A MDE-based optimisation process for Real-Time systems

Olivier Gilles
GET-Telecom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
olivier.gilles@enst.fr

Jérome Hugues
Université de Toulouse, ISAE
10, avenue E. Belin - BP 54032,
31055 Toulouse CEDEX 4, France
jerome.hugues@isae.fr

Abstract—The design and implementation of Real-Time Embedded Systems is now heavily relying on Model-Driven Engineering (MDE) as a central place to define and then analyze or implement a system. MDE toolchains are taking a key role as to gather most of functional and not functional properties in a central framework, and then exploit this information. Such toolchain is based on both 1) a modeling notation, and 2) companion tools to transform or analyze models.

In this paper, we present a MDE-based process for system optimisation based on an architectural description. We first define a generic evaluation pipeline, define a library of elementary transformations and then shows how to use it through Domain-Specific Language to evaluate and then transform models. We illustrate this process on an AADL case study modeling a Generic Avionics Platform.

I. INTRODUCTION

Real-Time embedded systems (RTES) have to reconcile functional correctness and strict adherence to timing requirements. Such systems define both a hardware and a software architecture, and check they are matching. Different processes can be followed, but they usually revolve around two approaches. In an architecture-centric approach, a performance envelope of the system is defined, based on an architectural design, with threads, processes and interconnection through several buses. Then, this architecture is validated, and populated with functional elements. In a function-centric approach, the opposite path is followed: performance requirements are elicited from functional blocks. The choice of one process is mandated by the industry: to meet hardware requirements or to select hardware.

Hence, one needs to be able to 1) validate an architecture based on actual performance metrics, and 2) in some cases to correct the initial model in case of performance mismatch, to optimise it. Optimising functional code is now a well-established techniques, based on careful selection of algorithm, compiler tuning and profiling. Yet, optimising software architectures (set of threads/processes and connections) tends to overlook the issue of system evaluation, and either perform a single-step optimization at compile-time, or trust a formal model of the system to effectively demonstrate the value of a given optimized RTES instance. This approach used to be quite efficient with deterministic hardware and software components.

Unfortunately, it is becoming less meaningful for current systems, because of perturbations induced by caches, MMUs [11] or protocols. Furthermore, current optimization tools generally try to enhance just one aspect of the application (either memory footprint or response time), without regards for the actual needs of the final RTES. The choice of the optimization criterion and the verification of its effectiveness is left to the system integrator. This may come late in the process: recovering from non-working systems at this stage can have a big impact.

Model-Driven Engineering provides a framework to gather information related to both functional and non-functional blocks of a system, and then have a full view of system performance. In this paper, we propose a user-defined optimization process. We show how to build an evaluation pipeline (section II) on top of an architecture description language; elementary model transformations for optimisation (section III) and finally how to drive optimisation based on a DSL to evaluate an architecture and then to optimise it (section V). We conclude with a case study, derived from a Generic Avionics Platform.

II. EVALUATION PIPELINE

Evaluation of the RTES performances strongly depends on hardware resources and scheduling constraints. When considering optimisation, one has to evaluate the architecture, and balance criteria. We consider the system is fully defined as one ADL-based model such as MARTE or AADL. As there is no a unique criteria, we want to give freedom to the end designer, to do so, we have to solve three challenges: 1) Defining evaluation criteria, 2) evaluating architecture, that is computing each value, and 3) combine them in an evaluation pipeline. We use AADL for our work; other formalisms like MARTE would offer similar support.

A. Evaluation criteria

Although the notion of performances on a RTE System is highly dependent on the system domain, one can define three typical performance factors: schedule, data flow latency and memory. Their valuation heavily depends on the topology of the system (nodes and threads connection patterns), the scheduling policy and resource allocation strategies. Computing these values can be performed by third-part tools.
In some cases, these are simple computations based on elementary formulas, e.g. summing a property values over a set of components.

We have defined REAL [3], a domain specific language as an annex for the AADL. REAL allows one to define formulae for each criteria, and to attach them to model components. Listing 1 shows how to compute the worst case execution time of a set of threads.

| — Computes the WCET of a set of threads
| theorem threads_wcet
|   foreach th in Local_Set do
|     var wcet := last (property (th, "Compute_Execution_Time"));
|     return (Max (Mum (wcet)));
| end threads_wcet; |

Listing 1. REAL example

Computed values can then be put back in the original AADL model using a MDE framework. For simplicity and readability, we chose to store them in a new version of the architectural model used for code generation: the annotated model. In this paper, references to the model actually design the annotated model, since model annotation is the first natural step towards optimization.

We defined a library of functions to evaluate local criteria using REAL, and then combined them to build one global value, biased towards some architectural patterns:

- **Maximum distance to deadline**, for each non-schedulable thread – and minimum distance to deadline whenever the system is actually schedulable;
- **Maximum memory overhead**, for each overloaded process, and minimum free memory;
- **Maximum task response time** for each top-level sub-program, or distance to the response time upper limit if such value had been defined.

### B. Evaluating an architecture

An architecture modeled in AADL is just one artifact, to be exploited. Actually, one can define three stages where to evaluate model’s performance:

- **Model-Level Evaluation**, which evaluates the criteria values on the current model;
- **Operation-Level Evaluation**, which computes the value of the criteria after the application of a set of transformation operations on the current model.
- **Binary-Level Evaluation**, which measures the value of the criteria on the actual system executable generated from the current model;

**Model Level Evaluation** (MLE) relies on information computed on the current model. It is a direct application of REAL formulas, or external tools like schedulability that can be verified directly on the model using Cheddar [14].

**Operation Level Evaluation** (OLE) relies on a priori knowledge of the impact of some optimisation steps. Impacts of the different operations are presented in section III. This computation provides an estimate and does not ensure full accuracy. Yet, MLE and OLE can be performed at model-level and can be quite efficient to reduce the number of candidate architectures during optimisation.

In some occasion, one can generate code from a model. We take advantage of this new source to evaluate the model deeply. **Binary level evaluation** (BLE) relies on external tools which measures the binaries WCET and memory footprint. This information is more precise than a priori values from OLE, yet is more time-consuming: one has to generate code, compile it and runs benchmarks or other tools.

To fetch the information required for BLE, we use a set of internal and external tools. To manipulate the AADL models and generate a full application from a set of AADL models and functional code, we use the Ocarina toolsuite [7]. In addition, Bound-T [10] allows to extract WCET and stack size from application built for SPARC-like processor. We take advantage of static memory pre-allocation enforced by Ocarina to use GNU binutils and get information on memory consumption. Instrumentation can provide additional information. To reduce manual work, we exploit the information contained in the architectural model used for system generation. In our experiments, we used the AADL [13], which allows to define properties for any components or connections. We used this feature to add properties corresponding to the information needed by each tool.

### C. Evaluation pipeline

Figure 1 illustrates the evaluation pipeline that implements this process. In red are the different levels of evaluation, while the yellow ellipse shows the current architectural model as initial state. As the figure suggests, a system to be optimized can be evaluated in 3 ways, and the actions for each. There are two branches: full model-based evaluation, and binary-level evaluation. Doted lines shows actions done when an evaluation does not select a system. This generic evaluation pipeline needs to: 1) select a candidate transformation, or 2) decide on the actual evaluation to be performed. These are controlled by the evaluation criteria defined by the user, through the library of evaluation functions and the optimisation algorithms selected.

Let us note BLE is the most time consuming path: it implies two more stages : code generation and compilation, and binary analysis. It should be performed only at key steps of the global optimisation process.

The outcome of one run of the evaluation pipeline is to decide the list optimisation transformation to be performed to build a new intermediate model. We list elementary transformations we designed in the next section.

### III. ELEMENTARY TRANSFORMATIONS

We defined elementary operations that allow to explore the different configurations of the system: merging, moving
or splitting thread activities. In this study, we focus on merge and move, and make the hypothesis that threads execute just one function (at the model level), so splitting threads is not relevant.

A. Merge

The Merge operation produces a new thread that will encompass the legacy code of the former threads, and dispatch them at required rate. In order to achieve this operation, an automaton is generated, with low memory and instruction overhead (a switch/case construct, an enumeration declaration and an array of bounded size) to guarantee determinism. This automaton will be dispatched at a rate which is the new thread period, defined by the greater common divider (GCD) of the former threads periods. A local scheduler ensures that at any dispatch the automaton triggers the due code. Detailed explanations about the merge operation can be found in [4]. Merge operation impacts performances:

- by allowing to serialize inter-thread connections, it allows to remove synchronizations constructs such as mutexes, thus reducing the measured WCET;
- by factorizing the system resources, it allows to decrease memory usage;
- by reducing the number of potential context switches, it reduce slightly the actual WCET and the scheduling complexity;
- however, we should note that by removing possible preemption, it decreases the system schedulability.

B. Move

The Move operation migrates a thread from one process to another one. The connections to other blocs are maintained to preserve semantics, thus changing the configuration of inter-process connections. We do not allow to move threads accessing to local pool of data, since it would imply to build new connections and is unlikely to lead to system optimization.

While the operation does not induce overhead, it does impact on the system behaviour: it impacts the pattern of inter-thread communications and then lead to changes the process buffers and synchronization constructs. It also has an indirect impact by allowing or restricting the number of potential merge in either source or target processes. While distribution-related impacts have not been studied in the scope of our work, we defined two obvious impacts of this operation at local level:

- change source/target process CPU load and thus schedulability;
- change source/target memory occupation.

This is to be noticed that indirect impact does not have to be measured at that stage, since it is to be revealed by model evaluation in a following iteration.

IV. Operation-Level and Model-Level Evaluation with REAL

We saw that REAL could be used in order to perform queries on the AADL model. In this section, we explain how to perform criteria evaluation with REAL, either at model or operation level. As an example, we illustrate this process for the Minimum Distance To Deadline (MDTD) criteria.

A. Distance to deadline at Model Level

The Merge operation impacts the model structure, since it serializes previously concurrent subprograms. While performing model-level evaluation of the MDTD, one should consider that the threads evaluated are either optimized (i.e. have been previously merged) or not.

In a regular AADL models, threads non-functional values needed by evaluation are directly associated with the thread component with standard AADL properties: Deadline and Compute_Execution_Time (the latter being the WCET). Computing a single distance to deadline can thus be done by the REAL theorem illustrated in listing 2, where the Local_Set is expected to contain a single element (this property is verified by the unique subtheorem).

```plaintext
theorem distance_to_deadline_regular

foreach th in Local_Set do
    var wcet := last (property (th, "Compute_Execution_Time"));
    var deadline := property (th, "Deadline");
    requires (unique);
    return (Msum (wcet − deadline));
```
In a optimized thread, the actual WCET can be approximated by the sum of the WCET of the subprograms that it can run in a single execution. In the worst case — during the hyperperiod — all the subprograms can be run in a single dispatch. Thus, an optimized thread WCET can be described as the sum of all the WCET of the subprograms that it call directly. Those subprograms can be associated with WCET with the standard AADL property Compute\_Execution\_Time. Thus, we define the REAL theorem 3, which compute the distance to deadline of an optimized thread. In this theorem, the Is\_Called\_By predicate return true whenever the first parameters is called (hence, is a subprogram) by the first one (which can be either a thread or a subprogram). We notice that the property function can be applied to sets, in which case it returns a list of values. In the same way, last can be applied to a list of range, in which case it return a list of integer (composed of the last value of any range in the former list).

Finally, computing the MDTD consists in compute the minimum values of the different distances to deadline in the system. Thus we defined the REAL theorem 4. In this theorem, we predict whether the current thread is optimized or not by using the Fusion\_Occurred property, defined in the Transformations property set, which is set (with the value true) on all thread resulting from a Merge.

In the case of Merge operation, the operation itself will impact on the MDTD value, since it will create a new thread with possibly tighter distance to deadline. Hence, we must compute this new thread MDTD before actually build it. In order to do so, we build a set which contains the threads candidate to merging, and pass it to the theorem that will compute the new MDTD into the Local\_Set. We use the theorem 1 illustrated above, in order to compute the sum of the WCETs of the candidate threads. The GCD function compute the Greatest Common Divisor between the parameter-given list — thus it computes the future period of the merged thread. Since we iterate on the system set, the final expression is only computed once.

Finally, computing the MDTD consists in compute the minimum values of the different distances to deadline in the system. Thus we defined the REAL theorem 4. In this theorem, we predict whether the current thread is optimized or not by using the Fusion\_Occurred property, defined in the Transformations property set, which is set (with the value true) on all thread resulting from a Merge.

B. Distance to deadline at Operation Level

In the case of Merge operation, the operation itself will impact on the MDTD value, since it will create a new thread with possibly tighter distance to deadline. Hence, we must compute this new thread MDTD before actually build it. In order to do so, we build a set which contains the threads candidate to merging, and pass it to the theorem that will compute the new MDTD into the Local\_Set. We use the theorem 1 illustrated above, in order to compute the sum of the WCETs of the candidate threads. The GCD function compute the Greatest Common Divisor between the parameter-given list — thus it computes the future period of the merged thread. Since we iterate on the system set, the final expression is only computed once.

Finally, computing the MDTD consists in compute the minimum values of the different distances to deadline in the system. Thus we defined the REAL theorem 4. In this theorem, we predict whether the current thread is optimized or not by using the Fusion\_Occurred property, defined in the Transformations property set, which is set (with the value true) on all thread resulting from a Merge.

B. Distance to deadline at Operation Level

In the case of Merge operation, the operation itself will impact on the MDTD value, since it will create a new thread with possibly tighter distance to deadline. Hence, we must compute this new thread MDTD before actually build it. In order to do so, we build a set which contains the threads candidate to merging, and pass it to the theorem that will compute the new MDTD into the Local\_Set. We use the theorem 1 illustrated above, in order to compute the sum of the WCETs of the candidate threads. The GCD function compute the Greatest Common Divisor between the parameter-given list — thus it computes the future period of the merged thread. Since we iterate on the system set, the final expression is only computed once.

Finally, computing the MDTD consists in compute the minimum values of the different distances to deadline in the system. Thus we defined the REAL theorem 4. In this theorem, we predict whether the current thread is optimized or not by using the Fusion\_Occurred property, defined in the Transformations property set, which is set (with the value true) on all thread resulting from a Merge.

B. Distance to deadline at Operation Level

In the case of Merge operation, the operation itself will impact on the MDTD value, since it will create a new thread with possibly tighter distance to deadline. Hence, we must compute this new thread MDTD before actually build it. In order to do so, we build a set which contains the threads candidate to merging, and pass it to the theorem that will compute the new MDTD into the Local\_Set. We use the theorem 1 illustrated above, in order to compute the sum of the WCETs of the candidate threads. The GCD function compute the Greatest Common Divisor between the parameter-given list — thus it computes the future period of the merged thread. Since we iterate on the system set, the final expression is only computed once.

Finally, computing the MDTD consists in compute the minimum values of the different distances to deadline in the system. Thus we defined the REAL theorem 4. In this theorem, we predict whether the current thread is optimized or not by using the Fusion\_Occurred property, defined in the Transformations property set, which is set (with the value true) on all thread resulting from a Merge.
In order to elect the first element of the set of merging candidate, we use the maximum number of in and out connections connected to other threads of the same process, because we do know from previous studies than serializing connections allows to reduce significantly the system WCET. Other heuristics could be easily specified.

This solution theoretical complexity in terms of operation is in $O(n^3)$, $n$ being the maximum number of threads in a process. However, it is usually quite lower, for the same reasons than the fully greedy solution.

This solution theoretically complexity in terms of operations is in $O(n^4)$, $n$ being the maximum number of threads in a process. However, it is usually quite lower, for the same reasons than the fully greedy solution.

### B. Half Greedy Heuristic

Using the criteria defined in II-A, we propose an algorithm that explore a larger part of the solution graph than the Full Greedy Heuristic, and thus is expected to return a result closer to the optimal.

This solution (algorithm 2) consists in building the optimal set of merge for each thread, and then select the better result according to our evaluation to perform the actual merge. This operation is repeated until no more merge is possible. One should note that this procedure actually has some greediness since we do not explore all the combinations of merging set but only the most optimal next merge at each step. This explains the “half-greedy” denomination.

This solution theoretical complexity in terms of operations is in $O(n^4)$, $n$ being the maximum number of thread in a process. However, it is usually quite lower, for the same reasons than the fully greedy solution.

#### Algorithm 1: Fully Greedy Algorithm

**Input:** System $S$

**forall p ∈ Process(S) do**

repeat

Sort(Threads($p$))

$T ← First(Threads(p))$

Candidate$_Set ← \emptyset$

repeat

$Best_Value ← 0$

forall $t2 ∈ Threads(p)$ do

if $t2 \neq T$ then

$Current_Value ← Compute_Value(Candidate_Set, t2)$

if $Current_Value > Best_Value$ then

$Best_Value ← Current_Value$

$Best_Candidate ← t2$

end

end

Candidate$_Set ← Candidate_Set ∪ Best_Candidate$

until noBest_CandidateFound ;

if Candidate$_Set \neq \emptyset$ then

$S ← Merge(Candidate_Set)$

$S ← Move(Candidate_Set,S)$

end

until Candidate$_Set = \emptyset$ ;

end

#### Algorithm 2: Half Greedy Algorithm

**Input:** System $S$

forall $p ∈ Process(S)$ do

repeat

$Best_Set_Value ← 0$

$Final_Set ← \emptyset$

forall $T ∈ Threads(p)$ do

$T ← First(Threads(p))$

Candidate$_Set ← \emptyset$

repeat

$Best_Value ← 0$

forall $t2 ∈ Threads(p)$ do

if $t2 \neq T$ then

$Current_Value ← Compute_Value(Candidate_Set, t2)$

if $Current_Value > Best_Value$ then

$Best_Value ← Current_Value$

$Best_Candidate ← t2$

end

end

Candidate$_Set ← Candidate_Set ∪ Best_Candidate$

until noBest_CandidateFound ;

if Best$_Value > Best_Set_Value$ then

$Best_Set_Value ← Best_Value$

$Final_Set ← Candidate_Set$

end

if $Final_Set \neq \emptyset$ then

$S ← Merge(Final_Set)$

$S ← Move(Final_Set, S)$

end

until $Final_Set = \emptyset$ ;

end

### VI. Test Case: The Generic Avionic Platform

We selected a case study to assess our solution, based on an abstraction of a complete system. The Software Engineering Institute at CMU, the Naval Weapons Center and IBM’s Federal Sector Division participated in the creation of the Generic Avionic Platform (GAP), as reported in [8], in the 80s. This model as been designed first to assess suitability of early revisions of the Ada language [9], although this model is no longer representative of current avionics architecture, it provides a freely available definition of a meaningful
RTES. We chose to model this system in AADLv2. Figure 2 illustrates its main threads, data flows and processes, in a representation which only take account of connection existence (multiples connections between two threads are represented by a single connection).

Figure 2. GAP main data flows

The GAP defines 16 threads, either periodic or sporadic, with different periods/minimum interarrival times, and a great yet heterogeneous amount of connections. Because of its complexity, the specification followed a functionality-oriented modeling, and offered schedulable implementations of the GAP. Following our optimisation process, we were able to merge those threads into only 5 threads, while preserving the global schedulability of the system. In the following, we discuss the different experiments performed.

In order to demonstrate the modularity of evaluation techniques, we run the optimization algorithms with the evaluation criteria described in section II-A:

- **connection-based**, which search for the maximum number of inter-connections in a set of threads;
- **deadline-based**, which search for the GCD of thread deadlines closer to the set of threads’ maximal deadlines.

Table I shows the content of the threads built by both Full Greedy Optimization (FGO) and Half Greedy Optimization (HGO), with the connection criteria for operation evaluation, and the period-based one. ‘+’ symbol denotes thread addition, i.e. two threads being present, ‘x’ denotes composition of threads. Apart from the move operation which moves a merged thread from Displays to Weapons in the HGO version of the model, we can see than the resulting models are slightly different. We discuss these differences below.

Table II reports time to perform the whole optimisation process, measured on the time to execute the algorithm and other tasks related to model management (parsing, manipulation, . . .). We note that most algorithms take a few seconds to complete. Memory consumption is decreased by more than 30% in each case. This mostly results in the merging of threads that reduce memory at runtime. Let us note that in all configurations scheduling is preserved.

1) **Fully Greedy Algorithm results:**

- **Displays** : With FGO, we impose the first operand of the merge operation. As indicated above, the main criteria for choosing this thread is the number of connections with others threads. In our example, it elects the Builtin_Test thread, which receives data from nearly all others threads in the process. The first merge done is with MPD_Status_Display, because it shares many connections with the former one. Target_Tracking, the second thread to be merged is chosen because of its connection with the previous, although its period is dangerously low (40 ms). Then a set of control and display threads are merged, because their higher periods and low CPU usage make their merging costless. The new thread has a period of 40 ms.

Table II

<table>
<thead>
<tr>
<th></th>
<th>FGO-CB</th>
<th>HGO-CB</th>
<th>FGO-PB</th>
<th>HGO-PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
<td>536</td>
<td>86</td>
<td>723</td>
<td>101</td>
</tr>
<tr>
<td>Duration (s)</td>
<td>4.19</td>
<td>2.63</td>
<td>2.88</td>
<td>3.37</td>
</tr>
<tr>
<td>Memory gain</td>
<td>30%</td>
<td>32%</td>
<td>39%</td>
<td>36%</td>
</tr>
</tbody>
</table>

A. **Connection-based optimisations**

Table I shows the content of the threads built by both Full Greedy Optimization (FGO) and Half Greedy Optimization (HGO), with the connection criteria for operation evaluation, and the period-based one. ‘+’ symbol denotes thread addition, i.e. two threads being present, ‘x’ denotes composition of threads. Apart from the move operation which moves a merged thread from Displays to Weapons in the HGO version of the model, we can see than the resulting models are slightly different. We discuss these differences below.

Table II reports time to perform the whole optimisation process, measured on the time to execute the algorithm and other tasks related to model management (parsing, manipulation, . . .). We note that most algorithms take a few seconds to complete. Memory consumption is decreased by more than 30% in each case. This mostly results in the merging of threads that reduce memory at runtime. Let us note that in all configurations scheduling is preserved.

1) **Fully Greedy Algorithm results:**

- **Displays** : With FGO, we impose the first operand of the merge operation. As indicated above, the main criteria for choosing this thread is the number of connections with others threads. In our example, it elects the Builtin_Test thread, which receives data from nearly all others threads in the process. The first merge done is with MPD_Status_Display, because it shares many connections with the former one. Target_Tracking, the second thread to be merged is chosen because of its connection with the previous, although its period is dangerously low (40 ms). Then a set of control and display threads are merged, because their higher periods and low CPU usage make their merging costless. The new thread has a period of 40 ms.

Since no other thread can be added, HUD_Display is selected as next candidate to merge. Its period being of 52 ms, thus the decomposition in prime numbers is 13-2, it is quite unlikely to support many merges. MPD_Tactical_Display, however, has also a period of 52 ms, thus is selected to merge. Finally, a third merge candidate is elected amongst the two staying thread (RWR_Threat_Response and RWR_Control). Their respective periods being of 100 ms and 400 ms, the merge actually occurs and produce a thread which period is 100 ms. Since no other threads in the process have WCET and periods allowing new merges, a move is tried, although it will not apply, since the current
<table>
<thead>
<tr>
<th>Original</th>
<th>Weapons</th>
<th>Weapon_Selection (Sporadic, 200ms) + Weapon_Trajectory (Sporadic, 100ms) + Weapon_Release (Sporadic, 200ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Display</td>
<td>HUD_Display (Periodic, 52ms) + MPD_Tactical (Periodic, 52ms) + Radar_Control (Periodic, 40ms) + Target_Track + MPD_Status (Periodic, 200ms) + MPD_Stores (Periodic, 200ms) + Builtin_Test (Periodic, 100ms) + Keyset (Periodic, 200ms) + HOTAS (Periodic, 40ms) + WWR_Threat (Periodic, 100ms) + WWR_Control (Sporadic, 400ms)</td>
</tr>
<tr>
<td></td>
<td>FGO-CB System</td>
<td>Weapons (Weapon_Selection x Weapon_Trajectory x Weapon_Release)</td>
</tr>
<tr>
<td></td>
<td>Display</td>
<td>(HUD_Display x MPD_Tactical) + (Radar_Control x Target_Track x MPD_Status x MPD_Stores x Builtin_Test x Keyset x HOTAS) + (WWR_Threat x WWR_Control)</td>
</tr>
<tr>
<td></td>
<td>HGO-CB System</td>
<td>Weapons (WWR_Control x Radar_Control x MPD_Status x MPD_Stores x Builtin_Test x Keyset x HOTAS) + (Weapon_Selection x Weapon_Trajectory x Weapon_Release)</td>
</tr>
<tr>
<td></td>
<td>Display</td>
<td>(HUD_Display x HUD_Tactical)</td>
</tr>
<tr>
<td></td>
<td>FGO-PB System</td>
<td>Weapons (Weapon_Selection x Weapon_Trajectory x Weapon_Release) + (Target_Track x Radar_Control x HOTAS)</td>
</tr>
<tr>
<td></td>
<td>Display</td>
<td>(HUD_Display x MPD_Tactical) + (MPD_Status x MPD_Stores x Builtin_Test x Keyset x WWR_Threat x RWR_Control)</td>
</tr>
<tr>
<td></td>
<td>HGO-PB System</td>
<td>Weapons (MPD_Status x MPD_Stores x Builtin_Test x Keyset x RWR_Control x Weapon_Selection x Weapon_Trajectory x Weapon_Release) + (HUD_Display x HUD_Tactical)</td>
</tr>
<tr>
<td></td>
<td>Display</td>
<td>(Target_Track x Radar_Control x HOTAS)</td>
</tr>
</tbody>
</table>

Table I  
ITERRATIONS OF THE GAP PLATFORM AFTER DIFFERENT OPTIMISATIONS.

A second iteration of the algorithm failed to find new optimization, thus stopping the algorithm, with an effective complexity of 536 operation evaluations.

2) Half Greedy Algorithm results:

- **Displays**: From the evaluation criteria, the algorithm searches first the nearer deadlines. A second iteration of the process merges low-deadline threads (all deadlines are 40 ms) into the new thread: thr_1. A second set of three threads of period 200 ms is then merged into a new one. Finally, two threads of periods of 52 ms are merged into the third thread. Threads with unique periods are then processed: all have a GCD of 100 ms, and thus they are merged with the second one whose period become 100 ms. No move is performed.
- **Weapons**: Two threads of period 200 ms are merged into one, then the merged thread (Target_Track x Radar_Control x HOTAS) is moved from Displays.
- **Navigation**: Like in previous execution, no merge nor move are possible.

A second iteration impacts Weapons, and trigger the merge of the last non-merged thread (of period 100 ms), and (Weapon_Selection x Weapon_Trajectory), changing its period to 100 ms. The algorithm then stop, with a total of 723 operation evaluations.

2) Half Greedy Algorithm results: The same than the connection-based evaluation is selected for merging. Then, although the order vary, the threads built are the same than in the half-greedy version, yet for a total of 101 operation evaluations.

B. Deadline-based optimisations

1) Fully Greedy Algorithm results:

- **Displays**: Compared to the fully-greedy algorithm, the first set to be selected in the test case includes WWR_Control but excludes Target_Track, because its tight period limits the number of further potential merges. Like in the fully-greedy algorithm, HUD_Display and MPD_Tactical_DISPLAY are merged. Finally, WWR_Threat_Response stays as is, suffering of its lack of connections with other threads.
- **Weapons**: It contains three strongly connected sporadic threads, with different minimum inter-arrival times (MIAT), all multiples of 100 ms. MIAT, however, is relative to a given signal, thus it should not be modified by the merging operation. Those three thread are merged into one, since their respective WCET allow their execution during the minimum MIAT of the merged threads. The move operation select thread from the overloaded Displays process, choosing randomly the first merged thread Target_Tracking for moving to Weapons.
- **Navigation**: Like in the half greedy algorithm, no merge nor move is performed.

A second iteration of the algorithm tries to optimize each new version of the processes, which have been modified by the last move operation. In our case however, tight periods make this step impossible, and thus stop the algorithm, after 86 operation evaluations.

VII. RELATED WORKS

The optimisation of models, prior to code generation, has already been discussed in various works.

In [12], authors discuss optimisation of architecture implemented as Simulink blocks. Yet, this work is restricted to one family of systems, and lacks control from the user.
In [2], the authors evaluate a bin-packing algorithm to allocate processes to processors in an AADL model. The level of granularity is that of a pool of threads. This approach allows one to deploy an application on a set of CPUs. Compared to this approach, our contribution proposes model rewriting strategies to achieve better CPU and memory usage while preserving schedulability, at thread-level. Both contributions are complementary.

In [1], authors evaluate an optimization tool for optimizing memory footprint of CCM-based applications. The proposed approach relies on the merge of CCM components, for soft real time system. This approach is similar to the one we propose. Yet, our contribution relies on a lighter middleware (the AADL runtime, implemented by our POLYORB-HI runtime), which is finely optimised, and on stricter scheduling discipline. Therefore, we extend this work to mission critical, hard real-time systems.

Furthermore, compared to most optimisation techniques, we provide control over the metrics to guide the optimisation process. We believe this is a requirement to address heterogeneity in RTES architectures.

VIII. Conclusion

Model-Driven Engineering is an appealing technology for building real-time systems. It allows one to focus on core functional and non-functional aspects of a system, prior to validation and code generation. However, optimisations of the final system are seldom addressed at model-level, and left to the integration phase where it is performed manually. In the worst case, the overall systems need to be redesigned if the system does not meet requirement.

In previous work, we have developed a suite of tools around AADL to support code generation targeting optimised runtimes for the high-integrity domain; and later shown how to use this information to gain precise information on compute execution time based on precise evaluation of the models or the executable.

Considering elementary transformation steps, and a DSL to evaluate architecture characteristics, we proposed a user-driven optimisation process to optimize an architecture: the user can define its own evaluation functions, and then use them in an optimisation process. We proposed two variants of an optimisation algorithm and different evaluation metrics, and applied them to a representative architecture modeling an avionics platform. Thanks to a mix of model-level and binary-level evaluation techniques, our results indicate the approach can tackle optimisation results and help system designers to reduce memory footprint or meet stricter schedules while preserving its non-functional properties.

Future works will extend our study to distributed applications, by taking into account communication time in the choice of specific merge or reorganisation of the model. Another extension is to careful review code generation patterns to remove useless synchronisation when scheduling policies and careful off-line schedule allow it.

References


