Livrable F.1.1.1 : Etat de l’art

Bernard Botella, Fabien Cadoret, Mathieu Clabaut, Frédéric Gilliers, Pascal Poisson, Dumitru Potop, Thomas Robert,

Ce document contient un état de l’art des méthodes d’ingénierie du logiciel pour les systèmes critiques. Deux aspects ont été étudiés : i) les processus de conception, développement et déploiement dirigés par les modèles, ii) les exigences non-fonctionnelles concernant le logiciel embarqué critique et son processus de production, ainsi que l’architecture des supports d’exécution critiques disponibles pour satisfaire ces exigences.

La suite du rapport est rédigée en anglais dans un souci de visibilité du travail fourni.
1 Introduction ........................................................................................................................................1-5

2 Requirement and functional modeling ............................................................................................2-7
  2.1 Requirement engineering ..............................................................................................................2-7
  2.1.1 KAOS method .........................................................................................................................2-7
  2.1.2 WRSPM ................................................................................................................................2-7
  2.2 Formalizing requirements in event-B ..........................................................................................2-8
  2.2.1 KAOS ....................................................................................................................................2-8
  2.2.2 WRSPM ..................................................................................................................................2-8
  2.2.3 UML-B ....................................................................................................................................2-8
  2.2.4 Modal Systems ..........................................................................................................................2-8
  2.2.5 Event-B and Triggers ...............................................................................................................2-8
  2.3 Decomposition of event-B models ..............................................................................................2-9
  2.4 Design patterns and refinement for event-B .................................................................................2-9
  2.5 Conclusion ....................................................................................................................................2-9

3 Software Architecture modelling approaches .................................................................................3-10
  3.1 Software architecture description ...............................................................................................3-10
  3.2 Architecture Description Languages Approaches ......................................................................3-10
  3.2.1 The Lightweight CCM and D&C approach ............................................................................3-11
  3.2.2 The Architecture Analysis and Description Language (AADL) ..............................................3-11
  3.2.3 The MARTE approach ............................................................................................................3-12
  3.2.4 The MyCCM / COAL approach ..............................................................................................3-12
  3.3 Conclusion ....................................................................................................................................3-13

4 Execution platforms: requirements & solutions ..............................................................................4-15
  4.1 Certification requirements: from development process constraints to execution platforms ........4-15
  4.1.1 Risk & impact analysis, and “Assurance profiles” ...................................................................4-15
  4.1.2 Model driven engineering and functional software integration .............................................4-17
  4.1.3 Targets relevant to PARSEC: fault segregation and time critical behaviors ................................4-18
  4.2 System-wide execution platform architectures .........................................................................4-18
  4.2.1 Independent and federated Architectures: properties, assumptions, and limits ......................4-18
  4.2.2 Standardized interfaces for functional software integration ..................................................4-19
  4.3 Software Execution Platform: predictable middleware and virtualization ...............................4-20
  4.4 Discussion and impact on the functional code integration process ............................................4-22

5 Code generation for predictable system deployment .......................................................................5-23
  5.1 Code generation support in Synchronous Language approaches .............................................5-23
  5.1.2 RT-Builder/Signal/Polychrony ...............................................................................................5-24
  5.1.3 AAA/SynDEX ............................................................................................................................5-24
  5.1.4 Comparison of candidate technologies and .........................................................................5-25
  5.2 Code generation support for architectural deployment modeling approaches ..........................5-25
  5.2.1 Code generation from UML timed profiles ............................................................................5-25
  5.2.2 Code generation from Architectural Description Language AADL .....................................5-26

6 Verification methods .........................................................................................................................6-29
  6.1 At functional model level or architectural level description ......................................................6-29
  6.1.1 Verifications on requirement and high-level elements in B .......................................................6-29
  6.1.2 At architectural model to verify deployment and system dimensioning ................................6-29
  6.2 At business source code level .....................................................................................................6-30
  What is a component? ........................................................................................................................6-30
  What is a contract? .................................................................................................................................6-31
How to formalize component contracts? ................................................................. 6-31
How to verify contracts at code level? ................................................................. 6-33

Synthesis and discussion .................................................................................. 7-37

References ......................................................................................................... 8-39
1 Introduction

The purpose of this survey is to present the various alternatives available in terms of models, methods and tools to build a Model Driven Engineering (MDE) framework to produce safety critical embedded software system. Thus, we cover the development process as well as the target architecture available to deploy such pieces of software. We will compare these solutions with respect to their features and how they could be integrated within an end-to-end MDE process from requirement to deployment.

PARSEC project aims at providing a framework to manage safety critical system design, development and deployment accordingly to the most stringent certification processes. In particular, we aim at providing tooling and methods in order to anticipate on DO-178C standard integrating formal proofs as an alternative to exhaustive and systematic system test, or provide hints on how to cope with ISO 26262 and CENELEC 5012X series very strong constraints.

Despite the number of development cycle models, waterfall, V-model, spiral… the kind of tasks to be carried out remain the same and are often described and constrained in certification guidelines [1]. In most cases, there are three main activities within the software production lifecycle: planning of development and verification tasks, development & deployment process, and Verification and Validation (V&V) tasks at system and development process level. In the case of PARSEC, we focus on the development, Verification and validation activities as they are strongly intertwined. As explained in the usual V cycle model, each macro step of the process contains analysis, development and V&V sub-tasks. In practice, these tasks are mixed in order to avoid catastrophic failure of the development process during the integration step at the end of the deployment activity.

We distinguish four steps along the development process:

- Requirement analysis and high level functional design extraction
- Design of system architecture as elementary functionalities with known functional and non-functional properties and expected behavior (including hardware resource description and execution platform properties)
- Verification and Validation tasks required or provided to pass certification process

Model driven engineering approaches try to fill the gap between accurate System Requirement definitions and potential reuse of software building block from previous development processes [2]. In order to define and manage these building blocks Component Based Software Engineering approaches have proven their advantages by providing modular, and well defined software units. CBSE has also proven it efficiency when dealing with deployment tasks. By deployment, we also consider the integration of the software component on the execution platform. In this context, components represent software or hardware units that can be manipulated and composed by the deployment process (usually to match non functional requirements). The conditions under which such component can be reused are particularly complex to handle in the case of safety critical systems.

In the field of embedded software, dedicated models are used to solve each specific task encountered during the development process: requirement extraction and analysis, functional CBSE design to encourage piece of software reuse, and aggregated functional and deployment model to describe how the functional description is or should be mapped on an actual system. One of the main issues encountered during CBSE end-to-end development processes lies in the transition from the functional point of view to the deployment issues.

In industry the consistency in the development process between requirements and the actual implementation is ensured by following guidelines specified in certification standards. These
guidelines are defined as requirement on the system properties and also on its development life-cycle. Among possible approaches, Model refinement and transformation is one of the core concepts within MDE approach providing insights on how to handle the development process. It appeared as one of the key mechanisms to manage a development process using several specific models. The goal is to provide transformation rules to translate the semantics of models defined according to a formalism F in a formalism F’ (at least as expressive as F). In theory, model refinement steps should be performed to deduce a model of the system implementation from its functional requirements. The implementation model should be sufficiently detailed to allow automating the integration of the functional code on the execution platform.

We will follow these lines in PARSEC. In this survey, we will present both aspects: the models and methods available to describe the system along all the steps of the development process, and a brief summary of the typical requirements to fulfill when dealing with certification tasks.

The survey is organized according to the type of modeling activity required in such a framework. We also include sections on code generation for integration of functional software on execution platforms and software execution platform in-use for safety critical embedded software systems.

1. Requirement and Function modeling
2. Component Based description integrating functional and non functional requirements
3. Certification constraints and execution platform description to guide, prove, or specify the actual system deployment
4. Code generation support for integration of predictable software on execution platforms
5. Verification methods at model and legacy code level

For each of the fourth first steps, we will identify the properties ensured by using the various methods presented along this survey.
2  Requirement and functional modeling

This section provides an overview of the existing techniques and languages dedicated to requirement and functional modeling. We focus our interest on formal or semi-formal methods suitable for safety related system modeling.

Previous experiences show us that refinement is a key feature to allow industrial size formal modeling and reasoning. Being one of the only modern formal methods offering such a feature, we will mainly focus our interest on event-B and related subjects.

Event-B is a formal method for system-level modeling and analysis. Beyond the use of refinement to represent systems at different abstraction levels, key features of Event-B are the use of set theory as a modeling notation and the use of mathematical proof to verify consistency between refinement levels.

2.1  Requirement engineering

2.1.1  KAOS method

The KAOS method [3],[4] (Keep All Objectives Satisfied) proposes a semi-formal methodology for goal oriented requirement engineering. The KAOS approach allows identifying and structuring the requirements as objectives the finish product must meet. For this purpose, four types of models are built successively to decompose and structure the requirements.

- **Goal model**: which is the basic model of the KAOS approach. It defines the objectives and the sub-goals necessary and sufficient for the objective to be achieved.

- **Responsibility model**: the responsibility model allows to rely on several agents of the system for being responsible of achieving different goals. Agents may be of all kind, from people to software components.

- **Inter-agent interfaces model**: this model defines the relationships between agents – which agents are observed or controlled by which other agent.

- **Operation model**: this model defines the behaviour of agents so that they can achieve their goals.

Those different models allow therefore to define the expected requirements and ensure their refinement during the various stages of system design.

2.1.2  WRSPM

The WRSPM approach WRSPM (World, Requirements, Specification, Program, Machine) allows the formalization of system requirements. In this approach one may differentiate artifacts from phenomena. Phenomena represent the state space and transitions of the system and the environment. The artifacts represent the stresses on states and transitions. The artifacts are set into groups as follows:

- **World**: description of the environment such as it is supposed to behave,

- **Requirements**: description of the desired behavior of the environment.

- **Specifications**: relationship between the requirements and the system.

- **Program**: software.
• **Machine**: execution support of the program.

### 2.2 Formalizing requirements in event-B

#### 2.2.1 KAOS

Work is in progress [13] to propose a formalization of the KAOS model, not only in terms of operational objectives (requirements) as has been proposed in [14] and [15] but also in non-operational objectives, taking into account the different types of refinement relations offered by KAOS.

#### 2.2.2 WRSPM

[7] proposes a method to maintain traceability between requirements, and the event-B model. The method based on the successive application of translation rules from the WRSPM partition to an event-B model also allows the management of changing requirements. A tool is being developed under the Rodin platform for this method [16].

#### 2.2.3 UML-B

The emergence of the UML as a de-facto standard for object-oriented modeling has been mirrored by the success of the B method as a practically useful formal modelling technique. The two notations have much to offer each other. The UML provides an accessible visualisation of models facilitating communication of ideas but lacks formal precise semantics. B, on the other hand, has the precision to support animation and rigorous verification but requires significant effort in training to overcome the mathematical barrier that many practitioners perceive.

A derivation of the B notation is used as an action and constraint language for the UML and defines the semantics of UML entities via a translation into B. A UML-B profile provides specialisations of UML entities to support model refinement. The result is a formally precise variant of UML that can be used for refinement based, object-oriented behavioural modelling [5].

#### 2.2.4 Modal Systems

Modal systems are used to model the behavior of a system in different modes. This type of system allows to represent on the same model different behaviors depending on the current mode and on allowed transitions between different modes (normal, degraded, ...). Modal systems and event-B [R4] are close, and it is possible to show that an event-B model is the refinement of a modal system. However, [6] does not offer constructive method for obtaining such a refinement of a modal system.

[18] reports on the implementation for the space domain of a method for modelling modal systems based on patterns and composition in event-B. This experiment is a first step towards the establishment of a constructive method for modelling modal systems, which is adapted to the formal development of complex systems by emphasizing reuse and composition.

Work was also carried out [22], [23] to specify a framework for modelling patterns in event-B. A plug-in has been developed to facilitate the modelling and refinement of modes.

#### 2.2.5 Event-B and Triggers

Some requirements (particularly that of availability) can not be expressed directly using the first-order logic. Timed linear logic (LTL) is then a classic choice for expressing such properties. [8] proposes the use of "trigger" to model such properties. A trigger is an event that must occur in a given period of time if its guard is satisfied during this period. [8] proposes some "patterns" for translating temporal requirements in B-event using triggers.

It should be noted that there is also a model checker property expressed in LTL available on the
platform Rodin (ProB).

2.3 **Decomposition of event-B models**

The decomposition of an event-B model aims to simplify the proof obligations (PO) and thus alleviate the proof task. The decomposition helps to redistribute the PO needed for successive refinements of a model to sub-components. Moreover, by construction, decomposition ensures that the system still meets the properties satisfied by the event-B model before decomposition.

The decomposition model can be considered according three different ways:

1. identification and definition of a general theory (for examples of stacks and queues) which theorems or axioms once defined can be reused in the main development. This approach is implemented by Rodin in the use of mathematical theories [20].

2. cutting a system into parts on which development can be pursued independently. This decomposition is based on the monotonicity of the refinement in Event-B. There is a Rodin plugin [9] for decomposing a model along two different routes: decomposition by shared variables, or decomposition by shared events. Decomposition by shared events is more appropriate to systems communicating by messages.

A decomposition method with shared variable proposed in [10] is based on a partition of the events of the model to decompose. The partition of events on the sub-component allows to define a distribution of variables between the different models. Some variables can be used by various sub-components, in this case they are "shared" and can not be refined further. In addition, “external events” are introduced to manage the behavior of shared variables.

3. hierarchical structuring of a system where part of the system is encapsulated in a self-supporting module within a modelling unit. These mechanisms are proposed in [17] and [19].

2.4 **Design patterns and refinement for event-B**

“Design pattern” is a design concept whose objective is the reuse of known solutions to solve traditional problems. The decomposition in event-B can also identify patterns [12]. The proposed approach relies on the reuse of a model and its refinement. The advantage of such an approach is also to offer proven solutions and promote the reuse of a model and its refinement. Moreover it allows factoring the proof effort by discharging the proof obligations during the pattern design.

A Rodin plugin to use design patterns for event-b is under development [12].

The concept of pattern refinement proposed in [21] is based on the idea of refinement rules. A refinement pattern is actually a transformer that allows models to factor in some measure the experience of refinement of models in the context of these patterns.

2.5 **Conclusion**

This section has provided an overview of the methods used for semi-formal or formal modelling either at the requirement level or at the functional level. It shows that if some tools and methods are available as a basis for system modelling, there is still a lot to do to assemble all the pieces and to define a methodology applicable to safety related, on-board and distributed software systems.
3 **Software Architecture modelling approaches**

This section is dedicated to modelling software architecture. We first succinctly define software architecture, then we study the related technologies that are related to Thales technologies in software architecture domain for embedded systems.

### 3.1 **Software architecture description**

Software architecture modeling is defined in [27] “as the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them”. There are many definitions of what the software elements are. Component-oriented methods for describing software architecture build on this generic approach and propose to describe software architecture as the composition of software components each representing reusable constituting entities of the system. As stated in [26], most software component models share at least the following properties:

- isolation of the interface vs the implementation
- ability for a component to implement several interfaces.

For example, the UML language defines components as “Components: A component is a self-contained unit that encapsulates the state and behavior of a number of classifiers. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces ». Such view leads to considering that the architecture of a software is described as the assembly of its constituting components and the topology of their communications.

The definition of the component’s interface (or contract) then becomes the key aspect of the component-based approach. Most component-based approaches focus on the syntax coherence: the contract validation is limited to the syntactic description (name of the service, types of the parameter and return values) of the services provided or required by the component. There are two other main categories of component contracts:

- Semantic description: the component describes the functional behaviour of the provided / required services.
- Non-functional description: the component specifies the execution requirements (worst case execution time, required priorities, etc…)

Depending on the application domain, there are several means to describe advanced component contracts, in the field of information systems, ontologies are used to identify services according to a common semantic base [28][29]. Regardless of the underlying technology, the precision of the components’ contracts definition is the main factor that allows verification of the expected properties of the software from architectural descriptions.

Specification languages dedicated to architecture description are called Architecture Description Languages.

### 3.2 **Architecture Description Languages Approaches**

This section provides an overview of the existing techniques and languages dedicated to modeling software architectures. This section is restricted to languages adapted to modeling critical systems.

The aim of Architecture Description Languages is to provide a structured representation of the software system or application. They are mainly based on the following concepts:
• **component**: the basic building block of the architecture description,

• **connector**: defining the interaction semantics between components,

• **configuration**: which defines the connection between the component instances and connectors, and the initial value of the configuration attributes.

One key aspect of ADLs is that semantics is associated with the language constructions (i.e. components, configuration and connectors): these are not just boxes and lines. There are several ADLs, depending on the target application domain. This section will provide a short overview of the capabilities of the following ADLs that have either inspired or are part of the MyCCM approach:

• AADL (Architecture Analysis and Description Language),

• MARTE,

• Lightweight CCM and D&C.

We then describe more precisely the approach proposed by Thales in the MyCCM-HI framework, since this approach will be further developed in the course of the PARSEC project.

### 3.2.1 The Lightweight CCM and D&C approach

CCM (CORBA Component Model) [30] is the original approach for component modeling provided by the OMG. Lightweight CCM describes the component model, and D&C [31] describes the deployment and configuration of the components constituting the system. Together they define an ADL. Components are of course the Lightweight CCM components, and D&C defines component assemblies, their connections and the required information for deployment.

Lightweight CCM is now a profile of CCM targeting embedded systems. It provides a simple, yet powerful component model that mainly lacks extensibility features (the component model has a defined set of abstractions, and no extension mechanism is provided). Extensibility has been added in a set of companion specifications such as QoS4CCM and DDS4CCM by adding an additional layer to the original component model.

The concept of the approach is to split the software system into components defining provided and required services and a set of configuration attributes. The component itself is seen as a black box from the rest of the system. Using this approach, the software is viewed as an assembly of component, and the associated configuration. The assembly and configuration capability is provided by D&C.

The main drawback is that these standards are not targeted at real-time systems. For example, D&C does not provide a simple means to configure the number of threads of the system (or their real-time configuration), and the deployment mechanisms are virtually impossible to implement without extensively using memory allocation, dynamic code loading, and other constructions that are to be avoided for the development of critical systems.

### 3.2.2 The Architecture Analysis and Description Language (AADL)

AADL [32] is an architecture language dedicated to real-time critical distributed systems. Its modeling capabilities range from high-level description of the system (thus close to some aspects of SysML) to the low-level description of software applications and their hardware execution environment (thus including threads, processors, buses, memory allocation, etc.).

In this context, the concept of **component** applies to any entity type defined by the language describing a part of the system. For example, any of the following elements, **processor**, **thread**, a **subprogram** or even a communication bus are **components** of the system. If the exact nature of a component cannot be known at a given abstraction level, the language provides the notion of **abstract component**, whose kind can be defined when the architecture is further refined. Although the notion of
connector is not explicit in the language, AADL provides the notion of **ports**, which are associated with components and allow connections between them. Ports and connections correspond to the notion of connector. Another notable feature of AADL is the management of explicit operational modes. This aims at defining the impact of a predefined set of operational modes on the configuration. For example, a connection between two components of the system can be active only in a given mode and inactive in all other operational modes of the global system.

Functional components, which represent a reusable unit of code (set of functions and associated data structure), are not covered by AADL; therefore, AADL is not suitable for the description of component-oriented software engineering, as it is a lower-level description language. This aspect of the modeling process must be covered by other languages. The implementation of functional components must then be integrated within AADL threads. AADL threads then carry communication ports that correspond to the interfaces of the functional components. Thus, the functional code (i.e. the implementation code of the functional components) is tightly bound to the execution resources (i.e. the AADL threads). This leads to the main drawback of AADL which can be employed to define very abstract specification (SysML like) using only abstract component, or very low-level and precise representations of the system, very suitable for code generation but far too complex and detailed and complex for software architecture design.

AADL comes with an extensive set of extensions allowing to model specific capabilities of a system. Two are of main interest in our context:

- The behavior annex allows the definition of the behavior of software entities such as subprograms and threads;
- The ARINC 653 annex provides a mapping from an AADL specification to a partitioned operating system.

Both these extensions will be extensively surveyed in the course of the PARSEC project in order to take advantage of the corresponding features.

### 3.2.3 The MARTE approach

MARTE [38] is a profile for the UML2 language dedicated to the modeling of real-time embedded systems, including non-functional properties and stereotypes for software and hardware elements. An interesting feature of the MARTE approach is that, as a UML profile, it can be combined with other existing OMG standards for the component-based approach (e.g. Lightweight CCM)—the component model of MARTE itself relies on UML2. MARTE inherits the UML2 ability to describe dynamic behavior, using for example state charts. This means that MARTE provides an interesting approach to describe both the architecture of the application and (part of) its dynamic behavior in a single modeling tool. MARTE also provides ways to model operational modes of the system and their impact on the architecture, inspired from AADL.

Many concepts in MARTE are also addressed by AADL, as both languages aim at describing the same kind of architectures. A specific annex of the MARTE standard describes how to use it to model AADL architectures.

The drawback is that MARTE is a language meant to address all aspects of real-time and embedded modeling. As such, MARTE is difficult to apprehend; it is necessary to identify which subset of the MARTE elements must be used for a given architecture model. This is one of the aims of the VERDE project where Thales will provide integration between MyCCM-HI and MARTE.

### 3.2.4 The MyCCM / COAL approach

Thales needs an architecture language that provides the capability to define the architecture of distributed real-time embedded systems with the same level of precision than a concrete AADL
description while preserving a more abstract *software oriented* view of the system. Another important aspect of many systems is their capability to adapt their behaviour to modifications of their operational environment.

This is why Thales decided to create a Component-Oriented Architecture Language (namely COAL [64]) that enables to benefit from added values of architecture description languages and component-based software engineering. COAL draws upon the Lightweight CCM standard for software component description D&C for deployment description and AADL for system configuration. COAL adds support for the description of the system reconfiguration using a set of communicating automata that allow describing the evolution of the architecture in an evolving operational environment.

COAL in itself does not define an abstract component model, instead it reuses the existing Lightweight CCM abstract component model and provides deployment and configuration capabilities adapted to real-time embedded systems allowing to precisely defining:

- thread configuration (including priority, periodicity, stack size);
- application mode management;
- component localization (hardware node);
- deployment inside the localization (separation in processes);
- connections between component ports.

The language also adds the capability to define the system’s operational modes and the dynamics of the transition between these modes using communicating automata. Schematically, the automata allow to define the evolutions of the system’s architecture when the execution mode changes. The reconfiguration mechanism can be executed *without memory allocation* which means that all the resources required for execution modes have to be statically allocated at system start-up.

The aim of the (Lightweight) CCM component model is to enforce separation of concerns between the *business part* of the application (namely the algorithmic part of the application), and the so-called *non-functional part* of the application, which roughly comprises real-time configuration such as thread management, and elements related to system deployment (distribution). The idea is that developers should focus on the business part of the application, i.e. write test optimize the algorithms that create the actual value of the system whereas the implementation of the configuration required to enable these algorithms to perform their task properly should be as automated as possible.

Thales created a component framework to implement these principles: MyCCM-HI. The MyCCM-HI framework generates the code required to implement the architecture of the application including deployment code to initialize component instances, connection of the component ports, data serialization, and real-time configuration. In order to comply with coding standards and to avoid excessive memory usage, only static code is generated from the architectural description. No interpretation is done at runtime which means that it is not possible to modify deployment information without modifying the generated code (which may require a full rebuild of the generated code).

Thus, modes automata, as well as modes switch impact are modelled in COAL so as to reduce the integration and validation time – the reconfiguration mechanisms are generated by MyCCM-HI so as to ease integration, and the model provided is analyzable so as to ease the validation process – while benefiting from advantages of component-based approaches [61]. This approach is more precisely defined and illustrated by an example in [63] and [64] (in French).

### 3.3 Conclusion

This section has provided an overview of the industrially applicable methods for software architecture description that are compatible with the approach currently developed by Thales. Our conclusion is
that no current available tool meets the requirements of complex real-time software development. Therefore, Thales developed a component-based framework (MyCCM) to address this issue. This framework draws on several existing tools in order to provide both the precision of the AADL language and the abstraction level of component-based approaches such as Lightweight CCM. The resulting framework provides a high-level approach enforcing separation of concerns, while preserving the required properties to allow effective code generation for the low-level technical code.
4 Execution platforms: requirements & solutions

In this section, hardware and software components involved in execution platforms related to PARSEC will be presented. In previous sections, modeling approach adapted to requirement analysis and system implementation specifications have been compared. The aim of this section is two-fold. First, it details constraints on both system development process and architecture imposed by certification processes. Then, it points out the relation between certification constraints and hardware and software architectures used in safety critical software development. There is a particular focus on the software execution platform layer, its structure and its features.

4.1 Certification requirements: from development process constraints to execution platforms

Certifications methods usually rely first on a set of recommendations about how to assess system dependability or security attributes. Then, these recommendations are refined into accurate requirements concerning the engineering process like qualitative or quantitative dependability attributes (mean time to failure, probability of failure on demand per hour…). These requirements may directly force to apply a given method. In avionics ARP 4754 plays this role detailing how system safety should be assessed and what kind of safety should be required on the different parts of the system (flight control system, cabin entertainment systems, track monitoring systems…) mainly depending on the impact of system failures. In railways and automotive systems, high-level requirements are specified respectively in IEC 61508 and ISO 26262.

In any case, the whole engineering process of safety critical systems will be impacted by such requirements and guidelines in order to enable system certification. In addition to the high level requirements, specific requirement are defined for several type of system subsystem: network, hardware, power supply, software… In the context of PARSEC, we are mainly concerned with hardware computing nodes and software related documents. In avionic systems, such recommendations can mainly be found in DO-178 and DO-254 series. In railway domain, one can find most of these information in CENELEC EN 50128, EN 50129 recommendations.

4.1.1 Risk & impact analysis, and “Assurance profiles”.

Dependability requirements rely on sets of pre-defined requirements for system architecture and development process taking into account system failure severity, e.g. system loss cost or impact on its environment due to its failure (including human losses). These requirements correspond mainly to prevent faults made during the software development process to alter drastically system dependability. Thus, it identifies organizational and methodological solutions to avoid these faults (requiring for instance independence between development and assessment, or requiring to apply validate task such as testing according to well defined criteria, e.g. MC/DC criteria standing for modified condition – decision criteria in avionics).

Let call these predefined sets of requirements assurance profile (as it is done in avionics). They are defined in order to obtain known or controlled risks of failures or at least prevent well-known threats to dependability. These profiles are assigned in three steps: first perform risk and impact analysis for software given know hardware reliability to determine potential failure occurrence rates, then define the expected failure occurrence rates (drawn from regulations or certification standards), eventually select the profiles providing sufficient confidence in system dependability depending on the initial risk analysis. Profiles are ordered because they correspond to distinct levels of criticality of the software system.

In this context, Safety Integrity Levels embody levels of criticality for Railway systems, and are selected according to probability bounds for failure occurrence rates computed depending on the way the software is used (continuously or sporadically). Here, safety has to be understood in terms of
dependability as stated in [ALR2004] covering maintenance, integrity, reliability, availability and let say safety-innocuity. In avionics, criticality levels are selected according to software failure impact on the aircraft or its passengers. In both fields, there are five levels but they do correspond to distinct concrete requirements (e.g. different requirement profiles) on the development process.

Because profiles are related to failure impact severity, they are also related to some extent to quantitative constraints on failure occurrence within the whole system (including the execution support). One has to distinguish this kind of constraints from the requirement concerning the way the development process is conducted. They affect the type of execution platform that can be used to deploy the safety critical software. Notice that the quantitative requirement differs depending on the way the software is used, and on its role.

<table>
<thead>
<tr>
<th>SIL</th>
<th>Low demand mode of operation (Average probability of failure to perform its design function on demand PFD)</th>
<th>High demand or continuous mode of operation (Probability of a dangerous failure per hour PFH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10-2 &lt; PFD &lt; 10-1</td>
<td>10-6 &lt; PFH &lt; 10-5</td>
</tr>
<tr>
<td>2</td>
<td>10-3 &lt; PFD &lt; 10-2</td>
<td>10-7 &lt; PFH &lt; 10-6</td>
</tr>
<tr>
<td>3</td>
<td>10-4 &lt; PFD &lt; 10-3</td>
<td>10-8 &lt; PFH &lt; 10-7</td>
</tr>
<tr>
<td>4</td>
<td>10-5 &lt; PFD &lt; 10-4</td>
<td>10-9 &lt; PFH &lt; 10-8</td>
</tr>
</tbody>
</table>

Figure 1. Qualitative constraints associated to SILs depending on usage

In most cases, the whole assurance profiles lead to consider a mix of qualitative and quantitative constraints. For instance, the Railways European Standard has identified techniques and measures for the five levels of software safety integrity in addition to numeric constraints on failure rates:

1. Identify hazards, assessing risks and arriving at decisions based on risk criteria,
2. Identify the necessary risk reduction to meet the risk acceptance criteria,
3. Define an overall System Safety Requirements Specification for the safeguards necessary to achieve the required risk reduction,
4. Select a suitable system architecture,
5. Plan, monitor and control the technical and managerial activities necessary to translate the System Safety Requirements Specification into a Safety-Related System of validated safety integrity.

These constraints are applicable to any subsystem involved in safety critical systems used in railway control and protection systems, including application programming, operating systems, support tools, firmware and hardware platforms. Application programming comprises high-level programming, low level programming and special purpose programming (for example: Programmable logic controller ladder logic). These kind of global point of view for managing risk and dependability can be found in almost all the equivalent standard of transportation systems (avionics with DO-178 series, and automotive system with ISO 26262).

Suppliers of systems have to demonstrate in reproducible way that the specified techniques and

1 Risk reduction correspond to the inverse of the failure probability …
measures are fulfilled, by producing all the required evidences. This set of data is submitted to an independent assessor that either assesses the evaluation process or re-assess so features of the system. The assessor validate that the supplier’s assumptions, technical choices, processes, documentation, hardware and software components meet the specified requirements. The assessor forms a judgment as to whether the system is fit for intended purpose.

In most case, the compliance to quantitative constraints and some particular requirements on software integration strategy highly constraint the way these pieces of software can be actually deployed. More other, hardware specific requirement are defined to master hardware non functional properties. Remember that the execution platform and especially hardware components provide quantitative indicators for failure occurrence rates.

4.1.2 Model driven engineering and functional software integration

A large part of constraints defined in certification documents concerns requirement on how to manage requirements, implementation specification, development activity schedule, and system integration. Model driven engineering approaches offers design, implementation and integration methods adapted to cope with certification constraints. Three points need to be highlighted:

- Document accuracy to manage the transition from requirements to system implementation specification (at high or detailed level),
- Configuration data management (generation, verification …),
- Verification-based and correct by construction integration strategies.

**Document production and assessment:** one of the main interests of MDE comes from the availability of automatic methods to detect or avoid consistency, exhaustiveness or even sounded issues. As they are most of the time supported by tools, they provide usual versioning systems to tag and identify documents (e.g. models). It can be used to perform preliminary safety analysis (out of the scope of PARSEC) or ensure, given such analyses, that required safety properties are actually enforced in operation (which is the core of PARSEC project).

**Configuration data management:** [66] identifies two major usages for these data, functionality tuning or initialization, and safety mechanisms or execution platform service configuration. Whereas the first aspect highly depends on the type of functionality considered, the configuration of execution platform is one of the most important steps of the process as it is recognized as an important source of software faults during the integration of functional code on execution platforms. One of the answers of MDE approach to avoid development or integration faults in configuration data is to directly infer the integration code or strategy (glue) from models and expert knowledge of standardized execution platforms. It provides the ground for safe and correct system-wide integration strategies.

**Verification-based and correct by construction integration strategies:** one of the first benefits of MDE is to integrate automatic verification capabilities as early as possible within the design process. With such synthesis task, MDE can provide two complementary features: verification and synthesis of models to ensure correct software system design and deployment. The synthesis process is more concerned with generating models of execution, architecture of configuration with proven pre-defined properties. These properties are ensured by the generation process, and thus said “correct by construction”.

Whatever synthesis or verification is used to ensure the correctness of the functional code integration on execution platforms, in-deep knowledge of execution platforms is required in both cases to achieve the same confidence level in the system predictability. As pointed out before, the availability of standardized programming interface of the software layer provided by execution platforms is a major progress in automating the integration of reusable functional software elements.
4.1.3 Targets relevant to PARSEC: fault segregation and time critical behaviors

In section 2, various high level requirement modeling and refinement approaches provided some insight on how to obtain a modular high level description of system functions. The previous sections present various modeling framework to describe these functions as software components enhanced with non-functional attributes. Non-functional aspects are mainly related to either timing constraints or software component binding to hardware. The latter is impacted by dependability considerations and timing aspects at once. This bidding has to ensure that timing constraints will be met and that the bidding implements the desired failure confinement areas.

Software failure containment can be achieved either through software or hardware mechanisms. For decades, hardware fault tolerance provided efficient but expensive solutions to software failure containment (mainly through modular redundancy). Such approaches lead to limit the number and type of functionalities integrated on a single computing node. Thus, several design patterns were considered to define reusable fault tolerance mechanisms. These services are mainly provided either by the hardware or low-level software layer (e.g. they are provided mainly by the execution platform).

Even if no redundancy is available, separating functionalities on various pieces of hardware provide for physical containment area for most of software failures. This solution has been privileged for decades in avionics (with line replaceable units), and automotive (with electronic control units). Recent progress in

4.2 System-wide execution platform architectures

This subsection describes architectural templates used in safety critical systems according to the well-known design patterns (especially avionic ones). In these templates, the architecture is mainly organized to provide neat failure isolation between various functions. In 80’s and before, only hardware-based isolation provided enough evidence to cope with certification requirements. Thus, the software deployed on such system was simply concerned with functional and timing predictability. Thus, real time executive or operating systems were deployed upon independent or federated avionic architectures. The goal oriented approach followed to design and implement software had been slightly biased by the a priori that no particular isolation has to be provided by software components. Such isolation properties were the main targeted of DO-254 series. This a priori is no longer valid.

The choice of the hardware component provide in most cases the numeric evidences on system failure rates. In order to achieve risk reduction ratio up to $10^4$, hardware fault tolerance mechanisms are directly integrated to hardware architectures. Nevertheless, there is currently a push to move from independent electronic equipments to complex architectures connecting computing node hosting several functionalities at once [68]. Expected benefits from of the

4.2.1 Independent and federated Architectures: properties, assumptions, and limits

In the early 60’s, the so-called independent architecture was ensuring that equipments were fully separated. It was the mean to ensure that equipments at distinct level of criticality will not interfere. The main drawback of this approach was the duplication of electronic or display devices within the system. The shift to computing nodes connected through networks led to a more integrated architecture providing means to enable communications between equipments: the federated architecture. The idea behind the federated architecture is to keep the physical separation of function in distinct hardware equipment. Thus, if there is a function that has failed permanently, the module can be disconnected and withdrawn from the system. In this context, the interferences between modules were mastered as they only share the network a priori. One of the big advantages of this approach was that these modules called line replaceable unit in avionics could be developed rather independently or even as COTS from selected sub-contractors. One of the issues of such system lies in the need for more and more bandwidth for network communications to integrate these modules. as the number of node increase.
The second issue is the relative cost for unit integration to build the “federation”. [68] points out that the current development processes in avionics do not provide sufficiently automated procedure to make it tractable on the long term. Thus, either there is change in the architectural pattern used to deploy such systems or the development process is improved providing automated design steps with respect to function integration and deployment.

4.2.2 Standardized interfaces for functional software integration

As stated before, the integration process of functional code on execution platform is a major source of development faults. The use of standardized programming interfaces validated by user communities and/or certification bodies facilitated the use of software execution platform off-the-shelf like OSEK-VDX compliant kernel or ARINC 653 compliant operating systems.

There has been a push in avionics and other domains like automotive to consider that many functions may be deployed on a same piece of hardware, even if they do not belong exactly to the same assurance profile. Concurrently, industries of these domains decided to agree on a standardized interface to allow generic integration of functional software on execution platforms. The most famous of these standards are the ARINC 653 [70] and AUTOSAR standards [71]. This need come from practical considerations in terms of performance and implementation cost of federated architectures, and the hazards represented by heterogeneous integration interfaces. For instance, more sophisticated cars contained more than 50 interconnected electrical units with different processors through a field bus (e.g. CAN), it turned out to be nearly intractable to verify such architectures with heterogeneous software structure. The core feature of such standards is that they define interfaces and execution platform expected behavior but not how to implement it nor how to integrate the functional code on it.

Consider the ARINC 653 case, this standard defines an execution platform supposed to provide for isolation of independent software modules as defined in IMA design guidelines (remember that failure containment was the core property of federated architecture). In avionics, the Integrated Modular Avionics (IMA) standard provides insights on how to obtain a software architecture relying on independent software modules deployed on a same hardware node. ARINC 653 defines the interface of an execution platform allowing such architectures. IMA guidelines provide the following architectural pattern:

- Functionalities are implemented by software modules deployed within pods called partition providing software failure containment through time and space isolation.
- A runtime support is provided to each hardware node to support partitions and provide communication between partition inside and outside the node.
- Software module deployed on a given node may not share the same criticality level but the gap is limited in practice as in DO 178B do not provide any means to certify system with any kind of mixed criticality (there are groups allowed combinations (A-B), C, (D-E)).
These rules illustrate the shift from failure containment through hardware isolation to software isolation. The direct consequence of such an approach is that the software runtime support has now a key role in failure containment. Thus, it has to be certificated to the highest level of the assurance profiles depending on the domain (SIL 4 in railways applications, and DAL A for avionics).

The main issue while implementing these independent modules is to understand how they will be integrated with the runtime support, and how the application non-functional features need to be configured and integrated upon such software layers. The ARINC 653 architecture boils to consider two layers: the AP-EX and the CO-EX (respectively standing for Application executive and core executive). The principle is to distinguish the interface to integrate application software on the module (APEX), from the actual implementation of module runtime service (scheduler, communication, error handling) with time and space isolation between partitions (COEX). Nevertheless, as stated before no clear guidelines are provided on how the services provided in these layers should be validated or implemented. Thus, we propose to investigate available architecture and assume that architectural verification strategy or correct by construction design could be applied to validate run-time supports.

4.3 Software Execution Platform: predictable middleware and virtualization

As said in subsection 4.1, one of the key concern regarding the Software Execution Platform (SEP) is related to its certification. In this subsection, several real-time run-time operating system will be briefly described according to three criteria: i) compliance to some certification standards, ii) kind of support for real time executions, iii) support for software failure containment. We do not have the pretention to provide an exhaustive list of real time operating systems, yet we tried to quote the most relevant one in the field of avionics, automotive …

Internal architecture will not be discussed exhaustively here. For detailed survey of virtualization solution and partitioned systems, see [67]. We will briefly recall different possible architectures providing hierarchical execution models with failure containment services and real-time scheduling features. Hierarchical scheduling is in charge of managing partition and inside-partition activities schedules. Two architectures have to be pointed out: isolators and virtual machine monitors.

**Isolators** architectures implement the hierarchical scheduling policy are a single service managing at the same time both partition and partition task schedules. The main benefit of isolators is to concentrate scheduling services in a single component. Thus, the whole scheduling policy can be easily mapped from a hierarchical system-wide schedule to execution platform configuration files. Performances are also better as the execution platforms are seen as whole trusted area. Thus, no specific additional segmentation is needed on the kernel state, decreasing the number of reconfiguration of the access control system (through MMU). One of the drawbacks comes from the validation of such implementation. Either, the system integrates all possible services and become very difficult to validate, or it encompasses required services only and need some reliable configuration strategy ensuring correction execution platform configuration.

**Virtual Machine Monitors** provide in each partition the capability to host a complete operating system (that we will can guest OS). In practice, this capability is ensured providing either full-hardware virtualization or para-virtualization. In both case, the purpose is to separate the tasking model within partitions from the scheduling model applied among partitions. Most efficient solutions execute the partition manager directly on the hardware without any intermediate software layer (called hypervisor, or virtual machine monitor). Notice that, depending on the expected tradeoff between interoperability and performances, the hypervisor and the hosted operating system may require some co-design for performance purposes, or provide a complete abstraction of the system hardware in each partition. Monitor and guest operating system co-design is used to avoid creating a complete abstraction of the system hardware. Guest OSs need to be adapted to control the way they access to hardware resources. It is the case of PikeOS that requires co-design on several best-effort but main line OSes (like Windows series, or secured Linux).
Full hardware virtualization is in charge to fully mimic hardware interfaces and control access to these virtual resources. Such a solution leads to performance losses but increase interoperability and increase the separation of preoccupations between the partition integration, and the development of a single partition.

Here is the list of platform considered: QNX Neutrino RTOS Safe Kernel [72], SYSGO PikeOS [73], WindRiver 653 & Cert Platform [74][75], INTEGRITY suite (RTOS & Middleware) [76], POK [77]. Most of these systems are commercial product for safety critical software deployment but POK (being open source and royalty free systems). In the table next page, we will use detail the scheduling model provided by these partitioned systems (among partitions, e.g. monitor level, and within partitions, e.g. at task level), and the failure containment properties (containment of real-time scheduling failure, and resource partitioning failure)

<table>
<thead>
<tr>
<th>Platform/Framework</th>
<th>Standard compliance</th>
<th>Real-time support</th>
<th>Software failure containment + architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNX Neutrino Safe Kernel</td>
<td>Safety: IEC61508-SIL3 API: POSIX PSE52</td>
<td>Monitor level: Adaptive Partitioning Scheduler (slack time reallocation) + minimal activity ratio Task level: unique priority scale</td>
<td>Space isolation: memory segmentation, but no sanitization a priori, nor resource partitioning Time isolation: adaptive scheduler can be disabled providing strict time slice partitioning Architecture: micro-kernel + Drivers in Partitions</td>
</tr>
</tbody>
</table>
## INTEGRITY

**Safety:**
- DO-178B
- IEC 61508-SIL4
- API ARINC653, POSIX

**Monitor level:** time slices

**Task level:** fixed priorities + others with partitions personalities (virtualization)

**Space partitioning:** fully static resource partitioning + system memory error detection mechanisms (exhaustion…)

**Time isolation:** Fully preemptive robust time partitioning

**Architecture:** not available

## POK

**Safety:**
- no certification,
- DO-178B compatibility (test)

**API:**
- ARINC653, POSIX

**Monitor level:** time slices, and fixed priorities preemptive partitions

**Task level:** fixed priorities + FIFO, Round Robin, (and EDF in stable development snapshot only)

**Space isolation:** Fully static resource partitioning, + memory sanitization

**Time isolation:** Fully preemptive robust time partitioning

**Architecture:** isolator with modular architecture to host user defined scheduling policy with partitions. Drivers in partitions for most devices

Remark: robust preemptive time partitioning mean that no partition will block the system more than $\Delta$ time unit once it has overran its allocated time budget. ($\Delta$ being the partition switching temporal upper bound – encompass CPU time, hardware sanitization time…).

### 4.4 Discussion and impact on the functional code integration process

On of the key issue in recent execution platforms lies in their configuration: these platform have been designed to offer a very large variety of non-functional services for security, fault containment, real-time scheduling… All these features may not even be compatible. With respect non-partitioned system, the core issue is related to distributed architecture with heterogeneous hardware components. For partitioned systems, one of the key concerns is related to experimental platform availability whereas several commercial solutions exist but scarcely provide enough detailed on the system internals. On the other hand, Telecom ParisTech has developed a first incomplete prototype with direct access to code as it has been developed under BSD license. Compared to other implementation of partitioned system, this implementation enjoys several good features for efficient collaboration (direct access to system code, experience in using and improving the prototype, several freely available use cases). Yet this prototype is far from providing all required features with respect to PARSEC objectives: poor modularity in scheduler implementation. No programming interface to integrate user define partition schedulers.

The next step is the functional code integration on execution platforms. There are issues related to wrong usage of required services (e.g. for real-time constraints). Moreover, there are also problems coming from incompatible services provided by a same execution platform. Concerning the second point, we can expect that no specification would require conflicting services. Nevertheless, we have to provide some evidence that the functional integration process will not create such conflict. These evidences could be obtained by construction or obtained afterward through verification. In both case, the integration process need to be automated.
5 Code generation for predictable system deployment

Once a suitable model is available for a system, the remaining problem is that of producing a correct and efficient running implementation. A good implementation methodology should provide:

- Automatic synthesis of the implementation code, to help reduce coding errors.
- Synthesis of the real-time schedules, or at least scheduling analysis to check whether the implementation satisfies the specified resource use limits (timing, memory, etc.).

Both the frameworks and tools chosen for use within the Parsec project (SynDEx and Ocarina) provide significant aid in these directions. The AAA methodology and SynDEx tool provide automatic real-time scheduling of synchronous specifications under timing constraints and automatic implementation of the generated schedules on various distributed platforms. Instead of synchronous specifications, Ocarina takes as input AADL models where the scheduling policy is already fixed. It provides scheduling analysis, automatic implementation generation for various platforms including ARINC/POK, and an interpretation of AADL models into Petri Nets which allows the use of formal verification. We first present how such framework can be connected either through gateways and semantics equivalence to existing high-level component based description languages like COAL or MARTE. Each approach is associated to some extent to gateways allowing them to draw their specific input models from higher level description like component based design in MARTE, SySML or COAL. These approaches belong to two distinct families of approaches for automatic code generation on execution platforms: code generation from synchronous execution models, and code generation for event based or asynchronous execution models.

5.1 Code generation support in Synchronous Language approaches

From the beginning, the development of synchronous formalisms [95][90] has been driven by the need of safe real-time systems. There are today two main commercial products based on synchronous formalisms research: Scade Suite and RT-Build. A third product [93] has an uncertain availability status (property of Synopsys, inc.), and since its input language is similar to that of Scade we are not detailing it here.

In addition to these tools, we shall provide brief information on their academic counterparts (Signal/Polychrony for RT-Build, Lustre and Esterel for SCADE Suite), and some academic works relevant to this project. We shall present in extensor existing work on the AAA methodology and the SynDEx tool.

5.1.1 SCADE

SCADE Suite (Esterel Technologies) is a model based development environment dedicated to critical embedded software. It is a commercial tool, developed by Esterel Technologies. The SCADE development environment allows:

- The specification and simulation of system functionality using a graphical synchronous formalism. The formalism allows the joint hierarchical definition of both data flow and automata defining execution modes. SCADE can import Simulink/Stateflow specifications, and UML/SysML architecture models.
- Rapid prototyping, high-level simulation, testing, and formal verification of the models (the latter using state-of-the-art tools from Prover Technologies).
- Automatic generation of optimized sequential implementations and simulators.
- Manual refinement of the specification going all the way to the generation of multi-rate, multi-task implementations and simulators [98]. In this process, emphasis is put on the use of memory cells, to avoid data corruption. It is not clear if real-time guarantees are given for these implementations by the Scade Studio tool, or if they must be verified using separate tools (formally or by simulation).

Exiting work on Lustre/Scade, not included in the commercial product allows the optimization of memory use [102], and the generation of distributed time-triggered implementations over TTA (P. Caspi, 2003)[99].

The graphical formalism used by SCADE is based on Lustre [96], Esterel [91] [92], as well as the graphical formalisms SyncCharts [100], Argos [101], SAO which it largely supersedes. Both Lustre and Esterel offer approaches for automatic code distribution and worst-case execution time analysis. The quasi-synchronous approach for automatic control system development has been developed for use on top of the Lustre/Scade tool chain.

5.1.2 RT-Builder/Signal/Polychrony

RT Builder (Geensoft) is a specification and analysis environment dedicated to critical embedded software. With RT-Builder, users can model, simulate and validate executable specifications and prototypes of their systems prior to HW availability. Based on the polychronous formalism Signal/Polychrony (INRIA/Espresso team) [97], RT-Builder supports both asynchronous and synchronous aspects of real-time developments thanks to its GALS (Globally Asynchronous Locally Synchronous) approach.

RT-Builder comes with existing libraries modeling advanced Real-Time Operating Systems (RTOS) and Bus libraries: ARINC653 for Integrated Modular Avionics, CAN, OSEK, FlexRay standards for Automotive. These libraries can easily be customized or extended to integrate dedicated needs or provide a powerful support of customer's real-time choices.

Existing work on Signal/Polychrony, not included in the commercial product, allows the generation of multi-threaded code. Like all synchronous languages, Signal/Polychrony can be used to perform a manual refinement of the specification allowing a multi-rate, multi-task simulation.

5.1.3 AAA/SynDEx

SynDEX [103] is a system level CAD software based on the algorithm architecture adequation (AAA) methodology for rapid prototyping and optimizing the implementation of distributed real-time embedded applications onto multicomponent/distributed architectures. SynDEX takes as input 3 elements: 1) a synchronous functional specification, 2) a specification of the target execution architecture (hardware and basic software), and 3) a timing specification defining the WCETs of the basic computations and communications on the elements of the architecture. It produces an implementation satisfying desired timing constraints.

The SynDEX tool allows:

- Rapid prototyping of complex distributed real-time embedded applications based on automatic code generation in three steps:
  1. Implementation onto a workstation for functional simulations,
  2. Implementation onto a multi-workstation for studying the parallelism benefits and accelerating functional simulations,
  3. Implementation onto the targeted multi-component architecture, made of programmable components (processors) and non-programmable components (specific integrated circuits)
all together connected through communication media (bus, links, crossbar, etc.), for real-time execution.

- Automatic generation of safe and optimized distributed real-time code thanks to formal verifications and exploration of possible implementations manually, or automatically with optimization heuristics, based on multi-periodic distributed real-time scheduling analyses.
- Hardware/Software co-design through multi-component architecture when some parts of the application must be implemented by software running on processors, while other parts must be implemented by hardware running on specific integrated circuits.
- Interface with domain specific languages (DSL) such as the synchronous languages (Esterel, SyncCharts, Signal) providing formal verifications, AIL a language for automobile, Scicos a Simulink-like language (access to the Scicos-SynDEx gateway), CamlFlow a functional data-flow language, UML2.0 with the MARTE profile, etc.
- Interface with the integrated circuit level CAD tool SynDEx-IC which allows the implementation of a function (operation) onto a specific integrated circuit that can be used as a non-programmable component in the multi-component architecture.

Thus, SynDEx is a system level CAD tool offering a seamless software environment to help the user from the specification level (functional specifications, distributed hardware specifications, real-time and embedding constraints specifications) to the distributed real-time embedded code level, through (multi-) workstation functional and timing simulations.

5.1.4 Comparison of candidate technologies and

While the SCADE-based implementation methodology is more mature from an industrial point of view, we chose for our work in the Parsec project the AAA/SynDEx methodology and tool. This choice is based on the following arguments:

1. Unlike SCADE, SynDEx proposes a fully automatic scheduling and code generation technique going all the way from the specification to the running implementation. Our objective is to extend this scheduling and code generation technique to cover the time-triggered platforms used in the Parsec project.

2. Given the proximity of the SCADE and SynDEx (synchronous) specification formalisms, the SynDEx tool can be easily adapted to run on SCADE specifications. However, this is outside the scope of the project, as our objective is to provide the proof of concept for the new automatic implementation technology.

5.2 Code generation support for architectural/asynchronous deployment & modeling approaches

Such approaches to handle the last step of the development process (i.e. integration of the application code on the execution platform), follows a bottom-up modeling approach. The problem is spelled as follow: given that the execution platform is predefined and already include the scheduling algorithms, can we provide some support to automatically configure the execution platform, generate the integration code to bind functional components to the execution platform. Most of existing solutions are either not adapted to real-time deployment or correspond to niche. Few solutions exist for real time code generation from UML/MARTE model but with substantial extensions.

5.2.1 Code generation from UML timed profiles

In this category, one can found many non-hard real-time solution dedicated to java source code generation like the HUGO framework [78]. Beside, Papyrus UML modeler provide code generation feature toward Ada95 or Java Code. This initiative is still not matching the requirement of real-time
systems. These plug-in essentially manage signature generation and thus provide partial code generation features. The underlying code generation engine is Acceleo [79] that work essentially based on templates. Thus, any optimization, or integration code generation has to remain simple or has to be managed at model level and has to be handled by model transformation before pretty printing by Acceleo.

Another technology available is for code generation from UML model is provided by GENE-AUTO, that allows code generation from Simulink [83]. Despite, the approach can be extended to any input model thank to the capability of GENE-AUTO to facilitate code generator design, the tool only provide support for synchronous model up to now.

5.2.2 Code generation from Architectural Description Language AADL

There exist several initiatives to generate code from AADL either in generic programming languages like C, Ada, or in dedicated simulation languages. Each code generator is often strongly bound to some execution platform or at least to its programming interface and configuration files.

[87] investigates code generation for OSEK/VDX platform (execution platform with standardized programming interface dedicated to automotive systems). This code generation service is part of a larger tool set dedicated to code generation for embedded systems and more precisely automotive embedded systems.

To our knowledge, there are no other code generation framework with already implemented code generation service for many distinct execution platform and programming interfaces given AADL model as input, but the Ocarina Tools set.

Ocarina is a tool suite written in Ada to manipulate AADL models. Ocarina proposes AADL model manipulations, generation of formal models, performs scheduling analysis and generates distributed applications. This tool suite can deduce from AADL model a large number of models for which there exist generic verification or profiling tools.

An AADL parser/printer has been developed to help validating and reformatting AADL models. It
can read an AADL description spread over multiple files. This is compulsory for large-scale project. AADL descriptions can be processed to fully deduce and then generate the configuration of the middleware. This approach allows one to describe an application and then fully generate it. This reduces the error-prone process of configuring complex runtime like CORBA middleware. The current version of Ocarina allows the user to generate code from an AADL architecture description towards an Ada application running on top of PolyORB. In this context, PolyORB acts as both the distribution middleware and execution runtime on all targets supported by PolyORB. In this context, it means essentially to generate the inter-task communication mechanisms, shared resources and ensure that the execution platform is correctly configured to fully support communication semantics used in architectural description of the application architecture.

In addition to the previous generator, a specific runtime for distributed high-integrity applications based on the Ada Ravenscar profile has been designed, PolyORB-HI. This runtime inherits concepts from the PolyORB middleware, while using a high-integrity compliant source base. Two target languages are supported: Ada 2005 and C.

By exploiting AADL models information and runtime knowledge, Ocarina provides a way to configure WCET evaluation tools such as Bound-T by model transformation and parameters extraction. WCET evaluation then can be exploited by scheduling analysis tools. This feature is completed by an constraint interpreter that checks architecture validity on a given fully instantiated model (i.e. bound to some hardware). This tool is called REAL interpreter.

**Gateways to code generator for predefined kernel with fixed scheduling policy**

In addition, MyCCM-HI also manages operational mode reconfiguration. Namely, the framework is able to let the architecture evolve according to a set of pre-defined transitions between modes. Evolutions of the architecture may comprise activation / deactivation of connections between components or threads. Since mode management is specified using automata, the framework is able to generate the code that implements the automata and ensures that the system remains in the range of specified states. The generated code also ensures that automatic reconfiguration of the system is performed when no running thread impacted by the modification is running.

![Figure 5-4: MyCCM-HI Code generation process](image)

The process associated to MyCCM-HI framework is shown on Figure 5-4. The process starts with the description of the architecture of the application (currently described using the custom textual
syntax COAL) and generates a low-level AADL description describing the artifacts required to implement the initial architecture; an abstraction layer allowing to provide the standard LwCCM API on top of the low-level architecture described in the AADL description.

The generated AADL specification is then compiled with Ocarina (cf. section XXX for details) to generate the low-level source code that manages real-time configuration, and low-level communication management (applicative message routing corresponding to components connections, periodic thread managements, priority management, etc.).

The implementation of the software component is provided by the user as a set of C source files that are integrated with the generated code in order to output a set of statically linked executables. The interface between the source files provided by the user and the code generator is defined by Lightweight CCM’s CIF (Component Implementation Framework) standardized interface.

As mentioned on Figure 5-4, there is an ongoing work supported by the VERDE project to bridge MyCCM-HI and UML/MARTE syntax (using the Papyrus tool developed by the CEA) in order to provide an already standardized syntax (MARTE) for the concepts defined in COAL.

5.3 Conclusion

This section pointed out the growing number of approaches developed to automate the deployment of application code upon standardized execution platform, i.e. execution platform known and stable API, behaviors and non-functional properties. These approaches correspond to distinct specification styles to describe the functional code expected behavior. This diversity in these description lead to framework that can hardly be compared: what to assess?

One of the lesson learnt is that if several specification styles exist to describe behaviors then it would be interesting to support at two specification styles that are not too close (but that can be compared with respect to common objectives: hard deadline enforcement, schedule validity…). This diversity may also be interesting in operation as it may be a way to enforce N-versioning supported by MDE and code generation.

The last outcome of this chapter is related to the trend leading to code generation framework targeting an increasing number of execution platforms: RTOSes, bareboard, dedicate run-time (Ada Ravenscar), AUTOSAR/OSEK, ARINC653. One important feature of code generation frameworks is to implement code generation as pretty printing of accurate auxiliary models of system deployment (e.g. in SynDEx or Ocarina).
6 Verification methods

We distinguish the verification methods depending on the step of the engineering process at which it can be applied. We distinguish three steps: on the functional specifications, on models of the deployed architectures. The two first step are grouped as the effort made in the engineering process lead to quite straightforward verification strategies.

6.1 At functional model level or architectural level description

6.1.1 Verifications on requirement and high-level elements in B

On of the main idea underlying event-B modelling activity is that proof is a task that is fully part of modelling task. In other words, the proof of the model constrains how one may finalized it. The tools developed [80] promote a rapid and constant interaction between the modelling and proof tasks.

The proof activity ensures that the model is consistent. This is however not enough. It is indeed quite possible to achieve a sound model entirely proven, but that is an inaccurate representation of the considered reality. It is with the aim to detect modelling errors that model animation techniques have been implemented within tools such as ProB or AnimB [81]

6.1.2 At architectural model to verify deployment and system dimensioning

One can distinguish two kind of verification task on architectural models, architecture consistency verification, and resources allocation and deployment feasibility. The later pretty often lead to performance evaluation issues.

Architectural constraint checkers

Many architecture constraints languages exist to verify the consistency issues. Languages like Alloy, or OCL [84] provide for interpreter that allows automating these verifications. These methods belong to lightweight model checking approaches and provide automation for consistency issues coming from modular models containing a lot of correlated data. These tools often allows to operate with tool checking models with stronger semantics and more complex properties to verify like SPIN model checkers, or timed automata related tools like UPPAAL. Nevertheless, there still exist third party tools that could be used to assess the system deployment correctness (in terms of resource allocation)

Third party tools

Scheduling analysis tools will play an important role within the verification process of time critical system. We have identified 3 main technologies: MAST [88], Cheddar [86], and CARTS [89].

MAST provide analytic schedulability analysis avoid simulation, and thus limiting approximation to the hypothesis on task interaction, operating system impact on resources. It provides a very accurate model in which shared resources blocking time is considered when computing worst case response time of each task. Nevertheless, this tool does not provide any model for hierarchical scheduling analysis.

Cheddar provides different set of hypothesis on which the analysis method can be selected. In non-hierarchical models, fine-grained analysis can be performed taking into account shared resources in response times. They also provide a method to check hierarchical scheduling model with independent task model. They also provide simulation capabilities that can be used to check schedule for deterministic execution models.
The CARTS Tool seems to provide all the required feature with respect to hierarchical schedulability analysis. It relies on novel verification algorithms. For instance, it enjoys scheduler configuration for ARINC 653 execution platforms given timing constraints and time budget consumption for tasks.

An important point is to understand how third party expert tools can be integrated in verification processes through verification scripting language. An example of such language is REAL.

An example of integration between third party tools and light-weight model checkers

REAL is a domain specific language that has been provided to build constraints adapted to check dimensioning properties over an AADL properties from its attributes and parameters computed by third party tools like WCET analyzers. Thus, REAL can check high level requirements on deployment strategies, once third party tool performs complex verifications task or computations. This tool has been used to conduct consistency verification on various middleware with strong resource allocation constraints in MOSIC for instance.

6.2 **At business source code level**

Architectural description language for embedded system architecture like MARTE, AADL or EAST-ADL can be used as inputs for configuration We discuss in this part the verification of component implementation at code level. We start from the formalization of component contracts and then we describe techniques used to verify software at code level. We conclude with the needs to extend COAL with behavioral specifications and to fill the gap between those descriptions and code level verification tools.

*What is a component?*

The UML 2 specification gives the following definition: **Components**: A component is a self-contained unit that encapsulates the state and behavior of a number of classifiers. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces. A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment. Interfaces: An interface declares a set of public features and obligations that constitute a coherent service offered by a classifier. Interfaces provide a way to partition and characterize groups of properties that realizing classifier instances must possess. An interface does not specify how it is to be implemented, but merely what needs to be supported by realizing instances.

These definitions insist on the notion of contract expressed via provided / required interfaces. In practice, in the CORBA Component Model for instance, contracts are expressed by means of the Interface Description Language (IDL): The OMG Corba Component Model specification says: **A component type is a specific, named collection of features that can be described by an IDL component definition […]**. Component supports a variety of surface features through which clients and other elements of an application environment may interact with a component. These surface features are called ports.

But the concept of contract as supported by the IDL is far from what is expected as stated by B. Meyer (in [39]): **In most modern component frameworks - such as the Interface Definition Languages of COM and CORBA - we can rely on some type specification for the arguments, but it is not enough. We badly need semantic specifications as well.** In [40], Meyer explains that a contract on a component should provide an answer to three questions: what does it maintain? (invariance), what does it expect? (pre-condition), and what does it guarantee? (post-condition). But contracts in current component-based designs are often limited to the specification of interfaces as a set of operations signatures.
What is a contract?

Several classifications of contracts have been proposed. We will retain two axes of classification, the first one by Beugnard [41] categorize contracts according to the level of information they carry:

- At syntactic level (level 1), which concern the signature of the operations supported by a component, that is to say the name of the operation, the list of parameters and the list of exceptions. This is what we find in CORBA IDL or in Java interfaces, for instance.
- At behavioral level (level 2), which express the conditions that shall be fulfilled and expected by the caller of the component’s services in a context where code is executed sequentially or “atomically”.
- At synchronization level (level 3), which express the dependencies between services provided by a component in terms of synchronization between method calls.
- At quality of service level (level 4), where the expected behavior specified by the previous levels of contracts is quantified. Quantification may concern safety, availability, maintainability, latency, reliability, and all other “ilities”.

On an orthogonal view, when component frameworks introduce composite components, this requires classifying contracts according to the kinds of interactions. Ozanne [42], for example, gives an overview of models putting the emphasis on the notion of composition (like SOFA, FRACTAL), he identifies:

- interface contracts: established on the connection point between to client/server interfaces and expressed on the entities which are visible on the interface,
- external composition contracts: on the external side of the component membrane and expressing component usage rules and external behavior,
- internal composition contracts: on the internal side of a composite component membrane, and only referring the component internal interfaces and the sub-components external interfaces,
- library contracts: functional contracts on the reusable units of the underlying language (C functions for instance) and on middleware services.

In this context, Plasil [43], describes the types of ties that this architecture involves on interfaces and then the verifications that must be done: (a) binding of a requires-interface to a provides interface between two subcomponents, (b) delegating from a provides-interface of F to a subcomponent’s provides-interface, (c) subsuming from a subcomponent’s requires interface to a requires-interface of F, (d) exempting an interface of a subcomponent from any ties (the interface is not employed). A component can also be specified as primitive, which means that there are no subcomponents and its structure / implementation will be provided in an underlying implementation language, out of the scope of the component mode.

How to formalize component contracts?

The formalization of component contract is based on the Assume/Guarantee principle, which uses the natural decomposition of the system into its components to break-up the verification task by checking components in isolation: If a component in the system provides the G property assuming the Hypothesis A, the system will provide G if all the M use contexts in the system respect A. This formalization may involve pre/post conditions on operations, invariants on component local data, protocols descriptions, input output automata …

We describe here the level of formalization of contracts proposed in existing component frameworks.
In COAL [44] as in CCM, only contracts at syntactic level are specified using IDL language. The mode automata, introduced by COAL, describe the dynamic configuration of the component connections but there is no specification of the behavior of the components themselves.

In the .NET framework context, Microsoft makes a lot of work concerning specification and verification of components [http://research.microsoft.com/en-us/projects/contracts/]. They have proposed several languages to express component specification. ASML[45,46] is based on abstract state machines, it aims at components model based testing and is equipped with tools for test sequences generation and runtime verification. It gives high level data types (sets, mappings ...), is able to manipulate .NET notions (interfaces, classes, methods, events ...), and comprehends constructs for parallel composition, sequencing of steps, non-deterministic choice, loops, mandatory calls and exceptions. SPEC# [47] is an extension of C#, it inspired from Eiffel and JML and provides formal specification of behavioral interface contracts for primitive components, based on notions of pre-conditions, post-conditions and object invariants. The goal of this language is to make formal verification of component implementation possible with use of automatic or assisted provers. More recently, they advocate embedded contracts [48] which are a generalized specification of the contract notions present in SPEC# and provide implementation of this specification in the form of a contract library available in each of the .NET language.

SOFÁ2 [http://sofa.ow2.org/] and FRACTAL [http://fractal.ow2.org/] are two component systems allowing component composition: Work on formalization and verification of contracts in these systems have been studied in the CoCOME contest [49, 50, http://agrausch.informatik.uni-kl.de/CoCoME/]. These formalizations often use behavior protocols proposed by Plasil [43] and extended in [50, 51]. The objective of this language is to provide an easy-to-read notation that supports specification refinement and tractable formal reasoning. This language is based on process algebra. A component’s behavior is described as it is observable by its environment, i.e. terms of capturing the traces of events related to method calls crossing the component boundary. Assuming component A requires an interface I that is provided by component B and assuming the interfaces A.I and B.I are bound together, these events are:

- issuing a method request M on component A’s required interface I: !I.M↑
- accepting a method request M on component B’s provided interface I: ?I.M↑
- sending a response to method request on component B’s provided interface I: !I.M↓
- accepting a response to method request issued on component A’s required interface I: ?I.M↓

A behavior protocol is then described by a regular-like expression built on these basic elements, where they can be connected together using several operators (; for sequence, + for alternative, * for repetition and | for parallel execution). To simplify expressing method calls, the following abbreviations are introduced: ?I.M (stands for ?I.M↑; !I.M↓), !I.M{P} (stands for !I.M↑; P; !I.M↓), where {P} specifies a reaction to accepting the method call (P is a protocol here). The abbreviations !I.M, and !I.M{P} have a similar meaning.

Behavior protocols are used at three levels:

- Frame protocol, which specifies the valid exchanges at the requires-interfaces / provides-interfaces of a given component.
- Architecture protocol, which specifies the valid exchanges when considering the uppermost subcomponents of a given component. Basically, this protocol is obtained by “merging” the frame protocols of the direct subcomponents.
- Interface protocol, which specifies the acceptable order of method invocations on one interface. This information is already contained in the frame protocol, but it provides a focused view and it supports the incremental specification of the component.

Accordingly, the following protocol compliance levels are proposed:

- Interface – interface compliance: an interface protocol P1 conforms to another interface protocol P2 if and only if it can generate a language which is a subset of the language generated by P2.
• Interface – frame compliance: for any interface I provided by the component, the interface protocol complies with the frame protocol. Reciprocally, for any interface I required by the component, the frame protocol complies with the interface protocol.

• Frame – architecture compliance: if the use of a component complies with the frame protocol (precondition), then the traces generated by the architecture protocol shall be allowed by the frame protocol. Checking conformance at this level may be useful to perform early verification of a design.

BIP [http://www-verimag.imag.fr/BIP,196.html] is a coordination language defined as an extension of the C programming language. It leverages on C style variables and data type declarations, expressions and statements, and provides additional structural syntactic constructs for defining component behavior, specifying the coordination between components and describing the priorities. The language enables to express the concurrent and sequential behavior of systems, as an interconnection of components. A system can be described hierarchically with the notion of compound component, and timing can also be explicitly modeled in the same description. At the leaf level, atom components have associated behaviors described as a set of transitions labeled by ports, guards and actions that can be calls to business functions coded in C, these behaviors are possibly non-deterministic. Connectors specify the co-ordinations between the ports of components, and the associated guarded actions, they enable description of heterogeneous interaction including strong synchronization by rendezvous as well as weak synchronization by broadcast.

This BIP approach “considers that components are the superposition of three distinct layers describing, respectively Behavior, Interaction and Priority. Interaction involves synchronization between components behavior with possible transfer of data. Priority is an elementary control mechanism for conflict resolution which can be used to restrict non-determinism due to concurrency and allows direct expression of scheduling policies. Composition of components consists in composing separately the corresponding layers by using associative and commutative operators parameterized by "glue".

The tool-chain contains a model compilation and execution engine. This engine also provides access to model-checking tools of the IF toolset such as Aldebaran and the D-Finder tool. These tools ensure that the model meets properties such as deadlock-freedom, state invariants and schedulability.

**How to verify contracts at code level?**

We restrain here to the verification of primitive component (written in a programming language) against the specifications of this component.

In the Component Based Design community, there is little work on the verification of primitive components. Generally, in this context verification is considered at model level (for example in the BIP platform), and once the model is validated, the code is automatically generated. We can however mention the extraction of behavior protocols from primitive component coded in Java [53] using the Java PathFinder model checker, this work aims at inferring formal specifications for unspecified existing components, but it could be derived to compare the implementation to an expected specification.

In the case where the primitive component is manually coded, one can use the classical code level software verification techniques, but these techniques are usually applied on a complete application or on a modular way at a function or a class boundaries and the component specifications, written at the component boundaries, will have to be converted to fit the function level.

These techniques mostly aim at the detection or the proof of absence of runtime-errors (divide by zero, array overflow, invalid memory access ...), the conformity with functional properties (expressed with function pre/post conditions and global or types invariants). In certain cases it is possible to verify
temporal properties (expressed by temporal logic formulas or automata); this can be done by translating the formulas into assertions (pre/post) associated to the events occurring in the formula. [54] uses this technique to verify temporal formulas on java applications by converting them into JML assertions; the Frama-C Aorai plug-in [http://frama-c.com/download/aorai] gives a similar solution for C programs.

The main code level verification automatic verification techniques are:

- Automatic test generation: since exhaustive software testing is impossible, the goal is to produce tests with good properties, these properties being expressed according to code and specification coverage criteria. Tools working on code coverage try to generate tests responding to the all-paths, all-branches, MC/DC criteria. PEX for C# [http://research.microsoft.com/en-us/projects/pex/] and PathCrawler for C [55] are two examples of such tools. When specification is available at code level (code annotations) these same tools are able to produce tests to cover the specifications [56]. And in this case, these tools can also be used to detect defaults in the code such as runtime-errors or functional non-conformity. These tools explore by symbolic execution the program control flow graph to obtain path conditions (conditions on the input data needed to follow this path). They transmit these conditions to a solver which is in charge of finding a solution.

- Abstract interpretation uses abstractions (interval, octagon, polyhedron …) to avoid exploration of all program execution paths. It gives a sound verdict but with the drawback of losing precision (false positives). The tools built on these concepts (Polyspace [http://www.mathworks.com/products/polyspace/], Astree [http://www.astree.ens.fr/], Frama-C Valviewer plug-in [http://frama-c.com], …) are mainly used to check the absence of execution errors (overflow, divide-by-zero, out-of-bounds array access, and other run-time errors in source code. AbsInt Company [http://www.absint.com/] proposes a bunch of such tools to analyze architecture dependent properties such as worst case execution time, cache performance, stack usage and pipeline behavior.

- Model checking consists in exploring the state space of a model to check the validity of a property. This technique has been extending to software verification, but the combinatorial explosion implied by this analysis has conducted researchers to integrate solutions coming from other domain like symbolic execution or abstraction. [57] makes a good survey of the techniques used in model checkers and [58] gives an overview of current model checkers and evaluates their applicability to embedded software.

- Automatic or assisted proof: Tools like SPEC# static verifier [http://research.microsoft.com/en-us/projects/speersharp/] for C#, ESC/JAVA [http://kind.ucd.ie/products/opensource/ESCP Java2/] for JAVA, HAVOC [http://research.microsoft.com/en-us/projects/havoc/] and the Frame-C JESSIE plug-in [http://frama-c.com/jessie.html] for C, VCC for concurrent C [http://research.microsoft.com/en-us/projects/vcc/] use logical inference to prove the validity of user defined properties. They transform the properties to check into verification conditions that they transfer to general provers. Some tools like JESSIE have a modular architecture allowing to address several automatic provers (Simplify, Alt-Ergo, Yices, Z3, …) or when the verification is too complex to transfer the problem in proof assistants (Coq, PVS, Isabelle/HOL, …) where the user will have to guide the proof.

All these tools need a way to specify the properties to verify or the context in which the verification must be done. This means way to express post and pre conditions invariants, but also means to express complementary information enabling the coding of the properties for example logical predicates, ghost variables or functions … Languages to represent those properties are generally closely tied to the programming language, hereafter are the main specification languages defined for several programming languages.

- For JAVA: JML (http://www.eecs.ucf.edu/~leavens/JML/) is a behavior interface specification language which allows including annotations into JAVA comments. It is widely used and, in

- For ADA: SPARK (http://libre.adacore.com/libre/) is a strictly defined and enforced subset of Ada complemented by an expressive system of contracts that precisely convey the design or the specification of the program itself. It can be completed by the RavenSPARK option that brings the power of Ada’s Ravenscar tasking profile to SPARK. Those languages are used for example by Adacore verification tools.

- For C#: SPEC# (http://research.microsoft.com/en-us/projects/specsharp/) used by SPEC# static verifier and PEX test generator.

- For C: ACSL is the core specification language used in Frama-C platform (http://frama-c.com) and is exploited by the Frama-C verification plug-ins.

We make a focus on the Frama-C platform with the ACSL specification language and on the PathCrawler tool that are partly developed at CEA and that will be used in PARSEC. Frama-C is a suite of tools dedicated to the analysis of the source code of software written in C. It gathers several static analysis techniques in a single collaborative framework. The collaborative approach of Frama-C allows static analyzers to build upon the results already computed by other analyzers in the framework. See for a good introduction, the Dassault Aviation paper describing an industrial experiment of code verification with Frama-C [59].

The main features of Frama-C are:

- The ANSI/ISO C Specification Language (ACSL) is a behavioral specification language for C programs. The design of ACSL is inspired of JML. It also inherits a lot from the specification language of the source code analyzer Caduceus, a previous development of one of the partners in the Frama-C project. ACSL can express a wide range of functional properties. The paramount notion in ACSL is the function contract. ACSL allows users to write contracts that range from the low-level (“this function expects a valid pointer to int”) to the high-level (“this function expects a nonempty linked list of ints and returns the greatest of these ints”). It is expressive enough to write complete specifications for many functions, but it can also be used for writing partial specifications. ACSL is a formal language. This means that the specifications written in ACSL can be automatically manipulated by helper programs, in the same way that a programming language is a formal language manipulated by a compiler, and by opposition to informally written comments that can only be useful to humans.

- The Value-Analysis plug-in applies abstract interpretation to automatically compute variation domains for the variables of the program. The results of the values analysis can be exploited to explore graphically the possible values of variables in each point of the analyzed program. They also can be used to infer the absence of run-time errors.

- The Jessie plug-in uses Hoare-style weakest precondition computations to formally prove ACSL properties of a program fragment and to do safety checking (validity of memory accesses, absence of integer overflows, validity of operations on integers, termination). Internally, Jessie relies on the languages and tools contained in the Why platform. Verification conditions are generated and submitted to external automatic theorem provers or interactive proof assistants.

- The Aoraï plug-in provides a method to automatically annotate a C program according to a LTL formula $F$ such that, if the annotations are verified, then this ensures that the program respects $F$.

- The Frama-C platform provides other specific plug-ins for browsing unfamiliar code (Impact analysis, scope and dataflow browsing, variable occurrence browsing) and for code transformation (semantic constant folding, slicing, spare code removing).

- Some constructions of the ACSL language are dedicated to the proof plug-in, but they cannot be easily taken into account by abstract interpretation plug-ins or for test purpose (runtime-assertion checking) of the code. In Hi-Lite project (http://www.open-do.org/projects/hi-lite/), a subset of ACSL is defined on which static and dynamic analysis methods will be used in a
collaborative manner (this project treats also the same objective in the context of ADA/SPARK).

PathCrawler:

- PathCrawler is a structural test generation tool based on symbolic execution and constraint solving. It analyses the source code and provides test cases according to a coverage criteria (today all-paths with a possible bound). It is able to reason on specification to provide test cases conforming to a specific context, and to look for test cases able to invalidate some properties.
- The PathCrawler tool is independent tool, but its integration with the Frama-C platform is currently studied to use it as a complementary tool to generate counter example for unproved properties.
- In the PARSEC project, work is planned to allow PathCrawler to exploit ACSL specifications.

So far we have discussed functional specifications. But we may also be interested in non-functional properties such as Worst Case Execution Time of component services that is necessary for automatic distribution of the application (Syndex) or for scheduling analysis. This information must be computed from the component code (when the component code is not yet available, it can be predicted and has then to be confirmed as soon as possible).

There are main two approaches to WCET evaluation. The dynamic approach used in tools like Rapitime [http://www.rapitasystems.com/] is based on measurement of execution on user defined scenarios, and the static approach proposed by tools like aiT [http://www.absint.com/] or BoundT [http://www.tidorum.fi/bound-t/] is based on analysis of code by abstract interpretation. The advantages of dynamic methods is that they are easy to use, and give a realistic value for the execution times, but, except when exhaustive testing is possible, they cannot ensure they give the worst execution time. The static methods give an over-approximation of the worst case execution time, they don’t need to instrument the code and to use testing, but they need accurate models of the processor, which are more and more difficult to build, and they need a lot of additional information from the user to ease the analysis. In the SPICES project [http://www.spices-itea.org], THALES and CEA have explored a third track which combines static and dynamic and is flexible to be able to evaluate the execution of the required portion of code (between two accesses to component ports) on the required platforms. This approach uses the PathCrawler tool that makes an analysis of the source code to identify the execution paths of the required operation and produces a test case for each path. The test cases are executed on the code produced and annotated with CPU time measurement mechanisms by the MicroCCM framework on the targeted platform.

As a conclusion, many tools are available to verify code at program or function level. To be able to apply them for primitive component verification, we need to extend the component specification language to be able to express the expected properties and to define how these properties can be translated to the code level specification. These extensions will have to be integrated to the selected component model within PARSEC.
7 Synthesis and discussion

In this survey, we focused on methods dedicated to software system engineering with modular system
descriptions. We identified core issues in the design of tools for managing such engineering processes.

In requirement engineering and high level functional modeling: one of the issues is to
determine where to stop in refinement process and when architectural framework should be used to
handle the system description. B methods provide formal foundations for engineering process of
systems. The main difficulty will lie in the interconnection of this formalism, and the associated
method with architectural modeling approaches.

Many architectural approaches exist. The choice of these approaches is more related and
guided by industrial requirement than methodological or technical consideration. In this context, some
partners have a long-term experience in the CORBA Component Model and its “embedded” version
LwCCM. One of the side conclusions of the section 3 on architectural approaches is that among model
such model, profiles need to be defined. Being UML or CCM both have been specialized in order to
handle by predictable engineering processes. Here, the choice of LwCCM and COAL is more related
to adaptation and formation effort minimization.

The section 4 allowed us to clarify the role of standardized programming interfaces like
OSEK/VDX, ARINC, POSIX… compared to non-functional requirement drawn from certification
processes. The former led to consider design patterns share by almost all the industrial actors in a
given domain. The example of avionics with ARINC standards is the most relevant in this field. Such
standardized programming interfaces open highway for automatic code generation as the interface of
the execution platform is now stabilized. Some difficulties remains in the way hierarchical scheduling
models could be analyzed enforced and assessed. This point will be of major interest with the work
planned in the PARSEC project.

These claims are backed by the catalogs of new methods available for automatic code
generation and functional code integration. In this field, two distinct types of generation processes
emerged: those defined from synchronous execution models, and those defined from asynchronous or
pattern based execution models. Here again, no clear connection is provided between very high level
modeling approach and such execution models. Moreover, there is still plenty of work on deploying
the functional code on partitioned execution platform on both sides (synchronous and asynchronous).

One interesting point was to notice that SynDEx and Ocarina toolkits offer an interesting diversity in
possible deployment target. For this reason, and in addition to the properties ensured by the code
generation processes they implement, these toolkits will ensure that the work performed in PARSEC is
integrated into tools that will be maintained, and will impact a large community.

Finally, the verification plans will be backed by two distinct verification approaches: verification on
models, and verification on code. Yet, one of the important features of PARSEC is to provide correct
by construction properties as much as possible. Thus, we expected to limit the verification effort
performing verification as early as possible and conserving these properties through predictable model
transformations.
8 References


Integrated Modular Avionic ARINC653 – ARINC600 – ARINC664 – standards

Clemens Szyperski1, Jan Bosch2, and Wolfgang Weck3, Component-Oriented Programming.


CORBA Component Model Specification, Object Management Group (http://www.omg.org)


8-40


[82] Jordi Cabot , Robert Clarisó , Daniel Riera, UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, November 05-09, 2007, Atlanta, Georgia, USA


[99] P. Caspi, A. C. (2003). From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. LCTES'03. San Diego, CA, USA.


