Comput Syst Sci & Eng (2011) 1: 3–18

International Journal of
Computer Systems
Science & Engineering

A MDE-based optimisation process for Real-Time systems: Optimizing systems at the architecture-level using the real DSL and library of transformation and heuristics

Olivier Gilles and Jerome Hugues

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 non-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.
Yet, we note the modeling process is driven by the engineer view of the system to be built. This view may fit a particular vision, e.g. a functional breakdown, but usually overlook another like hardware constraints. Thus, a re-factoring of the model may be required to have a better ?t and optimize resources to actual CPU or memory resources. Such rewriting may be time-consuming to ensure the semantics is preserved. Optimization is a typical step in a compiler. As for typical compilation techniques, we claim that MDE toolchains would benefit from automatic optimization techniques that preserve execution semantics, schedulability or other non-functional constraints. In this paper, we present a first step towards MDE-based system optimisation based on an architectural description. We first define a generic evaluation pipeline to assess model metrics. We then define a library of elementary transformations and show how to apply it to evaluate and then transform models using a Domain-Specific Language. Finally, we illustrate this process on an AADL case study modeling a Generic Avionics Platform.

Keywords: MDE, AADL, optimization, model transformation, domain-specific language.

1. 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 architectures, and check they are matching. Different implementation 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 components.

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 algorithms, compiler tuning and profiling of the generated machine code depending on the target processor. Yet, optimising software architectures (set of threads/processes and connections, use of mutexes or semaphores) 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 [18] or communication 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 nonworking systems at this stage can have a big impact.

In the mean time, Model-Driven Engineering emerged as a convenient support for modeling large system. It 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 architecture, blocks. From this view, it becomes possible to reason on the performance of the system as a whole, taking into account all its facets.

In this paper, we propose several contributions to support user-defined optimization process in a MDE process. This process is built on 1) the architecture description language AADL to model the system, 2) the domain-specific language REAL to evaluate metrics on this model, and 3) a user-parametric library of heuristics to evaluate relevant performance metrics. These elements are combined to form an iterative evaluation pipeline that explores the design space and looks for an optimal rewriting of the model elements.

Section 2 presents both the AADL and REAL. We present the evaluation pipeline in section 3 and show in section 4 how to extract information from AADL model elements to feed this pipeline. Optimization is built on a restricted set of transformations that optimize either memory or speed. We present these transformations in section 5. The selection of an optimal solution is driven by an heuristic that helps exploring the set of possible solutions. We implemented two variants based on greedy heuristics derived from the knapsack problem (section 6). We illustrate this approach, and associated results, on a complete case study derived from the Generic Avionics Platform in section 7.

2. OVERVIEW OF AADL AND REAL

In this section, we give a brief overview of the foundation bricks we use: AADL and REAL.

2.1 The SAE Architecture Analysis and Design Language AADL

The SAE AADL [21] is an international standard defining the basics of architecture description language dedicated to the design of real-time systems standardized by the SAE. AADL is component-centric and allows to specify both software and hardware parts of a systems. It allows one to define consistent block interfaces, and to separate them from block implementation.

An AADL model is made of *components*. Software components (data, thread, thread group, subprogram, process) are distinguished from execution platform components (memory, bus, processor, device) and hybrid components (system). Each component category mimics the semantics of their counterparts in embedded systems engineering.

The behavior of a system (e.g. how functional blocks interact) is fully defined in the standard by mean of "properties" (attributes with a dedicated semantics) to progressively refine the semantics of a system,
e.g. dispatching invariants, communication patterns; non-functional properties (such as timing, priorities, etc) applied to each model element; non-functional aspects of components can be described within an AADL model such as thread dispatching condition (periodic or sporadic), interface specifications and how components are interconnected. These have a deep impact on the system's behavior. Functional aspects (algorithmic specifications) are attached separately as source code by means of AADL properties. An introduction to AADL can be found in [5].

AADL proposes several user-defined extension mechanisms through property sets and annex languages.

- Property sets allow one to define custom properties to extend standard ones. This is the path taken by the "Data modeling annex document" [24] that allows one to model precisely data types to be manipulated, or the "ARINC annex document" [22] that defines patterns for modeling ARINC653 systems based on the Integrated Modular Avionics pattern.
- AADL annex languages offer the possibility to attach additional considerations to an AADL component like behavioral specification [23]. They bind a domain-specific language to components.

These extensions mechanisms are of particular interest to attach project-specific concerns to an architecture for further analysis such as electric power consumption, modeling of precise performance of buses, or error modeling. The combination of properties and languages enable in-depth system analysis.

Figure 1 illustrates an architecture with 4 AADL threads. Two periodic threads, LP Sender and HP Sender, whose periods and priorities are respectively 12 and 1000 *ms*, and 10 and 2000 *ms*, execute subprograms that send data events through their out event data port msg out to the Receiver sporadic thread in event data port msg in, with a Minimum Inter-Arrival Time (MIAT) of 500 ms. This calls the awakening of the call sequence containing the receiver subprogram at priority 6. A periodic Handler thread is dispatched at priority 8 each 1000 *ms* to perform health monitoring. Information on the worst-case execution time of each

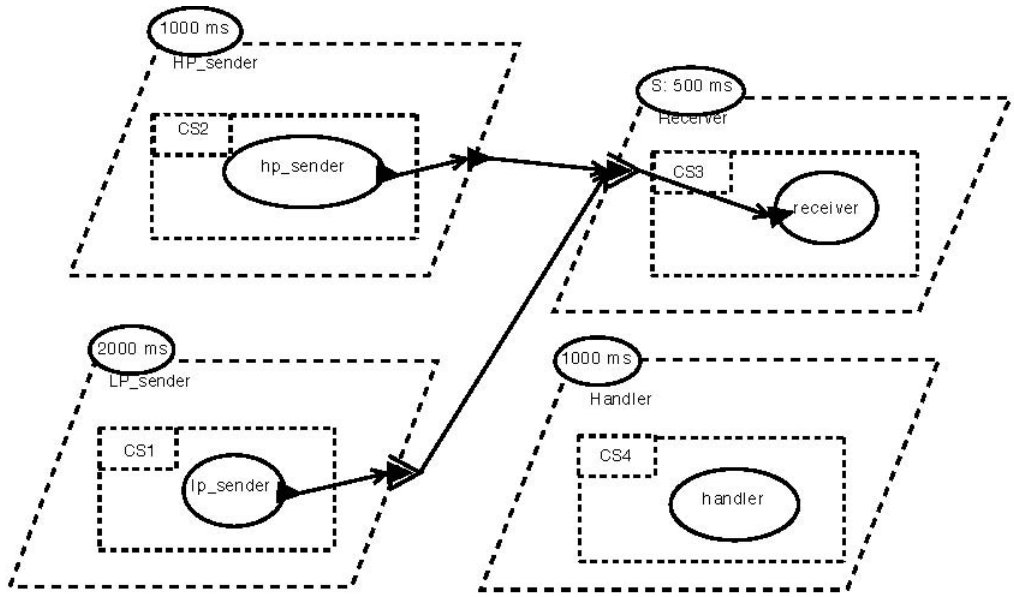

Figure 1 An AADL model.

functions, and overhead from the execution platform (duration of a context switch, time to send an event, etc.) enables full analysis of temporal behavior; whereas properties on the code and stack size of each thread would enable memory analysis.

2.2 REAL, an AADLv2 annex language

We note the analysis criteria can be quite large and cover concerns like memory, scheduling, power, security, safety, ... Besides, the evaluation of metrics be project or platform dependant, and rely on different analysis frameworks. Thus, one needs a versatile way to express and compute metrics.

These considerations lead us to define a AADL language annex: REAL. REAL (Requirement Enforcement Analysis Language) aims at checking constraints enforcement on architectural descriptions at the specification step, saving significant time over verification at execution time. In this section, we describe the main features of this language. REAL pursues multiple design goals:

- Enabling easy navigation through AADL meta-model elements, yet being at a high-level abstraction. To do so, we discarded the use of the UML Object Constraint Language (OCL) and decided to define a specific DSL based on AADL meta-model concepts to ease writing of constraints.
- Allowing to define generic rules. We note that mathematics universal quantifiers (∀, ∃) notation is interesting to define metrics that can apply to a wide range of models, not just specific instances.
- Allowing for modularity through definition of separate constraints that can be later combined.
- Being integrated to the AADL as an annex language, so that constraints are coupled to models in the model repository.

From these goals, we defined REAL with the following design decisions: REAL is based on set theory and associated mathematical notations. The basic unit of REAL is a theorem. A theorem verifies an expression over all the elements of a set that is called the range set. It allows one to build sets whose elements are AADL entities (connections, components or subprogram calls). Verification or computations can then be performed on either a set or its elements by stating Boolean expressions.

In order to write complex expressions, one can use predefined sets, which contain the instances of the AADL model of a given type, or build intermediary sets, using relations between elements of sets (e.g. returns the elements of the set A which are subcomponents of any elements of the set B). 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 (Msum (wcet));

end threads wcet ;
```

Listing 1: REAL example

Subtheorems calls can be used to extract values computed from range sets different than the current one -thus allowing constructs like *get all the instances of threads which periodicity is equal to the minimum periodicity in the system*. These can also be used to define pre-required constraints on the model.

Finally, subtheorems calls can also be used to build local or global variables, or to check pre-required constraints on the model. Callee theorems inherit during theorem interpretation from the caller environment (the local set), and the user can pass parameters. Thus, it is possible to design a library of theorems that will be used by higher-level, user-defined theorems.

REAL [7, 6] has been integrated as an annex language in

OCARINA [14], our AADL toolsuite. We present full examples of REAL in the next sections and show how it can help computing metrics of AADL models to drive an optimization process.

Let us note other formalisms like UML/MARTE would offer similar support. Yet, the expression power of AADL, combined with its simple extensions mechanisms to attach DSL to the core language provide a very pragmatic way to solve our problem.

3. 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 an evaluation strategy of the architecture, that is the best way to evaluate an architecture key metrics;
2. Defining single evaluation criteria,
3. Exploit these metrics to rewrite partially the system's architecture.

The combination of these three elements enables the definition of an evaluation pipeline: the automated execution of these steps to lead to an optimized variant of the user's model as output.

3.1 Evaluating an architecture

An architecture modeled in AADL is just one artifact representing an embedded system 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 [25].

Let us note the method we use is quite general. We focused only on per-subprogram WCET analysis. Integrating end-to-end flows computations like in [13] would provide a higher-level estimate of the actual end-to-end latency of complex computation chains.

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 5. This computation provides an estimate and does not ensure full accuracy. Yet, both 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 of inputs to evaluate the model in more depth. *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 [14]. In addition, Bound-T [17] allows to extract WCET and stack size from application built for SPARC-like processor. We take advantage of code generation from AADL performed by Ocarina, and on its code generation strategies to compute both memory and WCET values. The code generated only use static memory pre-allocation. Hence, we can use GNU BINUTILS and get information on memory consumption. Furthermore, we exploit the AADL models to derive a configuration file for guiding Bound-T in the evaluation of the WCET (see [8]).

The combination of AADL models, code generation and evaluation using REAL provides an efficient way to compute all information without user intervention. This is an important gain to perform automatic optimization of the whole system.

3.2 Evaluation criteria

Although the notion of performances on a Real-Time Embedded 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 and, e.g. summing a property values over a set of components; or requiring a dedicated tool for performing evaluation.

In this study, we only consider the first case, using REAL to express metrics based on the static view an architecture: hierarchy of components and resource metrics such as worst-case execution time of both the user code and the run-time environment, and static memory consumption.

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 have 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 subprogram, or distance to the response time upper limit if such value had been defined.

Other functions have been defined to check that the architecture is not overloaded: response time analysis for schedulability and global memory consumption. These metrics have been defined to express the "laxity" of the model, that is an evaluation of the available resources, and ensure that a potential rewriting of the model does not overload the system. Their careful combination will drive the global optimization loop: metrics are computed, then an optimization algorithm will select a transformation operation and apply it to generate an annotated model; and reiterate the process until it converges.

Let us note we defined these metrics to refiect particular optimization criteria. Since these are written in a separate generic language, other metrics can be computed and adapted for the system in consideration.

We illustrate this process for the *Minimum Distance To Deadline* (MDTD) criteria. They are formed of two metrics that compute the distance to deadline (or laxity). This criteria indicates how fiexible in timing the architecture is.

3.3 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).

```
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 ) ) ;
end distance to deadline regular ;
```

Listing 2: Distance to deadline on non-fusioned threads

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 3. 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 theorem resulting from a Merge.

```
theorem minimum distance
to deadline
foreach th in Thread Set do var distance := if exists
( th , " Transformations : : Fusion
Occurred ")
then compute
distance to deadline optimized ( th )
else compute
distance to
deadline regular (th); return (Mmin (distance));
end minimum distance to deadline ;
```

Listing 3: Minimum distance to deadline

3.4 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 creating it. 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.

```
theorem distance to deadline candidate
foreach s in System
Set do
var wcet := compute threads wcet ( Local Set ) ;
var period := GCD (property (Local Set, "Dead-
line"));
return (Mmin ( wcet - deadline ) ) ;
end distance
to
deadline
candidate;
```

Listing 4: Compute distance to deadline of a new thread

4. EVALUATION PIPELINE

The steps we described in the previous sections provide a way to perform the evaluation of one particular configuration. From the output of one step, one can decide which optimization step is to be taken. We define an "evaluation pipeline" that combines all these steps to perform a global optimization of the system.

Figure 2 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. 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. The precise evaluation of WCET at binary level relies on complex computations over the whole call graph of the system [26], and is the most time-consuming part of the process. Other steps like OLE and MLE only perform some computations over the model and are much faster. Thus, BLE 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.

In the next section, we present the intermediate description of model activities that is used by all evaluations steps. This intermediate description extracts only the information relevant to our optimisation scheme.

Then, we list elementary transformations we designed in section 5. Besides, the selection of the actual transformation can be driven by different algorithms, we discuss this in section 6.

5. FROM AADL MODELS TO ACTIVITIES

The AADL provides a rich way to define an embedded system. Yet, we only need a subset of this information to perform evaluation and optimization. We consider monolithic (non distributed) systems modeled using the AADL, and we focus on the optimisation of time and memory resources. We assume the scheduler in place follows the SCHED FIFO policy of RT-POSIX, or equivalent policy. We assume all tasks are either periodic, or sporadic (with a minimum inter-arrival time between events) to test for schedulability using Response Time Analysis.

Optimisation is typical in the compiler domain [1]: a program is abstracted, and transformed to reduce copy of variables, to propagate constants, or to unroll loops. . . We follow a similar pattern and abstract AADL models as a set of activities, carrying one atomic unit of computation.

From a high-level perspective, an AADL model is a set of interconnected components, *subprograms* define leaf piece of functional elements, *call sequences* define a list of subprograms by one *thread*. Threads are interconnected through ports. Subprograms can exchange event and data through threads' ports. Let us note that call sequences gather subprograms, yet compilers can aggressively optimize them. Therefore, we do not consider subprograms individually, but rather consider call sequences only.

Hence, the key abstraction in an AADL model we retain is the call sequence that uses a set of ports in *in* or *out* mode and execute subprograms. A Thread is reduced to the mechanism by which a subprogram is executed at some point in time. By reallocating subprograms to threads, we can reduce memory consumption while preserving scheduling.

5.1 Call sequences

We divide a call sequence in two aspects : functional aspects and temporal aspects. The functional aspects of a call sequence are defined as CSF=(ICS, $OCS(SP)i$), where ICS (resp. OCS) are input (resp. output) ports used by the call sequence, and SP is a suite of subprograms to be executed. We define the temporal aspect of a call sequence as CST=($TIWPS$) where

$$CS_T : \begin{cases} T & \text{is a triggering event, either timed event or event sent by some other entities} \\ I & \text{minimum interval between dispatch (or period)} \\ W & \text{the worst-case execution time;} \\ P & \text{priority of execution;} \\ S & \text{a function that maps input data and events set ICS to an output data set OCS:} \end{cases}$$

6. THREADS

A thread is a non-empty set of call sequences. Sequences within a thread cannot preempt each others, while they can preempt other thread call sequences, based on the scheduling policy. A thread also embeds calls to the runtime to interact with other threads.

We define a thread as : T= ((CST)TIP), where (CST)$T$is the set of call sequences run by T, and $I$the minimum interval of dispatch of T, which always verifies : I= $GCD(\{ICS|CS \in CST\})$, and P is the priority of the thread. With the definition of call sequence presented above, and to avoid dynamic priority changes, we need to assess that $\forall CS \in \quad TPCS = PT$

In the test case presented in 1, we have 4 threads, each one containing a call sequence. In terms of temporal point of view, we describe them respectively as :

- *LP Sender* = (($CS1$)2000), with $CS1$ =(*clock*, 2000200, 12(*clock* $\rightarrow$ (*msg out*)))
- *HP Sender* = (($CS2$)1000), with $CS2$ =(*clock* 100020010(*clock* $\rightarrow$ (*msg out*)))
- *Receiver* = (($CS3$), 1000), with $CS3$ =(*msg in*5002006(*msg in* $\rightarrow$ ()))
- *Handler* = (($CS4$)1000), with $CS4$ =(*clock*10002008())

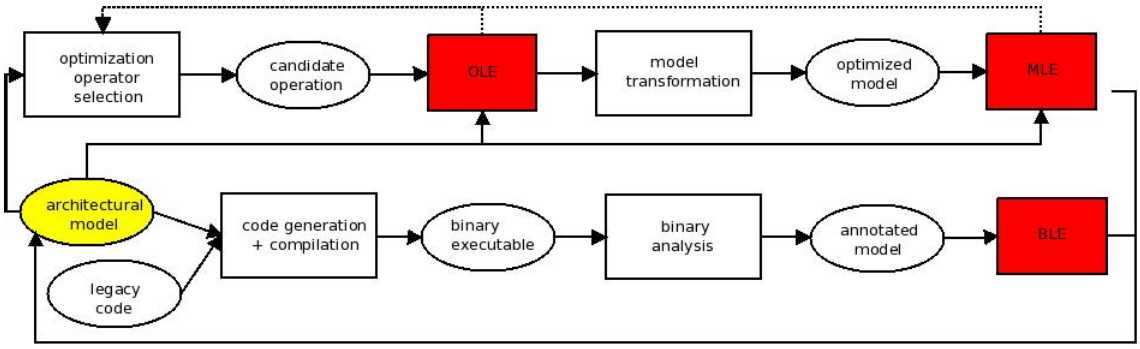

Figure 2 Evaluation pipeline.

where *clock* is a periodic dispatch event.

Having defined this intermediate representation, we now define atomic optimization operations.

7. OPTIMISATION THROUGH THE REALLOCATION OF ACTIVITIES TO THREADS

We propose three 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.

7.1 Move

The Move operation migrates a thread from one process to another one. The connections to other blocks are maintained to preserve the fiow of information exchange, thus changing the configuration of inter-process connections. We do not allow to move threads accessing to local pool of data from one process to another, since it would imply to build new connections and additional code to access the data remotely. This is unlikely to lead to a decrease in resource usage, and thus does.

While the operation does not induce overhead, it does impact 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.

The implementation of the Move operation and its selection relies only on the AADL topology and an evaluation of the load of each process.

7.2 Merge

The Merge operation produces a new thread that will encompass the legacy code of the former threads, and dispatch them at required rate. Its implementation relies on the notion of activity we introduced in the previous section. Merging two threads imply building a new call sequence with an equivalent semantics.

In order to support 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. The 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;
- yet, by removing possible preemption, it decreases the system schedulability.

In the general case, allocating call sequences to the same thread is equivalent to merging threads. Merging two threads implies building one new thread whose behavior is equivalent to the global behavior of the two threads: call sequences are dispatched when triggering events are received, while preserving their periods and deadlines.

We reduce the problem space to the study of three cases: 1) merge of two sporadic call sequences, 2) merge of two periodic call sequences, and 3) merge of a periodic and sporadic call sequences.

7.3 Case 1: Sporadic call sequences.

They are dispatched when an event occurs on one of its *in* ports, after the MIAT time has elapsed. To infer the resulting thread pattern, two factors have to be considered : the input events senders identities and the MIAT value of each thread to be merged.

Let us consider we want to allocate $CS1$ and $CS2$ to the same thread. If $CS2$ is dispatched as a result of an event from $CS1$, then $CS2$ is said to be dependent of $CS1$. At the model level, a dependency between two threads appears as a connection between the two threads passing through their respective event or event data ports.

In the general case, connections on AADL model are purely informative, $CS2$ may send no event at the end of its dispatch. As a result, in the merged thread, both call sequences will only depend on their respective trigger ports, and thus stay sporadic. The dependency will be represented as a connection from $CS1$'s out ports to $CS2$'s in port. In case of non-dependent call sequences, the same result remains.

As a consequence, the thread that will run the two call sequences will also be sporadic.

7.4 Case 2: Periodic call sequences.

Allocating two periodic call sequences to one thread means the resulting thread is able to schedule both call sequences at the correct period. Since a thread cannot stop the execution of one call sequence to start a new one, it means that we need to build a static off-line scheduling, and test its feasibility prior to the merge.

Each call sequence from the original threads will create a call sequence in the merged thread. Those call sequences will be executed sequentially at dispatch time. Figure 3 shows two periodic threads candidates for merge. The call sequences share a direct dependency. At each dispatch, the thread needs to select the list of call sequence to be run. Yet, AADL forbids a thread to call more than one call sequence.

To preserve model readability, we use the following modeling pattern: each dispatch will execute a call sequence that is the concatenation of call sequences to be run, each call sequence is attached to an AADL *mode*, a mode being one possible configuration of a thread. Mode is computed by the thread after each dispatch. By doing so, we preserve at modeling level the information of the different dispatch options. We call this pattern *"mode shifting"*.

Figure 4 illustrates the thread resulting from the merge of two periodic threads. Two call sequences, CS1 and CS2, send data to different ports. Color difference indicates that they are executed in two different modes.

In the figure, colored subprograms are added to handle local scheduling of the two call sequences:

- G1 and G2 are the temporal guards which control the dispatch time of CS1 and CS2;

Like in the sporadic case, a self-connection appears because of the direct dependency between the call sequences.

7.5 Case 3: Periodic and sporadic call sequences

Assigning both a periodic and a sporadic call sequence to a thread is the exact definition of an AADLv2 *Hybrid* thread. Therefore, the transformation is trivial.

The same transformation holds if one want to add another sporadic or periodic call sequence. In this case we fall back to the previous transformations. We detail these situations in the next sections.

7.6 Formal Description of the Merge Process

In this section, we define the transformations we discussed in the previous section formally.

From the previous discussion, we can define the merge function of two threads, so that for a couple of threads $T1$, $T2$ we build a new thread whose call sequences are a combination of the initial ones, and whose temporal properties derive from the properties of the initial threads.

The resulting threads need to be able to execute the same call sequences, at dispatch time that are compatible with the initial schedule. We identify two required corrections:

- The initial threads were dispatched by the executive, in a preemptive manner at specific instants. The resulting thread should dispatch the same call sequences, in a non-preemptive fashion. The thread should built a local scheduler that executes each call sequence. See algorithm 1.

- The initial threads were dispatched with different priorities. If we end up with one thread in the system, everything will be dispatched by the local scheduler. If the merge process ends up with more than one thread, then we need to preserve also the running priority of the thread.

In case of periodic call sequences merge, we need to preserve the system behaviour. The behaviour of a set of periodic tasks is defined by a period, yet a thread can execute itself only at a given, constant, rate.

To emulate the behavior of multiple threads in a single one, we build an off-line schedule of the original system. This scheduler defines which thread is to be executed at any point of the time. Then we divide the time between equals intervals short enough to include the tasks that are dispatched at the same time in the original system. This interval will be the new thread period. For each thread dispatch instant, we compute the set of tasks to be dispatched.

This cannot be done in the general case, and requires the initial set of call sequences matches one precondition: there is no call sequence with a total WCET greater than the period, as it will delay the next dispatch. The implementation of the schedule takes the form of a list of call sequences. Such list is bounded, since the execution schema actually repeats itself every least common multiple of the call sequences, so this notion -called hyper-period -is sufficient in order to define a schedule applicable at any time of the application.

We made the following design choice for our algorithm:

- to dispatch all call sequences, we need to define a minimal time base to dispatch the task, that will in turn execute call sequences. We choose the greatest common denominator among original tasks periods.

- if the periods of call sequences are non-harmonic, then there could be no call sequence to be executed at some dispatch

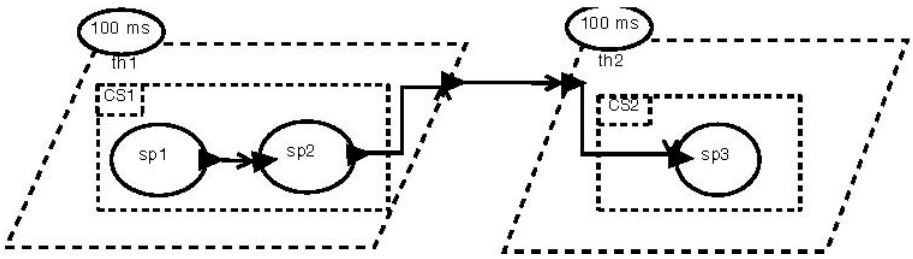

Figure 3 Two periodic threads.

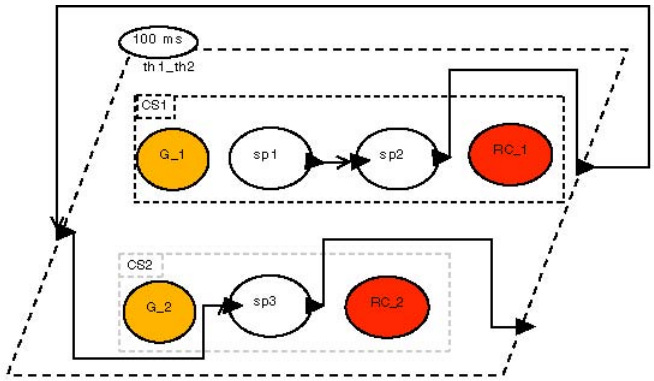

Figure 4 Merge of two periodic threads RC 1 (resp. RC 2) computes the next mode to be executed after the execution of CS1 (resp. CS2).

Input: $CS : Call\_Sequence[]$ – *Assume sorted*
Output: $Schedule : Integer[]$
$Quantum \leftarrow GCD(period(T1), period(T2))$ $HyperPeriod \leftarrow LCM(period(T1), period(T2))$
$Iter \leftarrow 0$
$Exec \leftarrow 1$
while $Iter \leq HyperPeriod$ **do**
  $CS_j \leftarrow First(CS)$
  while $CS_j \leq Last(CS)$ **do**
    $Per \leftarrow Period(CS_j)/Quantum$
    if $Iter \bmod Per = 0$ **then**
      $Schedule(Exec) \leftarrow in\_mode(CS_j)$
      $Exec \leftarrow Exec + 1$
    end
    $Iter \leftarrow Iter + 1$
    $Schedule(Exec) \leftarrow Temporal\_Guard$
    $Exec \leftarrow Exec + 1$
  end
end

Algorithm 1: Schedule builder algorithm.

instant. We add a temporal guard to prevent early execution of the call sequence

Transformation 5.1 (Rewriting call sequence #1). *If two call sequences are periodic, then we apply algorithm 1 to compute an offiine scheduling. If this algorithm applies, then two operators G and R are defined. They define the temporal guard and reconfiguration operators that will schedule CS call sequences. Each call sequence CSiis rewritten as:*

$$R1 : CSi \rightarrow$$

$$(G(CSi)CSiR(CSi))$$

.

In case of call sequences having to be executed at a given time of the hyper-period, one must define a priority policy. We consider: *Priority* of the pre-merged thread or call sequence, and *Temporal Precedence* of the call sequences.

In a typical, non-merged 1-call sequence thread, the call sequence inherits the thread priority. In a merged, multiple call-sequence thread, we need to define the priority of the thread. We chose to avoid explicitly dynamic priority change to preserve system determinism, and follow the recommendations from the Ravenscar profile [4]. We chose to pick one static priority for the thread.

Among the different potential values, we opt for the following: the thread inherits the lower priority amongst the merged threads call sequences. However, in order to keep the system behaviour identical to the previous model, one should allow to increase priority. To tackle this issue, we propose to use a dedicated mutex, configured using the Priority Ceiling Protocol, as a *"priority shifter"*.

Hence, the thread accessing a protected data inherits its priority. For call sequence $i$of priority Pistrictly greater than the thread priority, we create a local variable *priority shifteri*whose priority is Pi, and add an access to i's subprogram calls to the *priority shi f teri*. This provides an efficient way to adapt priority, without impeding determinism. We define the following transformation rules:

Transformation 5.2 (Priority). *The running priority of a thread, result of the merge of two call sequences is their minimum priority:*

$$PT = min(P(CS1)P(CS2))$$

.

Transformation 5.3 (Rewriting call sequence #2). *If the priority of a call sequence CS is higher than the priority of the hosting thread, then we add a priority shifter object to the system. It is a PCP mutex, whose ceiling priority is the priority of CS. The thread will execute the following call sequence:*

$$R2 : CS \rightarrow (PSCSget()CSPSCSrelease())$$

.

Transformation 5.4 (Chaining rules). *In the general case, merging two threads will use these different transformations, following the pattern:*

Input: *Threads* T_1, T_2
Output: *Thread* T_3, *merge of* T_1, T_2
$CS \leftarrow CS_{T_1} \bigcup CS_{T_2}$;
$P \leftarrow min(P_{CS_i} | CS_i \in CS)$ – *Rule 5.2*
foreach $CS_i \in CS$ **do**
  if $P_{CS_i} > P$ **then**
    $CS_i \leftarrow R_2(CS_i)$ – *Rule 5.3*
  end
end
foreach $CS_i \in CS | CS_i periodic$ **do**
  $CS_i \leftarrow R_1(CS_i)$ – *Rule 5.1*
end
$I \leftarrow GCD(CS)$ – *Rule 5.1*
return T_3 = *(CS, I, P)*

When two rules apply, we decide to apply first rule 5.3, then 5.1, so that $CS1$ becomes $R1(R2(CS1))$. The rationale is that we need first to decide whether $CS1$ needs to be executed (rule 5.1), then we adjust its priority (rule 5.3).

When iterating over a complex model, we store annotations in the model to know whether a particular thread is the result of a transformation. This allows us to recognize rewritten call sequences. Should we need to merge threads that are the result of the previous algorithm, we strip annotations, and undo rewriting rules, prior to applying again the whole algorithm. This guarantees optimal computation of the different parameters, at a minimal overhead.

The definition of the Merge and Move transformations are the two basic blocks for optimizing both memory and schedule of a system, we illustrate how to select operations to be performed in the next section.

8. DRIVING OPTIMIZATION

In the previous sections, we defined how to evaluate metrics on a model using REAL, and how to perform atomic transformations that could lead to an optimised systems. In this section, we illustrate how to drive the global optimization process.

To find a fitting solution for a given hardware deployment, it is necessary to find an optimal arrangement of the optimization operations. Since those operations' impact on performances depends on the model they are applied on, it is not feasible to build an a priori arrangement without controlling regularly the actual system value using either model-level or binary-level evaluations.

In the context of Real-Time Embedded Systems, this optimization process is equivalent in finding an optimal binding of all activities (threads, subprograms, etc.) onto execution resources (buses, processors) with limited capacity. Such optimal arrangements problems in finite capacity containers belong to the knapsack class [11]. In the case studied, the effectiveness of the inclusion of a given component into a set depends on the components already present in the set. This belong to the quadratic knapsack family of problems.

Thanks to this observation, we prototyped several solutions to solve this problem with a near-optimal solution. Each solution has been implemented directly in our AADL tool Ocarina. Out of generality, we selected two greedy heuristics. Our objective is to demonstrate the genericity of our approach. Several heuristics have already been defined, their implementation and comparison in the context of RTES is left as future work.

8.1 Fully Greedy Heuristic

Although the knapsack problems, as NP-hard problems, have currently no exact solution of polynomial complexity, one can try to find a near-optimal solution at lower cost, using some heuristics. For the quadratic knapsack problem, an algorithm providing a near-optimal solution in polynomial time is proposed in [10].

We propose in algorithm 2 an adaptation of this algorithm that addresses our actual problem. We use the heuristics found during the optimization operation study: merge only occurs when a candidate set of operation has been selected, and can be bounded by the number of thread components in the system.

Our solution consists in electing a node using evaluation criteria defined in the previous section, and search amongst a list of threads sorted according to the same criterion to found the optimal set of merging that can be done with this thread. The merge is then actually done, then a move operation is tried from another process to the current one (since it just lost at least one thread component due to the merge operation). The algorithm then iterate until their is no more merge possible in the system.

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, since sets to merge tends to count more than two elements, and a cost function in actual implementation usually prevents some obviously non-schedulable combinations to be explored.

8.2 Half Greedy Heuristic

Using the criteria defined in section 3.2, we propose an algorithm that explores 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 3) 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.

These two heuristics have been implemented directly in Ocarina. Since we separate model evaluation from design space exploration, the implementation of the heuristics itself is quite light, and required only a few hundreds line of code to be implemented. Other elements are either supported by REAL, or part of a more generic model transformation engine also integrated to Ocarina.

Hence, the approach we propose is fully generic: one can either define its own metrics using REAL; or implement another design space exploration strategy inside Ocarina. This exploration may be used to favor other metrics such as availability, bandwidth, etc. For instance, in [20], authors illustrate the architecture of a system can be reorganized to minimize the impact of a failure. Such method can be implemented using

```
Input: System S
forall p ∈ Process(S) do
    repeat
        Sort(Threads(p))
        T ← First(Threads(p))
        Candidate_Set ← ∅
        repeat
            Best_Value ← 0  forall t2 ∈ Threads(p) do
                if t2 ≠ T then
                    Current_Value ← Compute_Value(Candidate_Set,t2) if
                    Current_Value > Best_Value then
                        Best_Value ← Current_Value Best_Candidate ← t2
                    end
                end
            end
            Candidate_Set ← Candidate_Set ∪ Best_Candidate
        until noBest_Candidate found ;
        if Candidate_Set ≠ ∅ then
            S ← Merge(Candidate_Set) S ← Move(Candidate_Set,S)
        end
    until Candidate_Set = ∅ ;
end
```

Algorithm 2: Fully Greedy Algorithm.

our approach following the same patterns: define metrics, select an heuristics and apply it to the system under consideration.

In the following, we illustrate the efficiency of our process and its implementation.

9. TEST CASE: THE GENERIC AVIONIC PLATFORM

We selected a case study to assess our solution, based on an abstraction of a complete avionics 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 [15], in the 80s. This model as been designed first to assess suitability of early revisions of the Ada language [16]. 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 5 illustrates its main threads, data fiows and processes, in a representation which only take account of connection existence (multiples connections between two threads are represented by a single connection).

```
Input: System S
forall p ∈ Process(S) do
    repeat
        Best_Set_Value ← 0
        Final_Set ← ∅
        forall T ∈ Threads(p) do
            T ← First(Threads(p))
            Candidate_Set ← ∅
            repeat
                Best_Value ← 0
                forall t2 ∈ Threads(p) do
                    if t2 ≠ T then
                        Current_Value ← Compute_Value(Candidate_Set,t2) if
                        Current_Value > Best_Value then
                            Best_Value ← Current_Value Best_Candidate ← t2
                        end
                    end
                end
                Candidate_Set ← Candidate_Set ∪ Best_Candidate
            until noBest_Candidate found ;
            if Best_Value > Best_Set_Value then
                Best_Set_Value ← Best_Value Final_Set ← Candidate_Set
            end
        end
        if Final_Set ≠ ∅ then
            S ← Merge(Final_Set) S ← Move(Final_Set,S)
        end
    until Final_Set = ∅ ;
end
```

Algorithm 3: Half Greedy Algorithm.

Table 2 Optimization costs and gains.

	FGO-CB	HGO-CB	FGO-PB	HGO-PB
Iterations	536	86	723	101
Duration (s)	4.19	2.63	2.88	3.37
Memory gain	30%	32%	39%	36%

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 3.2:

- 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 1 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. Since no optimization criteria

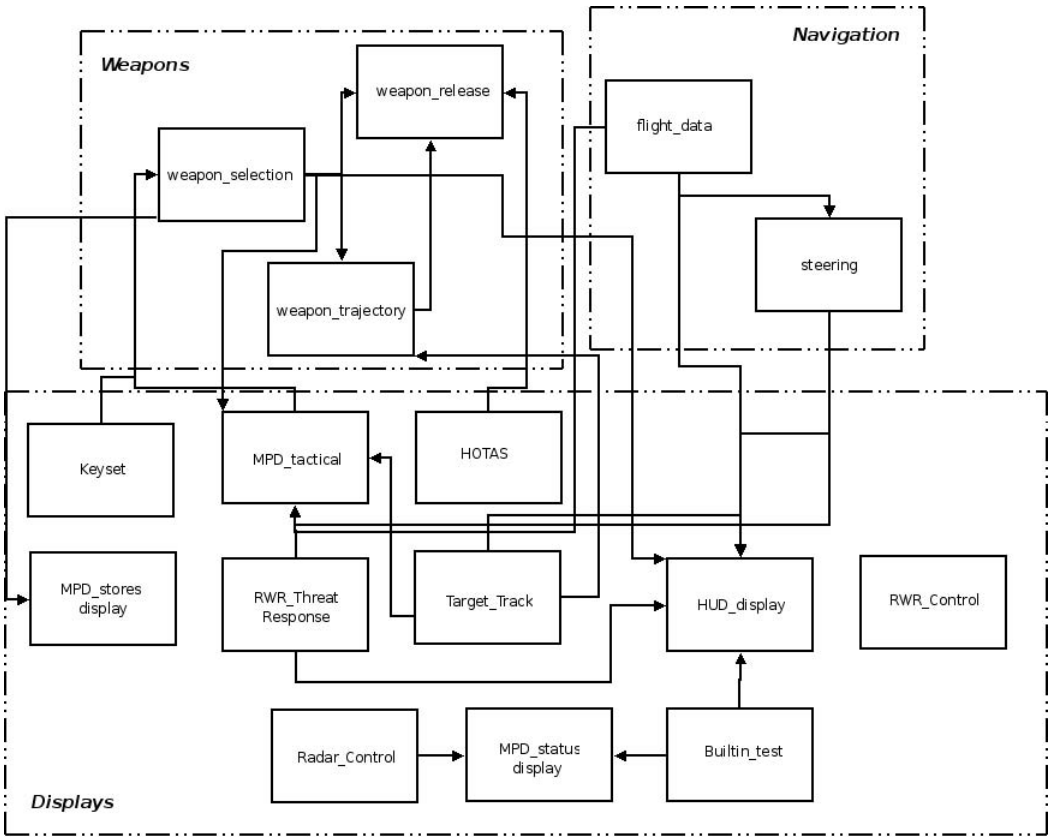

Figure 5 GAP main data flows.

Table 1 Iterations of the GAP platform after different optimisations.

Original	Weapons	Weapon Selection (Sporadic, 200ms) + Weapon Trajectory (Sporadic, 100ms) + Weapon Release (Sporadic, 200ms)
	Display	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, 1000ms) + Keyset (Periodic, 200ms) + HOTAS (Periodic, 40ms) + RWR Threat (Periodic, 100ms) + RWR Control (Sporadic, 400ms)
FGO-CB System	Weapons	(Weapon Selection x Weapon Trajectory x Weapon Release)
	Display	(HUD Display x MPD Tactical) + (Radar Control x Target Track x MPD Status x MPD Stores x Builtin Test x Keyset x HOTAS) + (RWR Threat x RWR Control)
HGO-CB System	Weapons	(RWR 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)
	Display	(HUD Display x HUD Tactical)
FGO-PB System	Weapons	(Weapon Selection x Weapon Trajectory x Weapon Release) + (Target Track x Radar Control x HOTAS)
	Display	(HUD Display x MPD Tactical) + (MPD Status x MPD Stores x Builtin Test x Keyset x RWR Threat x RWR Control)
HGO-PB System	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)
	Display	(Target Track x Radar Control x HOTAS)

changed the content of the Navigation process, we choose not to display them in the results.

9.1 Connection-based optimisations

Table 1 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 2 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 both 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.

9.1.1 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. Finaly, 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 process is already more loaded than the other ones.

- **Weapons:** The same merges are performed than in the fully greedy algorithm. The thread moved randomly is (Radar_Control x Target_Track x MPD_Status x MPD_Stores x Builtin_Test x Keyset x HOTAS)

- **Navigation:** It contains two connected periodic threads of respective periods 80 ms and 59 ms. Since the periods have a GCD of 1, no merge can be performed, and thus no move is actually tried.

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

9.1.2 Half Greedy Algorithm results

- **Displays:** Comparatively to the full-greedy algorithm, the first set to be selected in the test case includes RWR Control but excludes Target Tracking, 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, RWR Threat Response stays as is, suffering of its lack of connections with others 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.

9.2 Deadline-based optimisations

9.2.1 Fully 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. Finaly, 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.

9.2.2 Half Greedy Algorithm results

The same set of threads 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. This difference stems from the variant between algorithms: the Half Greedy algorithm is more aggressive and rejects more configurations that the Full Greedy variant.

9.3 Conclusion

The execution of these two heuristics lead to an overall reduction in memory consumption while preserving initial scheduling constraints. Table 2 lists a gain ranging from 30% to 39% depending on the heuristics and metrics selected. This confirms the initial requirement to have an adaptable process to test different model optimization strategies.

Furthermore, we note that the time to explore the full design space is less than 5 *sec*. This is partly due to the reduced number of admissible permutations in the design: irrelevant designs are discarded when they fail to respect resource constraints. Besides, these metrics are simple computations at model-level, they are quite fast. Object-level performance evaluation are more precise, but are seldom required.

One important aspect of the optimisation process is to ensure it preserves the semantics of the initial system, while enhancing one particular set of metrics. Since we can generate the initial system, and the optimised one, we could compare their behavior on long execution time so as to ensure scheduling orders and timing requirements are preserved.

10. RELATED WORKS

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

In [19], authors discuss optimisation of architecture implemented as Simulink blocks. Yet, this work is restricted to one family of systems, and is fully automated, without control from the user. Furthermore, they only address one dimension: reducing the number of intermediate buffers for transmitting data. Our contribution aims at more genericity by separating evaluation, design space exploration and model transformation. Only model transformation is canonical. Other elements can be adapted by the system designer to finely tune its process.

In [12] authors exploit the nature of data to be managed in transactional memory to decrease communication load, providing a very particular optimization techniques for these systems. Our work aims at generalizing optimization and integrate them in a general framework that mixes evaluations, selection of model transformation and code generation. We applied our work on a generic avionic platform, it can be adapted to other approaches by adapting metrics and heuristics, for instance for embedded, networked or critical systems by adapting metrics and tuning heuristics. Separation of concerns between metrics and heuristics enable efficient implementation of project-specific optimization.

Related to the AADL modeling framework, the ARCHEOPTERIX [9] project use combinatorial exploration of the design, and provide some heuristics to achieve better allocation. However, they rely on extensions of AADL through specific property sets. The designer has to adapt its model prior to optimization. Our contribution rely on pure AADLv2 notations to achieve the same objective.

In [3], 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 [2], 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 a 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.

11. 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 optimization strategies to distributed applications, by taking into account the topology of interconnected nodes, and the communication time in the choice of specific merge or reorganisation of the model. Another extension is to add proper analysis of the message queues and protected components of the system: the designer may have been too pessimistic in dimensioning queue, or in protecting some data. We could take advantage of the description of the whole system to remove useless protections. Finally, a third possible extension is to add more complex analysis frameworks to have a more precise definition of the resource usage.

REFERENCES

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. Krishnakumar Balasubramanian and Douglas C. Schmidt. Physical Assembly Mapper: A Model-driven Optimization Tool for QoS-enabled Component Middleware. In *RTAS'08*, 2008.
3. Dionisio de Niz and Peter H. Feiler. On Resource Allocation in Architectural Models. In *Proceedings of the 11th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'08)*, 2008.
4. B. Dobbing, A. Burns, and T. Vardanega. Guide for the use of the of the Ravenscar Profile in High Integrity Systems. Technical report, 2003.
5. Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, 2006. CMU/SEI-2006-TN-011.
6. O. Gilles and J. Hugues. Validating requirements at model-level. In *Proceedings of the 4th workshop on Model-Oriented Engineering (IDM'08)*, June 2008.
7. Olivier Gilles. REAL User's Guide. Technical report, T'el'ecom Paris, 2009. available at http://aadl.enst.fr/real.html.
8. Olivier Gilles and J'er ome Hugues. Applying WCET analysis at architectural level. In *Worst-Case Execution Time (WCET'08)*, pages 113–122, Prague, Czech Republic, July 2008.
9. Lars Grunske, Aldeida Aleti, Stefan Bjornander, and Indika Meedeniya. ArcheOpterix, An Extendable Tool for Architecture Optimization .
10. B. A. Julstrom. Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem. In *GECCO'05*, 2005.
11. H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springers, 2004.
12. Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luj'an, Andrew Dinn, and Ian Watson. Exploiting object structure in hardware transactional memory. *International Journal of Computer Systems Science and Engineering*, 24(5), 2009.
13. K. Kim and C. Im. Hybrid approaches for derivation of tight service time bounds of distributed emdedded computing systems. *International Journal of Computer Systems Science and Engineering*, 24(1), 2009.
14. Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jerome Hugues. OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In Springer Verlag, editor, *AdaEurope'09*, Brest, France, Jun 2009.
15. C. D. Locke, D. R. Vogel, and J. B. Goodenough. Generic Avionics Software Specification. Technical Report CMU/SEI-90-TR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1990.
16. C. D. Locke, D. R. Vogel, and T. J. Mesler. Predictable Real-time Avionics Design Using Ada tasks and Rendezvou s: A Case Study. In *IRTAW '90: Proceedings of the fourth international workshop on R eal-time Ada issues*, pages 118–125, New York, NY, USA, 1990. ACM Press.
17. Tidorum Ltd. Bound-T Execution Time Analyzer, url: http://www.bound-t.com.
18. Enrico Mezzetti, Niklas Holsti, Antoine Colin, Guillem Bernat, and Tullio Vardanega. Attacking the Sources of Unpredictability in the Instruction Cache Behavior. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, Rennes France, 2008.
19. Marco Di Natale and Valerio Pappalardo. Buffer optimization in multitask implementations of simulink models. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–32, 2008.
20. Sadi M. S., Myers D. G.and Sanchez D., and Jurjens J. Component criticality analysis to minimize soft errors risk. *International Journal of Computer Systems Science and Engineering*, 25(5), 2010.
21. SAE. Architecture Analysis & Design Language V2 (AS5506A), January 2009. available at http: //www.sae.org.
22. SAE/AS2-C. *ARINC653 Annex document for the Architecture Analysis & Design Language v2.0 (AS5506A)*, October 2009 2009.
23. SAE/AS2-C. *Behavioral Annex Language Specification for the Architecture Analysis & Design Language v2.0 (AS5506A) (draft 2.11)*, October 2009 2009.
24. SAE/AS2-C. *Data Modeling Annex document for the Architecture Analysis & Design Language v2.0 (AS5506A)*, October 2009.
25. F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar : a fiexible real time scheduling framework. In *ACM SIGAda Ada Letters*, New York, USA, December 2004. ACM Press.
26. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstr¨ The worst-case execution-time problem om. overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.