An Ontological Representation of the Characteristic Problems of Real-Time Systems

Antonio Monzón[1], José-Luis Fernández-Sánchez[2]

1: EADS-CASA, Military Transport Aircraft Division, John Lennon Av., 28906 Getafe, Spain
2: Industrial Engineering School, Technical University of Madrid (UPM),
José Gutiérrez Abascal, 2, 28006 Madrid, Spain

Abstract: Software Architectural Assessment is becoming a key discipline to identify at early stages of a system synthesis the most important problems that may become relevant in operation. This matter is especially critical for those systems with real-time constraints. Special emphasis shall be made on concurrency issues. Typical RTOS mechanisms supporting concurrency, such as semaphores or monitors, usually lead to execution time penalties hard to identify, reproduce and solve. For this reason it is crucial to understand the root causes of these problems and to provide support to identify and mitigate them at early stages of the system lifecycle.

The main objective of this paper is to propose a new classification of the most important problems related to real-time software systems and to provide mechanisms and guidelines to help engineers improve their architectural designs. The taxonomy has been applied to a particular architectural style (UML-PPOOA) and it is used as a reference to create a new assessment module on the PPOOA-Visio CASE tool [15] to support concurrency problems detection.

Keywords: Software Architecture, Real-Time, UML, Concurrency, Deadlock.

1. Introduction

Software Architecture Modelling is a relevant subject for the production of real-time systems. The development of AADLs in the last years has permitted to represent both structure and behaviour of such systems with very few details of the implementation.

The utilisation of UML notation to represent software architectures in combination with Model Driven Engineering ideas has become usual in the field [16]. Regarding this new approach to the problems of software architectures, some authors [9] have highlighted the intrinsic consistency problems of UML and have proposed rules to assure consistency of models based on UML abstractions. Other approaches to software architecture [14] focus on the notational concerns and the importance of visual capabilities to help engineers representing systems.

An architectural style [8][10] is a consistent set of modelling abstractions with architecting rules to plug constructive elements to build system models. This approach is very promising because the style well-formedness rules assure a minimum consistency level. Nevertheless, the capability of representing certain aspects of the system is not powerful enough to make an architectural style really useful. In addition to the notational or syntactic capabilities of a style, it is also required to provide guidelines and rules to help software architects to produce reliable models concerning to particular problem domains.

In the field of real-time and embedded software, formal methods have been developed to specify and verify system properties. One of the objectives of such formal methods can be, for instance, deadlock avoidance [1][4][13]. The main drawback of formal methods is that they are intrinsically complex and difficult to apply in industrial environments.

In this context, PPOOA architectural style [10] has been selected because it combines UML notation, MDE concerns, it allows software structural analysis and it is particularly useful to explicitly modelling concurrency aspects.

In addition to the engineering knowledge required to assess the quality of a model, it is also worth providing computer-aided architectural tools to support the automation of the engineering rules. Although many CASE tools exist in the market to support the software design activities, most of them focus only on the notation issues with very little concern on the engineering rules support. There are a few commercial tools in the market [2][18] supporting real-time characteristics, but with no specific feature to analyse concurrency problems. For this reason a gap exists to cover the engineering support in appropriate modelling tools.

In order to ease the process to understand the problem context where the problems take place, an ontological approach has been followed. The proposed ontology classifies the problems in two main categories: concurrency and structural problems.

2. PPOOA Architectural Style

A software architectural style encapsulates decisions about its building elements and emphasizes important constraints on the elements and their

relations. The PPOOA (Pipelines of Processes in Object Oriented Architectures) architectural style for RTS provides constructive elements such as components and coordination mechanisms [10]. Constraints on building elements are represented in the metamodel and by guidelines. These guidelines not only represent the semantics of the style, they are also helpful for the software architect using the style.

The UML stereotypes are extended with the elements of the PPOOA style (periodic and aperiodic processes, controller objects, and coordination mechanisms). UML Activity diagrams are also adapted for PPOOA style requirements, specifically modelling resources and considering scheduling points [11].

The PPOOA architecture diagram is used instead of the UML component diagram to describe the structural view of the RTS architecture. Coordination mechanisms, used as connectors, are also represented in the architecture diagram.

The RTS behaviour view is supported by the "Causal Flow of Activities (CFA)" representation. A CFA represents a system internal view of the flow of activities performed by the system in response to an event. PPOOA uses the UML activity diagram with partitions to support allocation of activities to the architecture component instances.

For the purposes of this paper, these are the relevant abstractions used in PPOOA for explicit concurrency modelling:

- Task: PPOOA building element representing tasks or threads is Process. It may be periodic or aperiodic.
- Resource: Logical resources can be represented in PPOOA by Domain Components or Structures, used at a logical architecture level. These building elements are abstractions of design classes and abstract data types respectively.
- Semaphore: A pure synchronization mechanism. It is the PPOOA building element that supports the synchronisation of tasks. Semaphores are used to protect shared logical resources.
- Bounded buffer: A coordination mechanism representing a FIFO queue used to communicate asynchronously two tasks.

3. Taxonomy of Architectural Problems

Although the problems identified in the proposed taxonomy are well known in real-time systems community and have been extensively treated in the past, there is a lack in the consideration of all of them as a **consistent set of engineering problems** to be detected and mitigated in the architecture definition phase. Main stress is here on the classification of the problems (see Figure 1) versus the classical approach of handling them as isolated problems in very particular conditions and usually at late implementation stages.

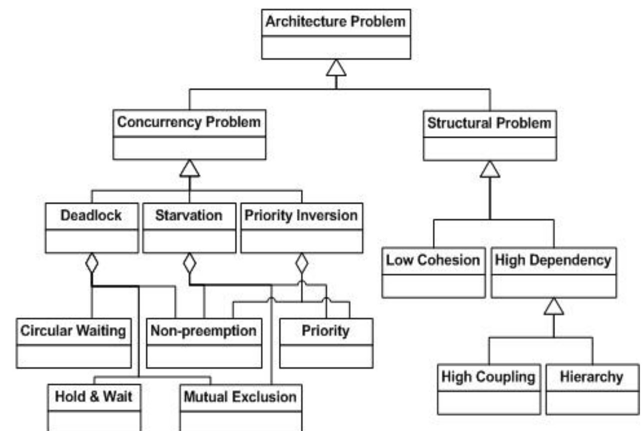

Figure 1: Taxonomy of Architectural Problems

Architectural problems are primarily classified in two main categories, regarding their relation to tasks synchronisation: Concurrency and Structural Problems.

The first category is further classified into three sub-categories: Deadlock, Starvation and Priority Inversion. Although the detailed justification for this selection is explained in a specific section, these problems have been particularly chosen because they are especially dangerous for the operation of system under time-constraints, as they cannot be detected in design time. They usually appear in execution time, even if the implementation is free of defects.

In addition to the classification, the main factors for the occurrence of such problems have been identified in the metamodel. They are discussed in detail in this paper.

In the structural problems category two main sub-categories are proposed: Lack of Cohesion and High Dependency. Both concepts are relevant for the proper design of architectural components. The second one is further sub-divided into other two categories: High Coupling and Wrong Hierarchy.

The relevant information for the development of this work comes from the following inputs (captured from PPOOA models):

- Dependencies: Usage relations among architectural components (UML-PPOOA Architectural View).
- Attributes: Values of the real-time attributes defined for the components (UML-PPOOA Architectural View).
- Semantic Information: Problem Domain related information (i.e., responsibilities implemented by a component).

- Time-related Information: Associated to PPOOA Activities (UML-PPOOA Architectural/Behaviour Views).
- Execution Sequence Information: Information related to the expected sequence of execution of activities allocated to components (UML-PPOOA Behaviour View).
- Tasks and shared resources priority information (UML-PPOOA Architectural View).

4. Structural Problems

4.1 Low Cohesion

In Object Oriented design, cohesion is a quality property of a component that shows how closely the operations within it (behaviour description) are related to each other. If a component contains non-related operations, the component implements heterogeneous functions, and thus it has a low cohesion. On the other hand, if all the operations within a component are related to each other, then the component is considered cohesive.

High cohesion is desirable, as it indicates good system decomposition. Low cohesion increases the complexity and therefore the probability of errors. Low cohesion components should be divided into others more cohesive. This factor has system robustness and maintainability implications, so it is clearly under the scope of this work.

In PPOOA, cohesion involves the responsibilities allocation to components. A component is considered cohesive if its behaviour is internally cohesive. The behaviour of a component is characterized by its interface, that is the set of all the operations contained in the components within the building element. The responsibilities are implemented in operations within components, according to the activities allocation represented in the CFAs. A component can be considered non-cohesive when its constituents are non-cohesive. At an aggregated level, a building element can be considered non-cohesive when some operations of its interface have no relation with the others.

4.1.1 Low Cohesion Assessment in PPOOA

An algorithm outline to detect low cohesion in PPOOA models is proposed here:

1. Check the operations that implement the interface of an architectural element in the PPOOA static model and consider those that use attributes of different internal components.
2. If the total amount of such disjoined operations is high, then there is a lack of cohesion.
3. Assign a percentage value to the lack of cohesion, considering the number of disjoined operations with respect to the total number of operations.

4.2 High Dependency - High Coupling

In Component Based Design, coupling is a property applicable to a set of components, which represents the number of dependencies among the individual components of a subsystem. A component depends on another when an operation of it needs to invoke an operation (or to access an attribute) of the other to perform its internal computation.

When many components depend on only one, it is considered that there is a high coupling in the set. Therefore, the set is very sensitive to changes in the invoked component. Low coupling is desirable, and high coupling should be avoided by distributing the responsibility of the invoked component among the rest of components. Once again, this factor has impact on the maintainability of the system.

4.2.1 High Coupling Assessment in PPOOA

An algorithm outline to assess high coupling in PPOOA models is proposed here:

1. Check the PPOOA static model and count the number of dependencies of each building element.
2. If there is one building element with more than 50% of the average dependency ratio, then the model could be considered highly dependent.
3. Assign a percentage value to the coupling factor, considering the number of high dependency components with respect to the total number of components in the model.

4.3 High Dependency – Wrong Hierarchy

Hierarchy is a general mechanism for the organization of a system in different aggregation layers, to deal with complexity at each level. It is a powerful mechanism while designing systems, but should be used according to the following rules:

- Components should have dependency relations only with elements at its level and not with components at other levels.
- Components can have only composition relationships with other level components.
- Dependency relations must be solved by dependency relations with lower level components.
- Elements at certain level can have relations only with the immediate lower level, not two levels below.

Although this one could be considered a less relevant factor, it has influence on the overall design consistency, and thus can also affect the maintainability.

4.3.1 Wrong Hierarchy Assessment in PPOOA

An algorithm outline to assess wrong hierarchy in PPOOA models is proposed here:

1. Check the hierarchy consistency rules in the PPOOA static model.

2. If the total number of violated rules is high then the hierarchy can be considered inappropriate.

3. Assign a percentage value to the wrong hierarchy factor, considering the number of violated rules with respect to the total number of rules.

5. Concurrency Problems

This kind of problem deals with the coordination of tasks in a multitasking scenario. Multitasking is a computing paradigm considering tasks that execute in parallel, sharing common resources (i.e. CPU, memory, database, etc.). At a given time only one task is permitted to be using the shared resource. Therefore multitasking computing entails using scheduling mechanisms to decide which task may be running at the resource at a given time and when other waiting tasks can get their turn to use the resource. A first kind of multi-tasking (known as cooperative) permits releasing resources only to the tasks belonging them. The issue with this kind is low efficiency. A common alternative to this strategy is pre-emptive multitasking.

Pre-emption is the ability of an operating system (RTOS) to pre-empt or interrupt the currently scheduled task in favour of another task. In this case the context switch does not depend on the active task but on the RTOS. Making a multitasking system pre-emptive is positive as it allows the RTOS to guarantee that each task accesses the shared resources, but it has also some drawbacks that should be addressed to properly design a system.

The most dangerous implication of pre-emption is the race condition. A race condition is in general the result of nondeterministic ordering of a set of events [17]. In computing, a race condition is an undesirable situation that occurs when a task attempts to perform two or more operations at the same time, but because of its nature, the operations must be performed in the proper sequence, otherwise the race condition entails negative consequences such as data corruption. In non pre-emptive systems, there is no risk of data corruption as only one task can access a resource until completion. But in pre-emptive systems, the context switch forced by the RTOS may derive in the usage of corrupted data by some task.

The most common way to handle race condition implications in pre-emptive systems is the mutual exclusion of resources. Mutual exclusion takes place when only the active task can access the resource at a given time. When a resource is protected for mutual exclusion, the rest of tasks are paused until the active task finishes to use the resource. The typical mechanisms to implement mutual exclusion are the binary semaphores and mutexes. Although it seems that the most severe problem associated to the race condition is solved with the mutual exclusion mechanism, some additional problems arise when using this technique, particularly: Starvation, Deadlock and Priority Inversion (Priority Inversion). All of them lead to execution time penalties and are very difficult to detect and prevent unless a proper architecture is created to handle them.

	Deadlock	Starvation	Priority Inversion
Mutual Exclusion	Necessary	Necessary	Necessary
Hold & Wait	Necessary	N/A	N/A
Non-Preemption	Necessary	Necessary	Necessary
Circular Wait	Necessary	N/A	N/A
Priority	N/A	Necessary	N/A

Table 1: Concurrency Factors

The factors affecting these problems (see Table 1) are:

1. Mutual exclusion condition: When two or more tasks are trying to use a shared resource, and a mutex for this resource is activated, only one task can use the resource and the rest must wait until the active task releases the resource.

2. Hold-and-wait condition: Tasks already holding resources are permitted to request new resources.

3. Non pre-emption condition: This condition takes place when only the task holding a resource can release it.

4. Circular wait condition: It happens when tasks are in a circular chain and waiting for a resource held by the next task in the chain.

5. Priority factor: High priority tasks lock lower priority ones to access resources.

5.1 Deadlock

Formal definition of deadlock is: "A set of tasks is deadlocked if each task in the set is waiting for an event that only another task in the set can cause." [17]

In computing world, deadlock refers to a specific condition when two tasks are each waiting for the other to release a resource, or more than two tasks are waiting for resources in a circular chain.

According to Coffman [5], the four necessary and sufficient conditions for deadlock are: Mutual Exclusion, Hold and Wait, Non Pre-emption and Circular Wait.

The simplest example of process deadlock can be represented by two processes (A and B) trying to access two shared resources (R and S).

In order to highlight the deadlock, Holt's graphs [12] can be used to show the mutual dependency (cycle) of both processes.

Deadlock situation happens when A holds S, B holds R, A tries to acquire R and B tries to acquire S. If R and S are released before they are requested by A and B respectively, no deadlock happens.

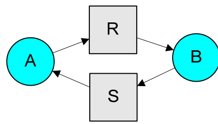

Figure 2: Holt's Graph

There are essentially three design decisions that may influence the four necessary and sufficient conditions for deadlock (see Figure 3): resource constraints, coordination protocol and process dependency. A resource constraint is an intrinsic characteristic of the resource that may affect its synchronisation behaviour (i.e. RAM memory is a preemptable resource). The coordination protocol describes the way a resource may be accessed by different tasks regarding, for instance, the exclusion or task ordering policy of a coordination mechanism. This protocol defines in fact the coordination mechanism operation. The task dependency represents the need of some tasks to use the computation results from other tasks.

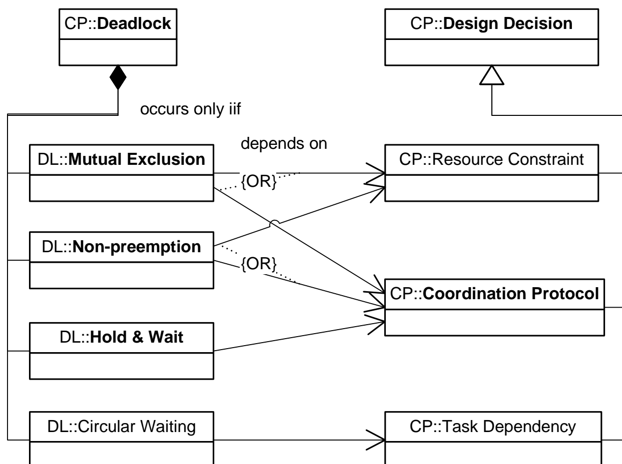

Figure 3: Deadlock Characterization

Circular waiting depends essentially on the tasks interdependency. Tasks must be in a dependency cycle to have a circular waiting. Hold and wait condition depends on the coordination protocol as it describes the way tasks are permitted to access resources. Non-preemption and mutual exclusion conditions may depend on resource constraints and on the coordination protocol.

5.1.1 Deadlock Assessment in PPOOA

Figure 4 represents a set of tasks and resources with a characteristic deadlock risk.

In PPOOA, tasks are represented through architectural elements of the type "Process" and resources in the diagram are represented by the abstraction "Structure". In the example, resources are protected by semaphores to assure mutual exclusion. This protection involves the first condition for deadlock. Circular waiting condition is represented in the diagram through a dependency cycle. In this case, tasks D, E and G conform a cycle highlighted in red. The rest of tasks in the diagram do not involve any cycle.

Non-preemption condition is implicit in the semaphore coordination protocol.

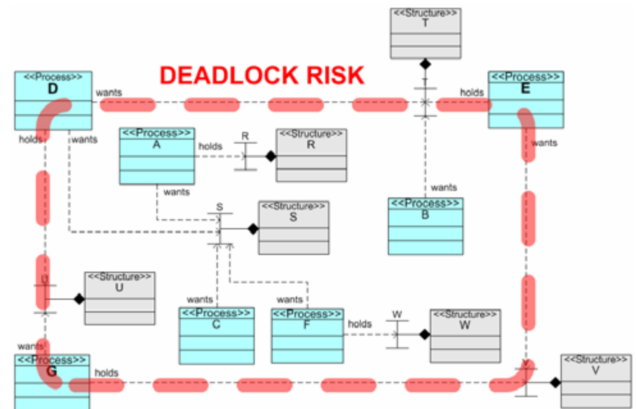

Figure 4: Deadlock in a PPOOA Architectural View

Structural diagram must be complemented with a CFA diagram (Figure 5) to evidence that hold-and-wait condition takes place. In the diagram it is necessary that the tasks in the cycle (D, E and G) hold and want the resources in sequence. This sequence involves that each task in the cycle is waiting and holding the resources.

For the purposes of this work the most relevant evidence of deadlock risk is the cycle detection in the structural diagram. If no cycle exists in this kind of diagrams, no deadlock may occur. The proposed characterization takes this fact into account to check first for cycles. In real world designs the identification of cycles is not so obvious as in the example. For this reason cycle detection must be implemented in an algorithm.

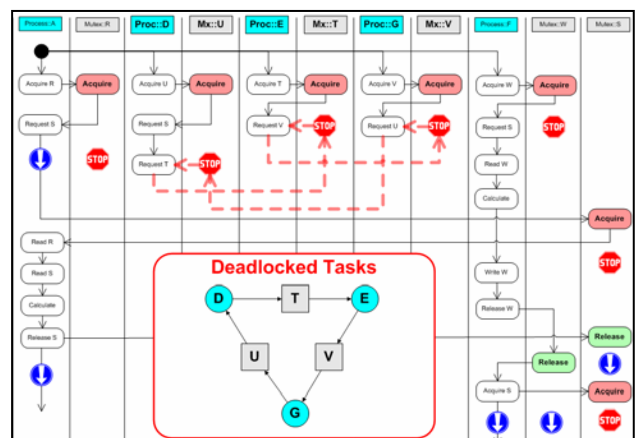

Figure 5: Deadlocked Processes in a CFA

The algorithm outline proposed to assess deadlock risk in PPOOA models is:

1. Search dependency cycles in architectural diagrams with protected resources.

2. If a cycle exists, check the CFAs to find sequence protocol information about the type of dependency of each task.

3. When each task in the cycle holds at least one resource and wants another, then deadlock may occur.

4. Assign a percentage value to the risk of deadlock based on the amount of dependency cycles detected. If no cycle is detected, then deadlock risk is zero.

5.2 Starvation

Starvation is a multitasking problem, where a task is perpetually denied to access the resources it needs for its execution [6].

Starvation is similar in its effects to deadlock, as the consequence is that a task does not run. In the case of deadlock the cause of this situation is task interdependence. Nevertheless, starvation occurs when a high priority task holds a resource other with lower priority needs. If no external event happens the high priority task may not release the resource and the low priority one remains waiting indefinitely. The problem now lies on the scheduling policy. Scheduling is supposed to allocate resources so that no task lacks necessary resources perpetually. But a wrong allocation policy may cause task starvation. For this reason it is necessary to detect the occurrence risk of this problem.

This situation may happen if there are several tasks of high priority that use one protected resource and just one of low priority depending on it. The low priority one may remain indefinitely waiting because the semaphore shall never let it run, due to higher priority tasks in the waiting queue.

5.2.1 Starvation Assessment in PPOOA

An algorithm outline to assess starvation in PPOOA models is proposed here:

1. Check the priority values of each task in a PPOOA static model and consider those protected resources where there are at least one low priority and one high priority task trying to access them.

2. Check if all the semaphores protecting the identified resources (in the CFAs) have a Release activity per each Acquire activity. If not, a potential risk of task starvation exists.

3. Assign a percentage value to the starvation risk according to the number of resources with high and low priority tasks and with no releasing activities in their CFAs, with respect to the total number of resources in the model.

5.3 Priority Inversion

Conceptually, priority inversion occurs when a low priority task holds permanently a shared resource needed by another high priority task.

A practical definition of priority inversion is [3]: “An unwanted software situation in which a high-priority task is delayed while waiting for access to a shared resource that is not even being used at the time. For all practical purposes, the priority of this task has been lowered during the delay period. Priority inversion arises when a medium-priority task pre-empts a lower priority task using a shared resource on which the higher priority task is pending. If the higher priority task is otherwise ready to run, but a medium-priority task is currently running instead, a priority inversion is said to occur.”

Priority inversion is not a problem itself, because the immediate consequence is just that priority precedence of tasks is temporarily inverted. The real problem related to priority inversion takes place when the task under inversion must meet a critical deadline. This may cause a catastrophic system failure and for this reason it is important to identify potential occurrence of such problem to be mitigated at early design stages.

Typical workarounds to mitigate priority inversion deal with the temporary modification of task’s priority while accessing protected resources. There are classically three priority allocation protocols that handle priority inversion issue: Priority Ceiling Protocol, Priority Inheritance Protocol and Highest Locker Protocol [7]. The objective of such protocols is not avoiding priority inversion to happen, but keeping it bounded to assure that high priority tasks meet their critical deadlines.

Priority inversion is an execution time problem very difficult to identify in design. Regarding the fact that only structural information is considered in PPOOA static diagrams, the objective of this work is just to highlight the potential risk of priority inversion in preliminary outlines of an RTS architecture.

5.3.1 Priority Inversion Assessment in PPOOA

As priority inversion is a problem associated to relative priorities of tasks accessing to protected resources in execution time, the only information valuable to identify occurrence risk is the task priority attribute.

An algorithm outline to assess Priority Inversion potential risk in PPOOA models is proposed here:

1. Identify all the tasks accessing protected resources in the PPOOA structural diagrams (only those tasks with dependency relations on resource building elements, protected with semaphores are susceptible to be in danger to suffer priority inversion).

2. Check the priority values of the tasks with dependencies on common resources. If there are tasks with high and low priority values trying to access a shared resource, and simultaneously there are one or more tasks of medium priority running (not necessarily accessing the shared resource), there is a potential risk of priority inversion.

3. Assign a percentage value to the priority inversion risk according to the total number of resources with such tasks.

6. Engineering Guidelines

The objective of this work is to propose mechanisms to detect occurrence possibility of the problematic situations described in this paper with no need to execute software or to specify execution times.

Once detected the problems, some engineering guidelines are proposed to reduce the likelihood of such problems.

Some engineering guidelines to produce reliable system models, regarding the problems identified in this work, can be summarised as follows:

- Deadlock: Avoid cycles when protected resources are shared by several tasks concurrently.
- Starvation: Avoid non-released semaphores in CFAs describing the behaviour of components including protected resources used by tasks of different priorities.
- Priority Inversion: Avoid many tasks accessing common resources with diverse priority values, specially medium priority.
- Cohesion: Break down each building element into a coherent set of components. Avoid complex components within each building element.
- Coupling: Avoid many dependency relationships on few building elements. Appropriately distribute responsibilities among them.
- Hierarchy: Avoid dependency relationships among components at different hierarchy levels.

7. Supporting Tool

PPOOA architectural style is currently supported under Microsoft-Visio®. This tool is flexible enough to extend its functionality to support additional engineering features to assess the problems identified in this work.

The strategy selected was to use an XML export add-on to generate an intermediate file that contains the dependencies and additional time-related information necessary for the algorithms to assess the models. An example of such XML file is shown in Figure 6. This feature is also used to export PPOOA –Visio models to RMA tools for schedulability analysis (i.e., Cheddar RMA analyser [19]).

DOCTYPE cheddar
cheddar
processors
address_spaces
tasks
task (6)

	task_type	cpu_na...	address_spa...	name	capacity	()	policy	deadline	()	priority	()	()	p...	jitter
1	PERIODIC_TYPE	my_processor	my_address_space	Alarma_Sonora	1	0	SCHED_OTHERS	200	0	0	4	2	200	0
2	PERIODIC_TYPE	my_processor	my_address_space	Supervisor	1	0	SCHED_OTHERS	200	0	0	4	2	200	0
3	PERIODIC_TYPE	my_processor	my_address_space	Sensor	1	0	SCHED_OTHERS	150	0	0	4	2	150	0
4	PERIODIC_TYPE	my_processor	my_address_space	Dispositivo_Alarma	1	0	SCHED_OTHERS	200	0	0	4	2	200	0
5	PERIODIC_TYPE	my_processor	my_address_space	Gestor_Zona	1	0	SCHED_OTHERS	125	0	0	4	2	125	0
6	PERIODIC_TYPE	my_processor	my_address_space	Gestor_ES	1	0	SCHED_OTHERS	300	0	0	4	2	300	0

buffers
resources
resource (5)

	cpu_name	addres...	name	state	protocol	resource_used_by
1	my_processor	my_address_space	Registro	1	PRIORITY_CELI NG_PROTOCOL	resource_used_by; resource_... Supervisor 0 0
2	my_processor	my_address_space	Configuracion_Sistema	1	PRIORITY_CELI NG_PROTOCOL	resource_used_by; resource_... Supervisor 0 0
3	my_processor	my_address_space	Configuracion_Zona	1	PRIORITY_CELI NG_PROTOCOL	resource_used_by; resource_... Gestor_Zona 0 0
4	my_processor	my_address_space	Teclado	1	PRIORITY_CELI NG_PROTOCOL	resource_used_by; resource_... Gestor_ES 0 0
5	my_processor	my_address_space	Display	1	PRIORITY_CELI NG_PROTOCOL	resource_used_by; resource_... Gestor_ES 0 0

Figure 6: XML File Example

An HMI prototype of the engineering tool to support the automatic assessment of the problems is shown in Figure 7.

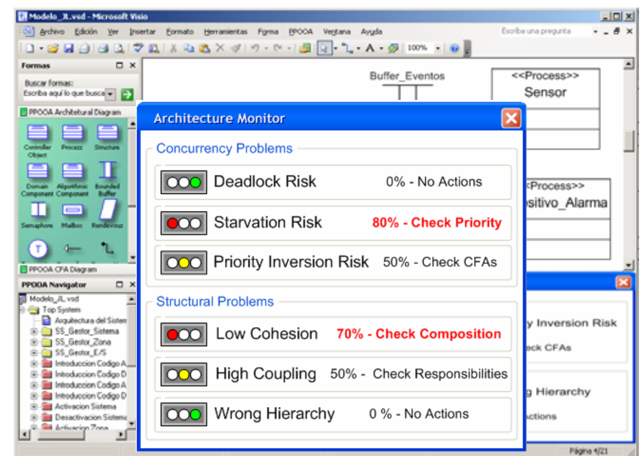

Figure 7: Architecture Monitor Tool

The Architecture Monitor is conceived as a tool to help system architects to assess the structural and concurrency robustness of their designs. But perhaps the most important aspect is that it enables them to compare the relative robustness of several design alternatives, in order to better make a decision on the most appropriate one.

8. Conclusions

A set of techniques of **static analysis** has been proposed to detect some characteristic problems of real-time systems, without performing any code execution or simulation. The objective is to early detect problems and to propose guidelines to modify designs in order to mitigate their occurrence.

This paper has highlighted the relevance of both structural and concurrency problems on the quality of real-time systems models. Main conclusions are:

- The traditional approach of trying to solve separately concurrency problems at late stages of the system life-cycle is counter producing in terms of rework effort and final product quality.
- The utilisation of a high level architecture modelling notation enables the identification of important concurrency problems at very early stages of the system design lifecycle to make decisions on them.
- The ontological approach proposed provides a better understanding of the typical problems focused, allowing the detection of many of them through the usage of static analysis techniques.

In order to effectively detect the problems identified in the ontology, a CASE tool shall be created to automatically analyse architectural models. This tool shall also implement the engineering guidelines to propose the designers the improvements in their models required to mitigate the problems addressed, and shall allow them to perform trade-off analysis among different architectural solutions.

9. References

[1] Alves, M., Dantas, C., Arai, N. and Silva, R.: "*A Topological Formal Treatment for Scenario-based Software Specification of Concurrent Real-time Systems*". International Conference on Software & Systems Engineering. Paris (France). December 2007.

[2] ARTiSAN Software Tools, Inc., Real-Time Studio, http:// http://www.artisansw.com/.

[3] Barr, M., "*Embedded Systems Glossary*", http://www.netrino.com/Publications/Glossary/.

[4] Chaki, S. and Sinha, N., "*Assume-Guarantee Reasoning for Deadlock*", SEI Technical Note, CMU/SEI-2006-TN-028, September 2006.

[5] Coffman, E. G., Elphick, M. J., and Shoshani A.: "*System deadlocks*", Computing Surveys, 3(2):67-78, June 1971.

[6] Dijkstra, E. W., "*Hierarchical ordering of sequential processes*", Dijkstra Manuscripts Archive, http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF, June 1971.

[7] Douglass, B. P., "*Real-Time Design Patterns – Robust Scalable Architecture for Real-Time Systems*", Addison-Wesley, 2002.

[8] Drake, J.M., González-Harbour, M., Gutiérrez, J.J. and Palencia, J.C. "*Description of the MAST Model*". http://ctrpc17.ctr.unican.es/mast.

[9] Egyed, A. "*Instant Consistency Checking for the UML*", 28th International Conference on Software Engineering, Shanghai, May 2006.

[10] Fernandez, J.L., "*An Architectural Style for Object-Oriented Real-Time Systems*". Fifth International Conference on Software Reuse. IEEE. 1998.

[11] Fernandez, J.L. and Monzon, A.: "*Extending UML for Real-Time Component Based Architectures*". International Conference on Software & Systems Engineering. Paris (France). December 2001.

[12] Holt, R. C., "*Some Deadlock Properties on Computer Systems*", ACM Computing Surveys, Vol. 4, No. 3, pp. 179-196, September 1972.

[13] Huang, J, Voeten, J and Corporaal, H., "*Predictable real-time software synthesis*", Springer Verlag, Real-Time Syst (2007) 36: 159–198, March 2007.

[14] Oquendo, F., "*Formally Modelling Software Architectures with the UML 2.0 Profile for π-ADL*", ACM SIGSOFT Software Engineering Notes, Vol. 31, No. 1, pp. 1-13, January 2006.

[15] PPOOA-Visio, http://www.ppooa.com.es/.

[16] Tanguy, Y., Gérard, S., Radermacher, A. and Terrier, F., "*Model driven engineering for embedded real-time systems*", 3rd European Real-Time Congress (ERTS), Toulouse, January 2006.

[17] Tanenbaum, Andrew S.: "*Modern Operating Systems*", 2nd edition, Prentice Hall, 2001.

[18] Telelogic, Inc., Rhapsody System Designer, http://www.telelogic.com/products/rhapsody/.

[19] The Cheddar Project, Cheddar RMA Analyser Tool, http://beru.univ-brest.fr/~singhoff/cheddar/.

10. Glossary

AADL: Architectural Analysis and Design Language

CASE: Computer Aided Software Engineering

CFA: Causal Flow of Activities

CPU: Central Processing Unit

HLP: Highest Locker Protocol

HMI: Human Machine Interface

MDE: Model Driven Engineering

PCP: Priority Ceiling Protocol

PIP: Priority Inheritance Protocol

PPOOA: Pipelines of Processes in Object Oriented Architectures

RMA: Rate Monotonic Analysis

RTOS: Real-Time Operating System

RTS: Real-Time System

UML: Unified Modelling Language

XML: Extensible Mark-up Language