values using a delay operator denoted $\$$ which shifts values of a signal by a fixed number of steps.

For each signal s is defined a clock $\hat{s}$ which can be rep-

[1] Fonds Unique Interministériel, http://competitivite.gouv.fr/
[2] Signal's Syntactic Model under Eclipse

resented as a boolean variable in the domain of clocks. A clock $\hat{s}$ denotes the set of logical instants at which the signal s has a value. The null clock $\hat{0}$ is always absent. Signal programs are described as *processes* (written here P or Q) which define relations between clocks and values of *signals* through signal equations. $(P \mid Q)$ denotes the synchronous composition of processes P and Q where equations represented by both P and Q are simultaneously considered: it corresponds to a union of constraints. The process P/x restricts the lexical scope of signal x to the process P.

Signal provides polychronous operators that are interpreted as clock constraints on signals, such as *when* or *default*. For example, *a when b* is an expression over signals whose clock is $\hat{a} \wedge \hat{b}$: this boolean formula means that the signal *a when b* is present exactly when both signals a and b are present. Similarly, *a default b* has the clock $\hat{a} \vee \hat{b}$, which means that, if during the execution of system being designed, either signal a or b is present, the signal *a default b* is also present. Signals having equivalent clocks are said to be synchronous and can be combined with monochronous operators, which include numerical operators (e.g. $a + b$) and spatial operators (e.g. array access).

A clock $\hat{d}$ is said to be a child clock of $\hat{c}$ if $\hat{d} \rightarrow \hat{c}$: whenever $\hat{d}$ is present, its parent clock $\hat{c}$ is also necessarily present. The signal compiler organizes clocks into a hierarchy during a static analysis called *clock calculus*. The compiler ensures that a signal is computed only when its clock can be determined to be true: it is the case if the clock is associated with a concrete boolean signal which is already known to be present and whose value is *true* at a particular reactive step. But it is also possible to infer that a clock variable is true based on other values and clocks, thanks to clock constraints. If there are enough constraints on clocks to generate a deterministic reactive system where clocks are always known to be either present or absent at runtime the system is said to be endochronous.

Apart from clocks, Signal also allows to declare dependencies between signals: *{a –> b} when c* is a conditioned dependency that is active only when the formula $\hat{a} \wedge \hat{b} \wedge \hat{c} \wedge c$ is true: (i) clocks $\hat{a}$, $\hat{b}$ and $\hat{c}$ must be present, and (ii) the guard c itself must be true. When this relation holds, the value of b cannot be computed before the value of a is computed first.

The Polychrony toolset[3] provides a model-driven environment for the Signal language, including in particular a compiler, a front-end in the Eclipse platform[4], a model-checker for formal verification and other translators. The Signal compiler organizes computations as a conditional directed acyclic graph where arcs are labelled with clock expressions. Clocks and dependencies allow to easily distribute computations across multiple processing units (e.g. threads), while preventing deadlocks and using a miminum set of communications between units: since clock presence or absence can often be inferred from exchanged values, it is not always necessary to reify them as concrete boolean signals.

1.3 Outline

In section 2, we present our work on an unambiguous static block scheduler for QGen which can compute both partial and total orders based on user preferences. This scheduler was developed to help QGen interoperate with other tools which work on data-flow graphs, like Signal. In section 3, we describe our transformation function from the P language to SSME. In section 4, we discuss some experimental results with respect to our initial objectives.

2 Block sequencing

In the System Model subset of the P formalism, blocks represent computation nodes with input and output ports and signals describe values flowing between ports. Data-flow graphs encode a partial order of block executions: block sequencing is the action of finding a strict total order that refines the partial one. A strict total order is required when compiling reactive systems into sequential code. A compiler that would produce parallel sequences of code would not necessarily have to produce a total order, as long as dependencies are satisfied. A block sequencer must also ensure that circular dependencies are detected and should reject malformed models. A formally verified sequencer was implemented for the previous Geneauto [14] project but could not be used in QGen, which provides its own sequencer.

In the particular case of QGen, System Models are inspired by the Simulink language and as such, systems are hierarchical and may be given user-defined priorities (integers). Moreover, blocks are not necessarily always activated but subject to activation condition and control signals. QGen's earliest version provided a block sequencer that we will call here sequencer O ("original"). We will also describe our alternative sequencer implementation called N ("new" sequencer). Even though sequencer N is not integrated as-is into the current version of QGen, most of the problems found and reported to the development team of QGen while working on sequencer N are now addressed in sequencer O.

2.1 System Model language

2.1.1 Blocks and ports

A system in P is a functional view of a model represented by a hierarchical data-flow graph of blocks linked by directed connections called signals. *Blocks* can be either *system* blocks, *interface* blocks or *elementary* blocks. Blocks have input and output *ports*, which may be either *data* or *control* ports. *Signals* represent an ordered pair of ports. All blocks are named and typed according to an external predefined set of block definitions called a *Block library*. Block types are represented by a string and accept generic key/value parameters which are understood by the compiler to generate Code Models elements. *Elementary* blocks provide the actual computations and are implementation-defined. *Interface* blocks are used to represent the "port-block" family of blocks: for each input and output data or control port of a system, there is a *block* which represents that port inside that system. For example, an *Inport block* I_n inside a system S is an *interface* block having exactly one out-

[3] http://polychrony.inria.fr

[4] Open-source framework POP with the Polarsys Industry Working Group: https://www.polarsys.org/projects/polarsys.pop

put *port* $o(I_n)$ which represents the value received from the input port $in(S)_n$ of S. *Interface* blocks differ from *elementary* blocks in that they hold an attribute called *portReference* which represents the interface port associated with the block. Also, interface blocks implementw some of the ports's semantics: the value of output ports can be *held* or *reset* when a port has been deactivated; *enabling* input control ports are used to trigger events when the input value is either raising, falling or changing.

2.1.2 Atomic systems

A *system* block is a container for nested blocks, and is assumed to be *atomic* in this section (we assume that models we consider have been preprocessed): (i) an atomic system can be computed only when all of its inputs are available; (ii) blocks that rely on outputs of an atomic system can be computed only when all the outputs of the atomic system have been computed; (iii) finally, while an atomic block is being computed, computation must not exit this block until all outputs are computed. See for example the blocks in figure 1: $i1$, $i2$ and $i3$ (resp. $o1$, $o2$ and $o3$) are three inputs (resp. outputs) of current subsystem. Blocks A, B and C are elementary blocks (interface blocks are not represented). Both blocks A and B are grouped into an atomic subsystem we call here S. Simple arrows represent actual signals whereas bold arrows which form crosses represent the dependencies implied by atomicity. The property (iii) above means that it is not possible to compute C between the computations of A and B, even though there is no dependency required by signals. In other words, the sequence (A, C, B) is not a compatible execution order for those three blocks. We cannot express atomicity with static partial dependencies because both (A, B, C) and (C, A, B) are compatible execution orders.

2.1.3 Datatypes

The P language defines a hierarchy of values to represent types in the target language of a compiled model. Types are either *primitive*, *array* or *pointer* types. Arrays represent multi-dimensional arrays and are composed of a base type and a list of dimensions. Among primitive types, there is a class of *numeric* types as well as *boolean*, *string*, *void* and *custom* types. Numeric types are divided into *complex* and *real* types (but complex are currently not used in QGen), where *real* types include both *integer*, *fixed-point* and either *single* or *double* floating-point values. Numeric types are described by a boolean value which indicates signedness (returned by *isSigned()*) as well as a width representing the number of bits available for this type (returned by *getNBits()*). Custom types designate external types. The QGen compiler provides C and Ada definitions for expected types, such as *GAUINT8* for unsigned 8-bits integers, as well as wrappers around mathematical and logical operators for those types.

2.2 Partial ordering of blocks

We defined the sequencer N to help QGen fulfill its interoperability objectives. QGen is expected to fit into existing methodological processes, which explains the strong emphasis put on separating concerns in the compiler: this can be witnessed by the existence of compiler flags for choosing which transformations are applied on a model, as well as for exporting it as an XMI document between steps. In particular, it can be desirable to delegate block sequencing to an external tool, such as SynDEX [12, 17] or Polychrony. The reason is that block sequencing is a type of static scheduling of resources, which can be optimized by specialized tools, especially with distributed code.

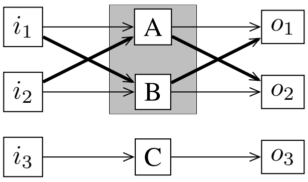


Figure 1: dependencies implied by atomicity of system block (in bold)

A requirement of this approach is that the external tool must provide an ordering that is *compatible* with QGen's ordering of blocks. However, this requires to define a compatibility criterion. For example, is the total order provided by an external tool said to be compatible if that tool ignores user-defined priorities? Similarly, let us consider control signals (see section 2.4.2): QGen's compiler considers that a block B triggered by a control signal emitted from block A is implemented as-if B's code was inlined inside the body of A's code. That semantics leads to additional dependencies which could be seen as artificial for other tools but should be respected when working with QGen.

In order to satisfy different interpretations of what is the minimal set of dependencies the sequencer should consider, we let the user provide a *sorting policy* list made of predefined sort criteria. Those criteria are names of refinement steps to be applied on the dependency graph being built. For example, the DF criterion introduces data-flow dependencies implied by data-flow signals, whereas UP represents those implied by user-defined priorities. The TR criterion forces dependencies of triggered blocks to be dependencies of calling blocks (recursively): this criterion is applied whenever the model is to be compiled by QGen, since it represents the specific way the tool deals with block activations, but is optional because one might want to give a model to another tool. Some criteria represent actions to perform during block sequencing: *LOAD* reads all the dependencies currently stored in a model and introduces them in the dependency graph stored in memory. Another criterion is *ENSURE_TOTAL*, which makes the sequencer abort if that graph still represents a partial order at the moment the criterion is applied. Also, we proposed to modify the existing sequencer so that it can produce either a partial or total order, based on those input parameters.

With this approach, we can avoid the problem of having to implement multiple block sequencers and we can apply the following methodology when working with external toolchains: for some input model M_1, run QGen's sequencer with a given sorting policy P_1, which results in a partial order, exported in model M_2. Pass M_2 to some external tool, which can refine block ordering and produce model M_3. In order to check that M_3 has a compatible order with M_2, run QGen on model M_3 with a modified policy P_2 defined as P_1 followed by a LOAD operation and possibly other refinement steps. Incompatible orderings are then defined as those which introduce circular dependencies in a model. They are detected during the LOAD operation, which could introduce errors if other tools did not respect the original partial order.

This approach needs to be able to represent partial orders in P models. This is done by adding a block attribute named *nextExecutableBlocks*, the list of all blocks that depend on a particular block. We also tried to adapt the original sequencer O so that it could compute partial orders, but there were shortcomings with the existing approach that made it easier to write a more general sequencer N. Sequencer N, unlike O, is split into two parts (two Ada packages): (i) a dedicated yet generic graph data-structure representing partial and total orders and (ii) the actual sequencer which translates blocks and their attributes as a graph while processing the sorting policy given by the user. We detail these in the following sections.

2.3 Graph data-structure

The original sequencer O used to compute dependencies by assigning a rank to blocks. More precisely, blocks with no predecessors are assigned rank 0, and blocks for which all their predecessors have a rank are assigned a rank $M + 1$, where M is the maximal rank of those predecessors. This first step admitted a simple implementation and ensured that data-flow dependencies were respected. However, ranks are natural numbers and give an over-constrained view of data-flow dependencies. In other terms, ranks are already a refinement of the minimal partial order consisting only of data-flow dependencies. While this approach was appropriate in the context of providing a total execution order of blocks, as it was the case for sequencer O, it was difficult to change the implementation to let it represent partial orders.

There are many optimized data-structures for dynamically computing transitive reachability [9]. We implemented a simple graph data-structure based on a dense matrix representation with a fixed set of vertices, for the incremental computation of both the transitive closure and transitive reduction of a graph while dynamically adding links.

Thus, the size required for a matrix with n nodes is $\Theta(n^2)$. Adding links between nodes requires a propagation step which costs $\Theta(n^2)$ in time. This propagation step also detects circular dependencies and maintains both the transitive closure and the transitive reduction of dependencies. Getting the list of all pairs of blocks that are unrelated has a complexity of $\Theta(n^2)$. Checking whether current order is total is a constant operation. In practice, each cell in the matrix requires 2 bits of memory[5]. The number of blocks in a subsystem is generally low enough to not be problematic with respect to the space complexity of our approach.

2.3.1 Invariants and properties

A graph of n nodes is represented by a square matrix M of size n^2. Each number k from 0 to $n - 1$ is associated to a node named n_k. We note $<$ the strict order between nodes that is represented by a graph. Each cell $M_{i,j}$ of the matrix represents whether the relationship $n_i < n_j$ holds between nodes n_i and n_j. $M_{i,j}$ is one of the three following values: *none*, *direct* or *indirect*. We ensure that the following invariants hold in a matrix: (i) a value of *none* at cell $M_{x,z}$ means that $n_x < n_z$ does not hold; (ii) when the $n_x < n_z$ relationship holds, then $M_{x,z}$ is *direct* if and only if there is no indice y such that $n_x < n_y$ and $n_y < n_z$; (iii) otherwise, $M_{x,z}$ is *indirect*. Moreover, the graph is never allowed to contain circular dependencies. The implementation described hereafter maintains the above invariants, which give us the following properties: (1) The set of cells where $M_{i,j} \in \{\textit{direct}, \textit{indirect}\}$ represents the transitive closure of the $<$ dependency relationship. For all indices i and j, a non-*none* value at $M_{i,j}$ means that $n_i < n_j$. (2) The set of cells where $M_{i,j} = \textit{direct}$ is a transitive reduction of the $<$ dependency relationship. For all pair of indices (i, j) such that $M_{i,j}$ is *indirect*, there is a sequence of indices $k_1, ..., k_p$ $(p > 0)$ such as $M_{i,k_1} = M_{k_1,k_2} = \cdots = M_{k_p,j} = \textit{direct}$. (3) If and only if the order represented by $<$ is total, then for all indices i, j either $M_{i,j} = \textit{none}$ or $M_{j,i} = \textit{none}$, but not both. In other words, for a matrix of size n, then $<$ is total if and only if there are exactly $\frac{n(n-1)}{2}$ non-*none* values in the matrix (more than this number would imply a circular path; less would leave at least a pair of nodes unrelated). We rely on the third property above to easily check whether an order is total.

2.3.2 Link addition in a graph

A graph structure is implemented as an Ada package where size is a generic parameter. A connectivity matrix of size n^2 is initialized with *none* values. Also, a variable named *Remaining_Cells* is initialized to $\frac{n(n-1)}{2}$. Everytime a *none* cell in the matrix is changed into another value, *Remaining_Cells* is decreased. Thus, we can implement *Is_Total* as a function which returns whether *Remaining_Cells* is zero. Adding a link between two nodes requires a propagation mechanism to maintain the invariants previously seen. Link addition is split into two procedures, *Basic_Link*, which checks for any circular dependency and decrements *Remaining_Cells*, and *Link*, which propagates the relationship being added to all predecessors and successors. The *Check_Cycles* procedure raises an exception in case of circular dependencies and is defined as follows:

```
procedure Check_Cycles
(Self : in out Graph; From, To : Matrix_Index)
is
begin
   if From = To or Self.Matrix (To, From) > NONE then
      raise Cyclic_Graph;
   end if;
end Check_Cycles;
```

The above is a simplified version of the actual code, which also logs the offending cyclic path. *Basic_Link* is defined as follows:

```
procedure Basic_Link (Self : in out Graph;
                      From, To : Matrix_Index;
                      Link : Link_Type)
is
  Previous : constant Dependency :=
    Self.Matrix (From, To);
begin
   if Previous = NONE then
      Self.Check_Cycles (From, To);
      Self.Remaining_Cells := Self.Remaining_Cells - 1;
   end if;
   if Previous = DIRECT or Previous = NONE then
      Self.Matrix (From, To) := Link;
   end if;
end Basic_Link;
```

Link_Type is a subtype of *Dependency* with excludes *none*. The above procedure ensures that it is not possible to put a

[5] Actual size reported by GNAT when providing the *Pack* directive.

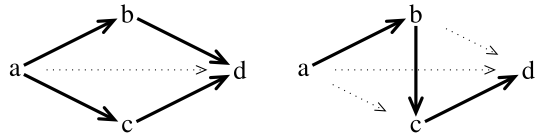
direct link if the link was previously known to be *indirect*. Indeed, we only add a *direct* link if there was previously no link between nodes, to be sure that cells with a *direct* value represent the minimal set of edges covering the whole graph. It is also possible to change a *direct* link to an *indirect* one, during the propagation initiated by the addition of a new, more *direct* link. The previously *direct* relationship can then be deduced transitively, which is why it should be replaced by an indirect one. This is done by propagating indirect relationships inside the *Link* procedure below (the *Self.Precedes* function simply checks that there is a non-*none* value in the matrix).

```
procedure Link
(Self : in out Graph; From : Matrix_Index;
 To : Matrix_Index; Link : Link_Type := DIRECT)
is
  Previous : constant Dependency :=
    Self.Matrix (From, To);
begin
 -- Add the "from < to" relationship
 Self.Basic_Link (From, To, Link);
 if Previous > NONE then
   return; -- Exit early (*)
 end if;

 --  Compute transitive connectivity
 for Y in Matrix_Index'Range loop
  for X in Matrix_Index'Range loop
   declare
    DX : constant Boolean := Self.Precedes (X, From);
    DY : constant Boolean := Self.Precedes (To, Y);
   begin
    if DX then
     -- (x < from) => (x < to)
     Self.Basic_Link (X, To, INDIRECT);
    end if;
    if DY then
     -- (to < y) => (from < y)
     Self.Basic_Link (From, Y, INDIRECT);
    end if;
    if DX and DY then
     -- (x < from),(to < y) => (x < y)
     Self.Basic_Link (X, Y, INDIRECT);
    end if;
   end;
  end loop;
 end loop;
end Link;
```

The above procedure assumes that the graph initially respects the properties expressed in the previous sections and ensures that they hold after adding the new link. When the previous value in our matrix at indices *From* and *To* was *none*, we can exit the procedure early *(*)*. Indeed, by construction a *none* value means that there are no direct or indirect relationship such as *From* < *To*. Moreover, if previously the inverse relationship *To* < *From* held, then *Basic_Link* would have raised an exception due to a cyclic dependency, which is not the case at this point. We can then safely assume that no link propagation is required.

The two nested loops add the required links between predecessors and successors of our nodes. Even though this can seem counter-intuitive, nodes previously known as *direct* are unconditionally changed into *indirect* ones. This is because the new *direct* link being added replaces a *none* value in the matrix, which, thanks to cycle detection, implies that the two nodes being linked were previously unrelated. Hence, the new link is currently the only one that allows to link *From* and *To* and cannot be removed. However, if a predecessor x of *From* used to be a *direct* predecessor of a successor y of *To*, then we have to change it to an *indirect* one since now there is another path from x to y (the previous *direct* link between x and y is not the unique one anymore). This does not have an impact on links which already were *indirect*. See figure 2 for an example of this behavior.



(a) Initial graph (b) Addition of $b < c$

Figure 2: *Direct* links (thick) being changed as *indirect* ones (dotted).

2.3.3 Internal nodes

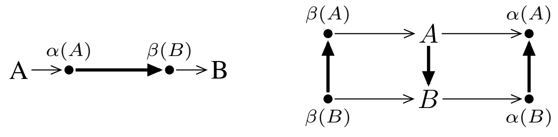
We allow to declare additional nodes in a graph, not bound to actual blocks. For example, an internal node could be used to model the set of all input ports of a block: the internal node would have outgoing links to each input port of the block and we could refer to this node whenever we want to add dependencies for all inputs. We use internal nodes to model trigger dependencies, as detailed in section 2.4.2. In order to handle those nodes, we allowed the size of the matrix n to be defined in terms of two parameters, b (blocks) and e (extra), such that $n = b + e$. The range $0..(b-1)$ represent nodes associated with actual blocks, called *block nodes*, whereas $b..(n-1)$ represents additional nodes called *internal nodes*: only block nodes are taken into account when checking whether current order is total. The overall square matrix is divided into two main zones: a square matrix B from $(0,0)$ to $(b-1, b-1)$ containing links between block nodes, and the remaining L-shaped region I containing links involving at least one internal node. We updated the graph structure with a normalization operation which projects all links inside I as links in the B area. This operation may change *indirect* links to *direct* ones in B and clears area I, while preserving the previous properties. Normalization is automatically applied when collecting all links, before saving current graph. It first clears the I area by resetting all cells to *none*, then recompute actual dependencies for block nodes. Computing the actual dependencies has an effect only on matrix cells $M_{x,z}$ that are set to *indirect*. If there is no index y such that $n_x < n_y < n_z$, then the dependency is changed as a *direct* one.

2.4 Converting dependencies as links

Thanks to the graph structure defined previously, the role of the sequencer is simplified: we only need to convert block relationships and attributes as links. We first show the general approach used when adding dependencies as graph, and then discuss the particular case of trigger dependencies.

2.4.1 Two-step refinement

We strictly apply the following two-step approach when refining a graph according to a sort criterion: *first*, compute the list of all pairs of blocks that are currently unrelated; *then* try to add a link between each pair of blocks according to current criterion. This separation allows us to prevent modifying the graph while looking for unrelated pairs of blocks, which would let the order by which pairs are visited influence block sequencing. For example, let A, B and C

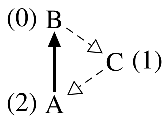


(a) Data signals

(b) Control signals

Figure 3: Converting data and control signals as dependencies

be three blocks; a data-flow signal connects A to B. With $u(x)$ being defined as the user-defined priority of a block x, we set block priorities as follows: $u(A) = 2$, $u(B) = 0$ and $u(C) = 1$.



According to priorities, we have both $C < A$ and $B < C$. In sequencer N, when introducing user-defined priorities, we first collect both (A, C) and (B, C) and then try to add links for each pair. This leads to a cyclic path in the graph which is reported to the user. With the original sequencer O, comparisons are made in an arbitrary order, which depends on implementation details, and eventually give either (A, B, C) or (C, A, B) as an execution order: instead of reporting a circular dependency, sequencing terminates normally. Note however that the order by which sort criteria are processed in a sort policy matters, because some of them are intended to refine a graph (e.g. priorities) whereas others unconditionally add all possible dependencies (e.g. data-flow dependencies).

2.4.2 Trigger dependencies

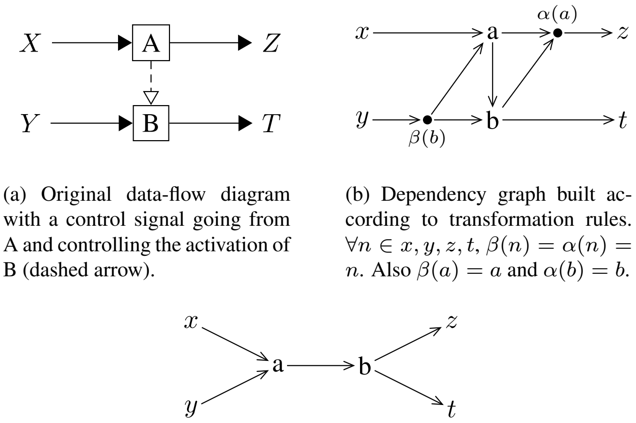
For each block b, we denote $dp(b)$ (resp. $cp(b)$) the set of direct predecessor blocks according to data-flow (resp. control-flow) dependencies. Those sets are determined by visiting incoming control-flow and data-flow signals, while ignoring blocks like *UnitDelay* which have no instantaneous dependency between input and output ports.

We convert data-flow and control-flow dependencies as links inside our dependency matrix. In order to do so, we need to propagate control-flow dependencies so that block activation follows a function-call semantics: when a block A is activated, any block B it controls is also activated and executed during the execution of block A. That means that (i) all the predecessor blocks in $dp(B)$ must already have been executed, and (ii) no block C such that $B \in dp(C)$ can be executed before A itself finishes its execution.

In order to express those constraints, we introduce intermediate nodes in our matrix. For all blocks, we define $\beta(B)$ ("before") as either the node B or the node which is directly preceding B with respect to node dependencies. Likewise $\alpha(B)$ ("after") is either B itself or the node directly succeeding B. Here, "directly" means that by construction we guarantee that no other node is ever scheduled between an intermediate node and the node of the block it represents.

By definition, $\beta(B) = B$ for all block B such that $cp(B) = \emptyset$, and $\alpha(A) = A$ for all block A such as there is no block B in current system block such as $A \in cp(B)$. In other cases, a new node is created and linked to its associated block.

Once those nodes are created, we iterate over all blocks in current system and transform data and control signals as two different patterns of links, as shown in figure 3. Dataflow signals between blocks A and B simply add a link between $\alpha(A)$ and $\beta(B)$. Control-flow signals are translated in such a way that predecessor (resp. successor) blocks of controlled blocks are placed as predecessors (resp. successors) of the controlling blocks. In figure 3b we can see that a trigger between blocks A and B introduces links: (i) between A and B, because B cannot be computed before A, (ii) between $\beta(B)$ and $\beta(A)$ and (iii) between $\alpha(B)$ and $\alpha(A)$. This transformation scheme is done locally, for each block, but dependencies are eventually applied transitively by the underlying data-structure. For example, if block B was controlling a block C, the predecessors and successors of C would be scheduled respectively before and after A. Figure 4 shows an example of transformation from the original data-flow diagram to the resulting dependency graph.



(a) Original data-flow diagram with a control signal going from A and controlling the activation of B (dashed arrow).

(b) Dependency graph built according to transformation rules. $\forall n \in x, y, z, t$, $\beta(n) = \alpha(n) = n$. Also $\beta(a) = a$ and $\alpha(b) = b$.

(c) Equivalent data-flow graph after transitive reduction, as computed by the sequencer.

Figure 4: Elimination of trigger dependencies

3 Conversion of System Models

We provide a transformation mechanism from the System Model subset of P to SSME. For that purpose, we also developed a high-level library on top of the generated EMF classes.

3.1 Definitions

For each element p in the P language we define $\langle p \rangle$ the unique identifier of p. In practice, $\langle p \rangle$ is built from the unique XMI identifier of element p, which is always the string representation of a natural number, prefixed with "P". This naming scheme produces valid SSME identifiers which can be used to link Signal elements to the original P elements from which they are generated. Also, for any symbol m in $\{$*label*, *local*$\}$, $\langle p \rangle_m$ is a valid Signal identifier derived from $\langle p \rangle$ but distinct from it.

The translation of an element x of type T as an SSME element S, with a modifier m and with respect to an environment E, is defined by equations of the form $T_m(E, x{:}T) = S$. The result of applying the translation is denoted $T_m(E, x)$. Modifiers are a way to implement return-type polymorphism, where the behavior of T is specialized according to desired type of the value to be returned. Not all modifiers are meaningful for all types of inputs. A modifier m is an optional symbol: we define $T(E, P) = T_{\emptyset}(E, P)$, where $\emptyset$ is the empty modifier. The returned type of the translation when using the empty modifier is either a process expression or a signal expression, whichever is the most relevant for a given parameter: arithmetic expressions are translated as signal expressions, whereas data-flow connections are translated as equations.

An environment E is a set of zero or more bindings $s_i \mapsto v_i$ from symbols s_i to values v_i. We note $E.s$ the value bound to symbol s in environment E. We define $F = E\left[v_1 \mapsto s_1, \ldots, v_n \mapsto s_n\right]$ any environment F derived from E such that for all i between 1 and n, $F.s_i = v_i$; for any symbol w different from all symbols s_1 to s_n, $F.w = E.w$. For conciseness, $T_m(P)$ represents $T_m(E, P)$ when the environment E can be unambiguously inferred from the context. There is an implicit default environment from which other environments are derived where global options are defined.

3.2 General approach

The conversion function is recursively applied on P hierarchical data-flow graphs. Subsystems and blocks are translated as two nested Signal processes where input and output ports are represented by input and output signals. The outer process accepts inputs and outputs as given from the environment whereas the inner process is only called when a particular block is activated. The outer process is thus responsible for filtering inputs and providing default outputs according to the expected Simulink semantics of blocks. We define an auxiliary Signal process which dynamically computes if a block is *active* given the block's *control-port*, *enable* and *edge-enable* ports:

```
1 process simulink_control =
2 (? event tick, sample, cp; boolean en, eden;
3  ! event active)
4  (| ckeden := ^eden | cken := ^en
5   | enabled := (eden default tick) and
6                (en default tick)
7   | active := cp ^+ ([:enabled]  ^* sample)
8   | active ^# ^0
9   | cp ^# (ckeden ^+ cken)
10  |) where event ckeden, cken; boolean enabled; end;
```

The above process also accepts input *tick* and *sample* events. The *sample* event is related to multirate models which are currently not considered in QGen: as a consequence, the *sample* event is always present in practice. The *simulink_control* process encodes some constraints related to blocks through clock relations: for example, line 9 represents the fact that if a block has a control-port, it cannot also have either an edge-enable or an enable port. Line 7 states that a block is active either if an event is present on its control-port or if the block is both *enabled* and at a tick where it should be computed. The *enabled* boolean is false soon as one of the enable (*en*) or edge-enable (*eden*) port has a false value (this value is computed elsewhere). The *[:enabled]* expression denotes an event that is present only when the boolean *enabled* is true. The above encoding can be used for all variations of system blocks, even if they do not have control-ports or enable ports. Indeed, when generating code for system blocks, it is sufficient to bind some input signals to the null clock to specify that a port is absent, which is then simplified by constant folding.

3.3 Dependencies and atomicity

We convert block-level dependencies as computed by QGen as signal dependencies between labelled process. A labelled process is a call of a process (e.g. a process representing a block) where all inputs and outputs are virtually associated with a label. The label, when used in a dependency, can be used to schedule all the input and output signals of a block before or after those of another process. We convert dependencies according to either the *nextExecutableBlocks* or *executionOrder* attributes of blocks. In order to model atomicity, we add a particular pragma, namely *Unexpanded*, to system blocks. This pragma is sufficient to instruct the Signal compiler to not interleave computations inside an atomic block with computations that exist outside that block.

3.4 System Model

A System Model element is a root element in P. Let s be a System Model containing n elements e_i, $0 \leq i < n$. s is translated in SSME as a list containing a single module M named $\langle s \rangle$ which contains a declaration D and and process definition P:

$$T(E, s : \textit{System Model}) = [M]$$

$$M = \textit{module}(\langle s \rangle\,, [D, P])$$

According to whether $E.type$ is *p* or *signal*, declaration D is respectively either (i) an import statement to a predefined Signal library, namely *import("P")* or (ii) a predefined list of external types τ: *type*(τ, *external*). P is a process q = *process("main",S,B)* with a signature S and a process body B. The P process accepts as many input signals as the union of input ports of all e_i elements and provide as many outputs as the union of output ports. S is thus defined as $\textit{io}(\cup_{i=0}^{n} I_i, \cup_{i=0}^{n} O_i)$, where for each i such that $0 \leq i < n$:

$$I_i = \cup_{p \in \textit{In}(e_i)} T_{\textit{decl}}(p) \quad O_i = \cup_{q \in \textit{Out}(e_i)} T_{\textit{decl}}(q)$$

The body B is a new Signal process defined as:

$$B = \textit{restriction}(\textit{composition}([]), [])$$

We recall that Signal defines two kinds of expressions: (i) signal expressions, which are equations over data-flow variables and (ii) process expressions, which include notably composition and restriction processes. The B process is a *restriction* process, which holds both a sub-expression e and a set of lexically scoped declarations d. Declarations made in d are only visible in d and e. Here, e is a *composition* process, i.e. a set of zero or more parallel processes. The declarations and expressions in B are initially empty but eventually populated by the translation of inner elements. For all elements e_i contained in s, we apply $T(E\left[\textit{module} \mapsto M, \textit{subproc} \mapsto q\right], e_i)$. The *module* and *subproc* symbols respectively represent the root element of the generated SSME element and the process associated with current subssytem: here the "main" process is referenced.

3.5 System blocks

System blocks contain zero or more children blocks. A system block s converted as an expression represents a labelled call to the process representing s. The arguments of the process call are translations of the input ports of s, which are variables declared in current scope. Likewise, the mutliple results of the process call are bound to the signals resulting from the translations of output ports of s.

$$T(s{:}\mathit{System}) = \mathit{labelled}(\langle s\rangle_{\mathit{label}}, \mathit{call}(T_{\mathit{decl}}(s), I, O))$$

$$I = \cup_{p \in \mathit{In}(s)} T(p) \quad O = \cup_{q \in \mathit{Out}(s)} T(q)$$

The declaration associated with a system block is a process definition. Each system block is translated as one *control* process c calling a *body* process b. The control process c is responsible for determining if the block is currently active and feeds the internal body b with filtered inputs. Moreover, the outputs from b are repeated (or replaced by default values) so that c can provide outputs at the same rate as inputs are given: c takes care of merging outputs from b either with a default value or the previous value of each output port, according to each port's *resetWhenDisabled* attribute.

$$\begin{aligned} &T_{\mathit{decl}}(s{:}\mathit{System}) = c = \mathit{node}(\langle s\rangle, S, X) \\ &X = \mathit{restriction}(Y, [D, ldecl]) \\ &Y = \mathit{composition}([ldeps, lclock, event, in, out, call]) \end{aligned}$$

The signature S is computed as above by converting all inputs and outputs of s. The set of declarations of the control process c holds a label declaration *ldecl* and a declaration D for the body process b, detailed thereafter. There are 6 parallel processes being composed, each of them having a specific purpose: *call* is a process call to the body process b; *ldeps* holds a list of inter-label dependencies, which allows to encode the partial order provided by the input model; *lclock* contains clock equalities for labels, indicating that all contained blocks are executed synchronously; *event* calls the *simulink_control* external process and contained translations of control, enable and edge-enable ports according to their attributes; *in* and *out* correspond respectively to filtering and merging equations, where local variables are declared for all input and output data-ports. That process b is declared in D in a modified environment F:

$$F = E \begin{bmatrix} subproc \mapsto b, & ll \mapsto ldecl, \\ ls \mapsto ldeps, & lc \mapsto lclock \end{bmatrix}$$

In addition to *subproc*, other symbols keep a reference to other parts of the control process where *convert* can add information for each block contained in s. Then, D holds a process definition for b with the same input/output signature as c and the recursive conversion of each block of s as expressions in environment F. During this conversion, *convert* populates the labels declarations and constraints held in the control process c.

3.6 Elementary blocks

By default, elementary blocks are expressed as external processes in Signal. The reason for this is that we do not aim to provide a complete implementation of Simulink's semantics in parallel to the one currently implemented by QGen. Indeed, there are over a hundred of blocks defined in QGen's default Block library and each of them allows multiple set of configurations that may have an important impact on their behavior. For example, the *Sum* block can process scalars, vectors and matrices depending on the data linked to its input ports. We would rather reuse the existing code generation mechanism of the compiler to produce target code. Indeed, QGen's Block library defines, for each block, a set of functions that provide the imperative statements implementing the different steps of computation of this block, namely *Make_Memory_Variables()*, *Get_InitStatements()*, *Get_ComputeStatements()* and *Get_MemUpdateStatements()*. Those functions encode the actual semantics of each block according to its parameters and provide statements or declarations in a subset of P expressing code, called Code Model (e.g. statements, operators, functions). Thus, we are interested in converting the generated Code Model elements as signal expressions instead of providing an ad-hoc implementation of each possible block. We currently support only a small subset of Code Model elements. The approach consisting in compiling Code Model elements requires to build a control-flow graph, as commonly done in compilers [16]. However, we cannot currently assume that Code Model programs generated by QGen are in a static single assignment (SSA) form [7], which would simplify data-flow analysis [18] and could let us exploit our previous results with translation from C/C++ to Signal [2].

3.7 Datatypes

We propose two different ways of translating P data-types to SSME. First, we can treat all types as external types and translate mathematical operations as call to external processes. Alternatively, we can translate types as Signal types whenever possible, which tend to produce simpler code that does not depend on the external library of types provided by QGen. In both cases, it is sometimes necessary to convert values directly as Signal constants, like in array indices for which QGen does not define a specialized type. There are however limitations with this approach, because Signal does not define unsigned types nor 64 bits integers. The translation is straightforward, except that when an unsigned type is requested, we use a larger signed type so that all values can be represented. Also, we produce warnings if the required number of bits is too large for Signal.

4 Validation and applications

We validated our approach with the test suite used by QGen which is composed of over two-hundred small-sized Simulink models. We tested both block sequencing and model transformations.

4.1 Side-by-side testing of sequencers

We ensured that our implementation did not regress from the previous one by (i) developing a dedicated sorting policy and a ranking function for our implementation that replicated the behavior of the existing sequencer, and (ii) comparing the results of both implementations with side-by-side tests. Those tests showed discrepancies for exactly ten models which were acknowledged to be due to some defects in sequencer O: (i) the failure to reject models where some blocks cannot be unambiguously sorted, either because there were circular dependencies or because blocks were treated as equal (we found out that the tie-breaking comparison functions on block names could fail because systems could contain blocks having the same name after preprocessing; this was fixed by using the fully qualified name of blocks) and (ii) the incompatibility between the computed order and the compilation strategy regarding triggered blocks (sequencer O would produce an order which would not describe how triggers are implemented).

4.2 From P to SSME

We managed to load and process a public use case provided by Rockwell Collins France composed of safety-critical components implementing display logic for helicopter data. The XMI file resulting from QGen's compilation weights 9.7MiB and is loaded as an EMF runtime object in our Clojure environment in over 9 seconds; Comparatively, translation to SSME takes between 1 and 2 seconds, which is the same time required for pretty-printing the same model as a Signal textual file; exporting the runtime model as an XMI document takes a little less than 500ms. The produced model is made of 287 Signal processes which mirror the hierarchical organization of the original model. It represents the data-flow and control-flow constraints among blocks inside the system. We can process that model with Polychrony in order to compute a flattened network of blocks grouped by common dependencies into 109 separate clusters of sequential code. Polychrony can produce multithreaded code from clusters where threads synchronize themselves either with a message-passing protocol or with signal/wait operations [3].

We also experimented with the balance drive controller model present in QGen's test suite. We modified the input model so that each subsystem is declared to be atomic in order to obtain a hierarchy of Signal processes after translation. Then, we collected all original P subsystems and generated a map from their unique identifiers to a unique natural number, starting from zero and increasing. With this temporary map, we dynamically added *RunOn* directives [3] in our SSME model to each of its processes in order to bind subsystems to distinct execution units. This manual step requires fifteen lines of interactive code and simulates a partitioning of our system according to a possible architectural description of it. Then, the distributed Signal model can be used to generate distributed code that complies with *RunOn* directives. An automatic translation of architectural P elements could be eventually feasible in future versions: even though there is an existing architecture description language in P which allows to declare processors and buses, as well as non-functional properties such as the period and deadline of a task, there is currently no way to map System Models to architectural ones.

4.3 From SSME to P

Our original intent when integrating Signal and QGen was to use Signal as a model optimizer for P, with respect to static block scheduling, timing and architectural properties. For example, starting from a multirate model, i.e. a model where blocks are sampled at different periods, we could group blocks into different asynchronous components while preserving or adding synchronization flows when necessary. In order to perform this task, our tool is expected to return modified P models. In practice, we effectively have access to a model m, its translation s in SSME and the model s_p obtained by running Polychrony on s with specific parameters p. With the known mapping between m and s, we can determine whether an element in s_p was originally present in s or if that element is introduced by Polychrony itself. However, we do not have a systematic transformation scheme to build a meaningful copy of m taking into account the modifications of s brought by s_p at the model level. Instead of trying to modify existing models, we are now implementing a general purpose transformation from Signal to P, as part of Polychrony, which systematically converts endochronous processes as a hierarchy of triggered subsystems following the original clock hierarchy. Alternatively, it would be easier to directly generate P Code Model instead of System Models.

5 Conclusion

We developed an alternative block sequencer for QGen for the purpose of computing both partial and total orders from input models. The purpose of this sequencer is to allow QGen to interoperate with external sequencing tools while providing guarantees about the compatibility of external block execution orders with respect to both QGen's compilation scheme and user expectations. This sequencer is available as a separate tool and not fully integrated inside QGen. Our work contributed nonetheless to test and fix the existing codebase of QGen.

We also presented a model transformation tool from the P language used inside the QGen model compiler to the SSME language representing synchronous Signal programs. This work is based on a high-level API designed on top of SSME and can be used to transform a subset of Simulink to Signal. We ran the conversion tool and the set of models used by QGen for its regression tests and successfully converted medium to large models. The P language is capable of representing a useful subset of Simulink. That is why it is an interesting tool to help interpreting Simulink models and possibly architectural properties as executable Signal programs. Our perspective for this tool is to add support for the conversion of more Code Model elements, as generated by QGen, in order to produce executable programs with Signal. The programs currently produced with our transformation tool can be compiled by Polychrony and reorganized as clusters of smaller processes. We expect QGen to eventually provide an architecture description language inside P, and our perspective with regard to P remains to be able to automatically distribute models given some of their architectural properties while preserving synchronization constraints.

6 Acknowledgments

We thank the anonymous reviewers and especially our “shepherd” Eric Jenn for their helpful feedback.

References

[1] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events

and relations: the SIGNAL language and its semantics. *Science of computer programming*, 16(2):103–149, 1991.

[2] Loï Besnard, Thierry Gautier, Matthieu Moy, Jean-Pierre Talpin, Kenneth Johnson, and Florence Maraninchi. Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form. In *Ninth International Workshop on Automated Verification of Cirtical Systems (AVOCS'09)*, 2009.

[3] Loïc Besnard, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. Compilation of polychronous data flow equations. In *Synthesis of Embedded Software*, pages 1–40. Springer, 2010.

[4] Matteo Bordin and Franco Gasperoni. Towards verifying model compilers. In *5th International Congress and exhibition ERTS2*, 2010.

[5] Matteo Bordin, Tonu Naks, Andres Toom, and Marc Pantel. Compilation of heterogeneous models: Motivations and challenges. In *European symposium on Real Time Software and Systems (ERTS), Toulouse*, volume 29, pages 08–01. Citeseer, 2008.

[6] Alan Burns. The ravenscar profile. *ACM SIGAda Ada Letters*, 19(4):49–52, 1999.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[8] James B Dabney and Thomas L Harman. *Mastering simulink*. Pearson, 2004.

[9] Camil Demetrescu and Giuseppe F. Italiano. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms*, 4(3):353 – 383, 2006. Special issue in honour of Giorgio Ausiello.

[10] Francois-Xavier Dormoy. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, pages 1–9, 2008.

[11] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.

[12] Oussama Feki, Thierry Grandpierre, Mohamed Akil, Nouri Masmoudi, and Yves Sorel. SynDEx-Mix: A hardware/software partitioning CAD tool. In *Sciences and Techniques of Automatic Control and Computer Engineering (STA), 15th International Conference*, pages 247–252. IEEE, 2014.

[13] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[14] Nassima Izerrouken, Olivier Ssi Yan Kai, Marc Pantel, and Xavier Thirioux. Use of formal methods for building qualified code generator for safer automotive systems. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 53–56. ACM, 2010.

[15] Nassima Izerrouken, Xavier Thirioux, Marc Pantel, and Martin Strecker. Certifying an Automated Code Generator Using Formal Tools : Preliminary Experiments in the GeneAuto Project. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*, 2008.

[16] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[17] Christophe Lavarenne, Omar Seghrouchni, Yves Sorel, and Michel Sorine. The syndex software environment for real-time distributed systems design and implementation. In *European Control Conference*, volume 2, pages 1684–1689. Citeseer, 1991.

[18] Alexandre Lenart, Christopher Sadler, and Sandeep K. S. Gupta. SSA-based flow-sensitive type analysis: combining constant and type propagation. In *Proceedings of the ACM Symposium on Applied Computing*, pages 813–817, 2000.

[19] Yue Ma, Jean-Pierre Talpin, and Thierry Gautier. Virtual prototyping AADL architectures in a polychronous model of computation. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 139–148. IEEE, 2008.

[20] MISRA-C MISRA. Guidelines for the use of the C Language in vehicle based software. *Motor Industry Research Association, UK*, 1998.

[21] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 226–231. IEEE, 2009.