Application of Deadlock Risk Evaluation of Architectural Models

Antonio Monzón [1,*,†], José L. Fernández [2] and Juan A. de la Puente [3]

[1] *Airbus Military, John Lennon Av., Getafe 28906, Spain*
[2] *Industrial Engineering School, Technical University of Madrid (UPM), Jose Gutierrez Abascal 2, Madrid 28006, Spain*
[3] *Telecommunications Engineering School, Technical University of Madrid (UPM), Avenida Complutense 30, Madrid 28040, Spain*

SUMMARY

Software architectural evaluation is a key discipline used to identify, at early stages of a real-time system (RTS) development, the problems that may arise during its operation. Typical mechanisms supporting concurrency, such as semaphores, mutexes or monitors, usually lead to concurrency problems in execution time that are difficult to be identified, reproduced and solved. 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. This paper aims to present the results of a research work oriented to the development of the tool called 'Deadlock Risk Evaluation of Architectural Models' (DREAM) to assess deadlock risk in architectural models of an RTS. A particular architectural style, Pipelines of Processes in Object-Oriented Architectures–UML (PPOOA) was used to represent platform-independent models of an RTS architecture supported by the PPOOA –Visio tool. We validated the technique presented here by using several case studies related to RTS development and comparing our results with those from other deadlock detection approaches, supported by different tools. Here we present two of these case studies, one related to avionics and the other to planetary exploration robotics. Copyright © 2011 John Wiley & Sons, Ltd.

Received 17 January 2011; Revised 9 July 2011; Accepted 12 July 2011

KEY WORDS: deadlock; software architecture; MBSE; UML; concurrency

1. INTRODUCTION

Software architecture modeling is a relevant subject for the production of real-time systems (RTSs). The development of architectural analysis and modeling languages in previous years has allowed representing both structure and behavior of such systems with less consideration to implementation details.

In this context, an architectural style is a consistent set of building elements with architecture rules for using them to create system models. The style well-formedness rules assure a minimum consistency level. Nevertheless, in addition to the notational or syntactic capabilities of a style, process and guidelines are also needed to help software architects produce feasible models concerning particular quality attributes (e.g. efficiency or safety). The Pipelines of Processes in Object-Oriented Architectures (PPOOA) architectural style [1] has been selected because it combines UML notation and model driven architecture (MDA) concerns, allowing for software architectural analysis and evaluation. In addition, this style is particularly useful to explicitly represent concurrency issues. This style has been recently selected as one of the reference MBSE methodologies by the OMG [2].

Deadlock is far from being a solved problem. Concurrency is in fact an open issue for many research and technical work [3–7] and of high interest to the industry, especially in the real-time

*Correspondence to: Antonio Monzón, Airbus Military, John Lennon Av., Getafe 28906, Spain.
†E-mail: antonio.monzon@military.airbus.com

embedded systems domain. Over the last three decades, different formal methods have been developed to specify and verify system properties. In this context, model checking [8] has become a reference discipline for such an approach. Its main goal is to build a finite model of a system and check that relevant properties are present in it. What is remarkable about this approach is that an exhaustive search of the state space is performed to ensure the fulfillment of some property. One of the properties particularly checked through model checking techniques applied to concurrent systems is deadlock absence [5, 6]. Additionally, for the classical state explosion problem (the main drawback of formal methods), from a practitioner point of view, other relevant issues are the intrinsic complexity of the modeling techniques and their applicability to large-scale RTSs. Industrial applications require simple and practical approaches to be easily adopted by practitioners.

In addition to deadlock detection and prevention, the third traditional strategy is deadlock avoidance. Under this category falls a successful mechanism that provides deadlock freedom. The group of resource access protocols known as priority inheritance [9] has as a major objective the resolution of priority inversion. As a collateral benefit, deadlock is also avoided if the priority ceiling protocol (PCP) or highest locker protocol is supported by real-time operating systems. The main issue with this mechanism is that few commercial real-time operating systems support these protocols and their *ad hoc* implementation by the developers is complex and therefore onerous. Furthermore, some authors have reported performance overheads derived from the utilization of these protocols [10].

This paper describes the implementation of graph theory to characterize the deadlock risk of an architectural model of an RTS. The objective is not to avoid or detect deadlock occurrence or to prove that a design is deadlock-free but to assess the risk of deadlock present in an architectural model. It is assumed that a model has an intrinsic risk of deadlock, which may involve further design decisions. The underlying idea of this approach is to make designers aware about this risk as early as possible and with the minimum analysis effort. The kind of model verification proposed in this paper is static analysis of the system platform independent model (PIM) (Figure 1).

Figure 1 shows the model driven architecture (MDA) models and the evaluation techniques associated with them. MDA provides four modeling levels: computation independent model (CIM), PIM, platform-specific model and implementation-specific model. High level models, such as CIM and PIM, do not represent execution platforms and technical details. As Figure 1 illustrates, when the underlying execution platform changes, the upper level models can be transformed to the new platform without any remodeling.

We describe a new complexity metric based on the properties of the architectural models of an RTS. The information considered in the characterization of deadlock risk comes from the following sources: cyclic complexity of the model and structural and dynamic deadlock patterns. Although the technique described is applicable to any kind of system, the deadlock topic is of special interest in the RTS domain.

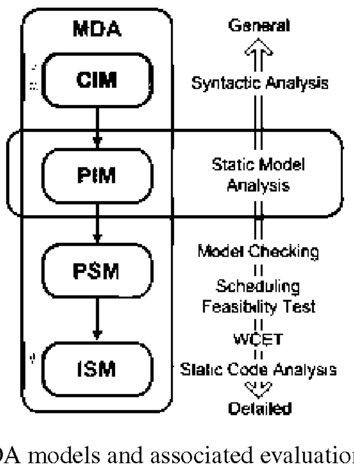


Figure 1. MDA models and associated evaluation techniques.

Although many computer-aided software engineering tools exist in the market to support software design activities, most of them focus only on the notational aspects with very little concern for engineering activities (e.g. alternative design tradeoffs). Several commercial tools [11, 12] support real-time characteristics but with no specific feature to analyze concurrency problems. Static analysis tools [13] partially cover this topic, but their main purpose is to analyze the quality of the software source code (not the models). Finally, model-checking tools [5] require the usage of detailed models with formal notations. A clear gap has been identified in current modeling tools to support the concurrency problem assessment at a high level of abstraction with semiformal notations (i.e. UML).

In Section 2, we briefly describe the state of the art in deadlock handling. The PPOOA–UML style is outlined in Section 3. The proposed characterization of deadlock and the tool created to automate this assessment (deadlock risk evaluation of architectural model (DREAM)) are presented in Section 4. The validation strategy is presented in Section 5, and Sections 6 and 7 present, respectively, two case studies from the avionics and robotics domains. Finally, Section 8 contains an outline of the decision process for the selection of the proposed characterization.

For conciseness reasons, most of the decisions made, their justification and other facts involved in the research work have not been described in this paper. Missing information is included in the doctoral dissertation in [14].

2. STATE OF THE ART

Table I shows a comparison among the various strategies and techniques currently available to address the problem of deadlock. It is assumed that all these techniques (except simulation) are applicable at abstraction levels near to implementation. Specific considerations for distributed systems are not taken into account as they do not add conceptual insights regarding the overall strategies shown in this paper.

Traditional prevention techniques are easy to implement but involve serious penalties for the use of resources and the time response of the system, requiring further specific implementations. These techniques are useful only when the criticality criteria prevail over efficiency (e.g. safety-critical systems).

The most effective avoidance techniques are indeed those derived from the priority inheritance protocols, especially the highest locker protocol, which has all the advantages of the PCP but without the drawback of development complexity. The most important problem with these techniques is practicality: they are supported only in some commercial operating systems and their implementation, based on COTS, is expensive. The alternative is *ad hoc* implementation on the specific platform, with the subsequent development and maintenance costs. This solution is an effective alternative at the implementation level but also has development costs (not negligible).

Formal methods are promising approaches, which have been very effective in space real-time applications (e.g. Mars Pathfinder), but they are computationally onerous and their applicability to large-scale industrial systems still has to be demonstrated. Although some strategies have emerged to address the state explosion problem (compositional techniques) [5, 15, 16], they have not shown sufficient evidence that can be applied to complex/large software applications. Besides their inherent complexity and their computational resource requirements, they also have serious practical drawbacks for their acceptance by the system designers. Given this approach, the natural tendency of the industry is to have simplicity and uniformity in the notation (e.g. UML) over the consistency of formal methods.

The most similar technique to that proposed in this paper is the simulation of concurrency in the Metropolis platform [17, 18]. From the detailed comparison of both approaches (documented in [14]), it can be concluded that the proposal of this paper has virtually all its advantages, in terms of level of abstraction and, in addition, increases the precision in the characterization of deadlock (through static and dynamic deadlock patterns) and eases its adoption by using a UML-based notation.

Another comparable technique is that proposed in [4]. The objective of this alternative approach is to use static analysis to reduce the overhead of dynamic checking for deadlock potentials. The main

Table I. Techniques for the management of deadlock (from [14]).

Strategy	Technique	Advantages	Drawbacks
Prevention	Concurrency prevented (cyclic executive) Negation of deadlock conditions (Havender)	• Deadlock is not possible • Very effective • Easy implementation	• Very inefficient • Can cause starvation • Low flexibility
Avoidance	Techniques derived from Banker's algorithm	• Negation of conditions is not required	• Complex evaluation of deadlock-free state • Low scalability (inflexible)
	Priority inheritance protocols: PCP y HLP	• Very effective • Priority inversion is also avoided	• Difficult implementation • Not supported by all RTOS • Potential negative impact on performance
Detection	• Resource Allocation Graphs (Holt) • State Graphs	• Easy implementation • No a priori restrictions	• Require execution-time monitoring • Complex recovery upon detection • Non-trivial protocols abstraction
	Formal Verification: • Model Checking • Petri Nets	• Very effective • Deadlock-freedom assurance	• State explosion problem • Reduced implementation flexibility • Complex modeling techniques
	Verification by simulation: • Metropolis-DSDG	• Allows conceptual assessment of architecture level models • Very flexible	• No standard notation • Requires specific simulation infrastructure

commonalties with DREAM are the usage of cycle detection in graphs to characterize deadlocks and the concept of deadlock potential to indicate risk of deadlock, independently of the real occurrence of it. The approach has the following important differences: the main difference is the level of abstraction, which is the source code with respect to PIM in DREAM; the second is the precision of DREAM deadlock characterization by the refinement through static and dynamic deadlock patterns, which allows reducing the number of false positives; and finally, the runtime overhead addressed by the authors is in fact not a problem in DREAM, because the architecture models used do not require any instrumentation on running programs.

Since the concept of deadlock was first presented, the entire research effort has been concentrated on those stages close to implementation, as highlighted in this section. The fundamental reason for this circumstance is that concurrency issues have been viewed as problems because of inadequate software coding practices (particularly data access and real time software). The leitmotiv of our approach is that concurrency issues are actually architectural problems to be evaluated from a system perspective. One argument supporting this idea is that the increasing complexity of systems, including distributed systems, has required an additional research effort and specific approaches to deal with deadlock in such environments. In the specific case of distributed systems, the management of local deadlocks does not guarantee global deadlock resolution. It is necessary to raise the level of abstraction to address such scope. This trend towards a wider scope is summarized in the systems engineering principle of 'system thinking'. Complex problems should not be addressed under a niche perspective but take into account global aspects of the system as a whole. The concurrency problems on systems of systems are one of the concurrency issues that are still unsolved.

In this direction, the MBSE paradigm promotes a higher level of abstraction in both model development and objective assessment of design alternatives (i.e. through model verification).

The approach selected in this paper is the evaluation of models, and therefore has nothing to do with the traditional mechanisms of execution time. The objective here is not to avoid deadlock occurrence, but to inform the software architects about its potential risk present in a model and which building elements are contributing to this risk, to help them take corrective actions at the level of architecture aiming to minimize the impact of this problem in further stages of development.

It is assumed that there are design decisions at the early stages of development that can lead to chronic deadlock problems during detailed design. For example, if a software architecture is designed containing many dependency cycles with interlocking patterns, it is quite likely that the detailed designs derived from this architecture will show the same or worse trend to deadlock. A design recommendation derived from this analysis is that the designer should minimize dependency cycles. It is also possible that an architecture holding a high deadlock risk thereafter implements mechanisms to avoid it (e.g. using resource access protocols, such as priority inheritance protocols). However, even in this case, the designer would be interested to know what is the deadlock potential of a model so he/she can make decisions at preliminary design stages or delegate them to the following levels of detail.

Deadlock is not an isolated problem but closely related to other concurrency issues, such as priority inversion. In [19], a taxonomy of concurrency problems was presented.

The approach followed in this paper proposes the assessment of concurrency properties of the system not by formal techniques but through metrics that do not require a deep knowledge of any specific modeling technique and have a simple physical interpretation.

3. PPOOA ARCHITECTURAL STYLE

A software architectural style encapsulates decisions about the building elements provided and emphasizes important constraints on the elements and their relations. The PPOOA architectural style provides building elements for RTS architectures, such as components and coordination mechanisms [1]. Constraints on building elements are represented in the PPOOA metamodel and by explicit guidelines or architectural heuristics. These guidelines not only represent the semantics of the style, but 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 adopted for PPOOA style requirements, specifically for modeling system responses [20].

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 visually represented in the architecture diagram.

The RTS behavior view is supported by the 'causal flow of activities' (CFA) representation. A CFA represents a behavioral 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 performing them.

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

- Task: PPOOA building element representing system threads or light processes. It may be periodic or aperiodic.
- Resource: Logical resources can be represented in PPOOA by domain components or structures. These building elements are abstractions of design classes and abstract data types, respectively.
- Semaphore/mutex: A pure synchronization mechanism. It is the PPOOA building element that supports the synchronization of tasks. Semaphores/mutexes are used to protect shared logical resources.
- Bounded buffer: A coordination mechanism representing a FIFO queue used to communicate asynchronously two tasks.

4. DEADLOCK CHARACTERIZATION

The process followed to create the characterization described in this section was iterative [14]. Three different characterization strategies were considered prior to the final decision. This decision was made based on the results extracted from four different case studies (two of them included in this paper). A summary of this decision process and a comprehensive rationale of the final characterization are provided in Section 8.

4.1. Deadlock definition

According to Coffman *et al.* [21], the four necessary and sufficient conditions for deadlock are: mutual exclusion, hold and wait, nonpreemption and circular wait.

Circular waiting depends essentially on the tasks' interdependency. Tasks must be in a dependency cycle to have a circular waiting. The hold and wait condition depends on the coordination protocol as it describes the way tasks behave to access resources. Nonpreemption and mutual exclusion conditions may depend on resource constraints and coordination protocols.

4.2. Deadlock in pipelines of processes in object-oriented architectures

A theoretical example is used to represent the structural model of a generic system to highlight the way deadlock is characterized in PPOOA. In Figure 2, tasks are represented by architectural elements of the type 'Process' and resources in the diagram are represented by 'Structure' elements of the PPOOA style.

In this example, the resources are protected by PPOOA semaphores/mutexes (coordination mechanisms in the style) to guarantee mutual exclusion. This protection involves the first condition for deadlock (mutual exclusion).

Circular waiting condition is represented in the PPOOA architectural view by a dependency cycle. In this case, tasks D, E and G conform to a cycle. The rest of the tasks in the diagram do not involve any cycle and therefore cannot contribute to deadlock risk. A nonpreemption condition is implicit in the semaphore/mutex coordination protocol and cannot be represented in this diagram.

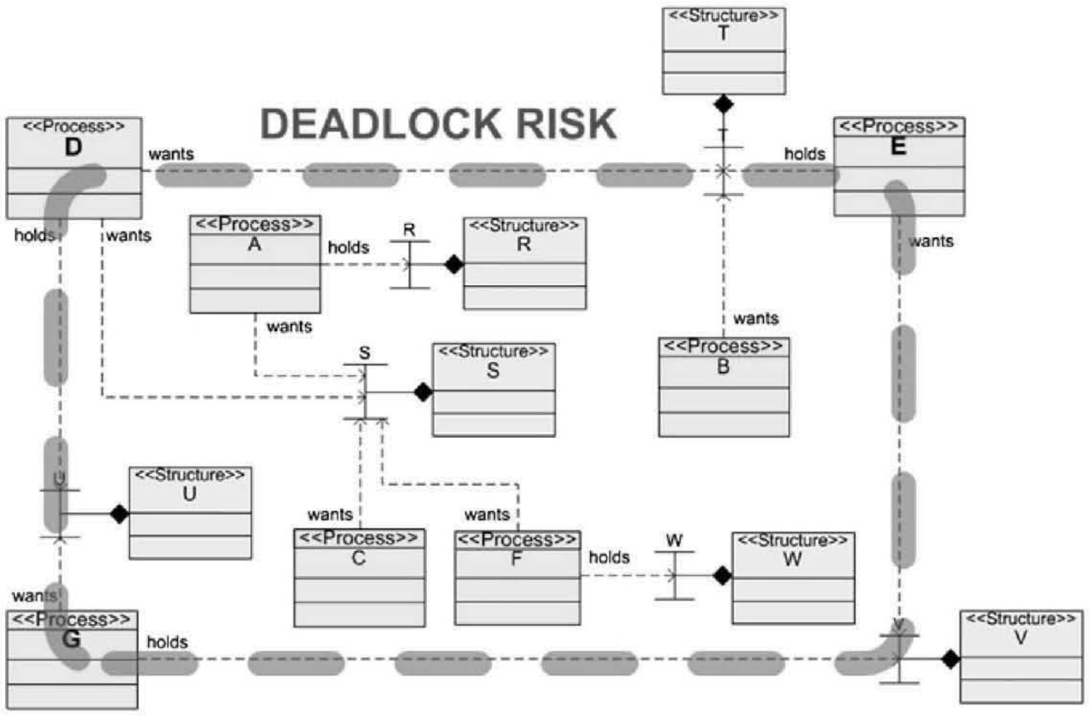


Figure 2. Potential deadlock in the PPOOA architectural view.

From the circular waiting condition, a first criterion for the characterization of deadlock risk can be extracted: identify all dependency cycles where two or more tasks are involved in an architectural model. The reason why two or more tasks are required is that at least two active elements must be competing for shared resources in a dependency cycle. The higher the cyclic complexity of the model, the higher its deadlock risk.

Cyclic dependency of several tasks is necessary but not sufficient for deadlock. For the purposes of this paper the rest of the conditions shall be summarized as follows: the tasks involved in a dependency cycle must be waiting for conditions depending on other tasks in the same dependency cycle.

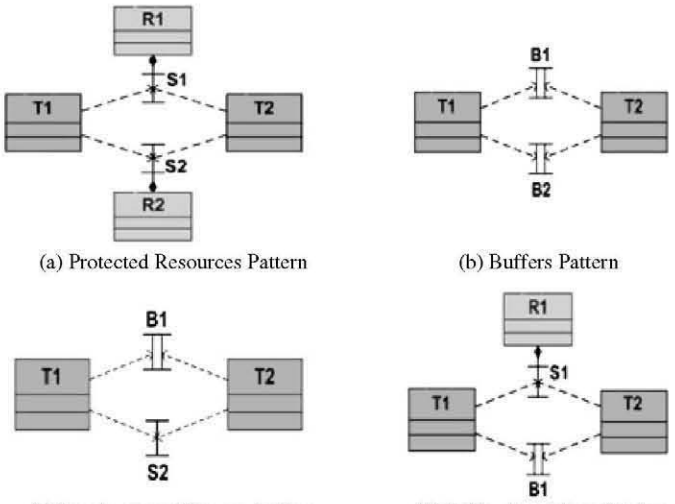
4.3. Deadlock characterization description

The approach presented consists in refining the cyclic complexity with additional criteria from the structural and behavioral views of an architectural model [14]. This refinement strategy is based on the identification of structural and behavioral deadlock patterns within the dependency cycles identified in the model. Deadlock risk is broken down into two factors: structural or intrinsic deadlock risk and behavioral or dynamic deadlock risk.

4.3.1. Structural deadlock risk. For the structural part of the deadlock risk, four deadlock patterns are proposed, considering the type of PPOOA constructive elements and the dependency relationships with others in the dependency cycles.

The first structural deadlock pattern (Figure 3(a)) considered in this characterization involves two (or more) tasks and two (or more) resources protected with respective semaphores/mutexes in the same dependency cycle. This is the classical deadlock case where several tasks are waiting for each other to use locked resources.

According to Sutter [7], protected resources are not the only architectural elements susceptible to cause waiting of tasks. Buffers can also introduce some risk of waiting when a task accesses them for some data and they are occasionally empty or full. For this reason, buffers can also be considered



(a) Protected Resources Pattern (b) Buffers Pattern (c) Synchronous Communication Pattern (d) Buffer-Semaphore/Mutex Pattern

Figure 3. Structural deadlock patterns.

as risky elements with respect to deadlock. The second structural pattern (Figure 3(b)) involves two (or more) tasks and two (or more) buffers in the same dependency cycle.

Synchronous message communication can be represented using a combination of a buffer of capacity one and a binary semaphore/mutex [22]. This intertask communication pattern involves task waiting; the producer task waits until the consumer task acknowledges message reception. Therefore, it can also be considered risky for deadlock. The third structural pattern (Figure 3(c)) involves two (or more) tasks, one buffer and one semaphore/mutex (not protecting resources) in the same dependency cycle.

Finally, for consistency with the first two patterns, a fourth pattern shall be considered (Figure 3(d)): two (or more) tasks, one (or more) buffer and one (or more) semaphore/mutex protecting resources in the same dependency cycle.

Each time one of these patterns is identified in a dependency cycle, DREAM records the architectural elements involved and marks them as risky elements from the structural point of view. The dependency cycle where they participate is therefore considered as risky from the structural point of view. The static deadlock risk is defined in this characterization as the total number of risky dependency cycles present in the architectural model.

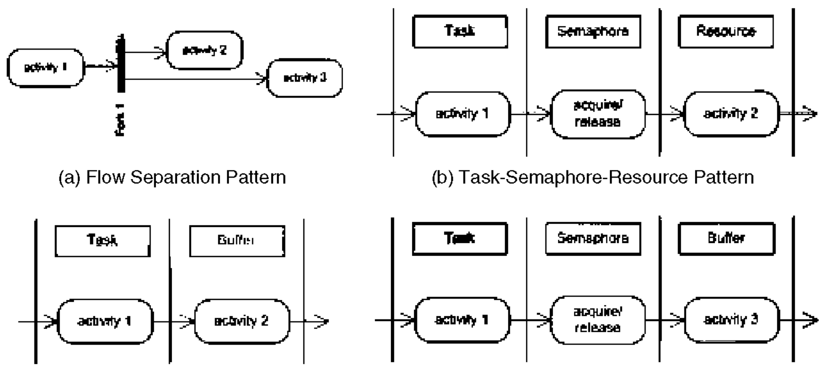
4.3.2. Dynamic deadlock risk. For the behavioral or dynamic part of deadlock risk, four additional behavioral patterns are considered in the behavioral diagrams of the PPOOA architectural style.

The first pattern (Figure 4(a)) represents the split of control flow in a CFA. This pattern involves execution parallelism of activities and therefore can be considered risky for deadlock.

The second pattern (Figure 4(b)) is represented by the sequence task–semaphore/mutex–resource in a CFA. This pattern is the behavioral counterpart of the first structural pattern. The third behavioral pattern (Figure 4(c)) corresponds to the second structural pattern and the fourth (Figure 4(d)) corresponds to the third structural pattern. The fourth structural pattern has no specific behavioral counterpart because it is in fact considered in the second and third patterns.

Each time one of these sequences is found in a CFA diagram of the system architecture, DREAM records the architectural elements involved and checks if they are included in the list of risky elements from the static point of view. In a positive case, the dependency cycle where they participate is therefore considered as risky from the behavioral perspective.

4.3.3. Deadlock risk interpretation. The overall deadlock risk is characterized by the total number of dependency cycles containing both structural and behavioral deadlock patterns. The interpretation



(a) Flow Separation Pattern (b) Task-Semaphore-Resource Pattern

(c) Task-Buffer Pattern (d) Task-Buffer-Semaphore Pattern

Figure 4. Behavioral deadlock patterns.

of these metrics is the following: whenever the architectural model of a system has a dependency cycle, there is potential risk of deadlock. This risk is confirmed when those dependency cycles contain structural deadlock patterns. If they do not contain structural deadlock patterns, they can be considered as (conceptually) deadlock-free, with the information available at this stage of the development. Finally, the risk is refined with behavioral deadlock patterns. Nevertheless, the absence of behavioral patterns does not guarantee deadlock absence, because this view can be lacking information of the implemented architecture (e.g. sometimes designers consider implicitly the participation of semaphores/mutexes in the access protocol to a protected resource described in a CFA).

As a summary, for the purposes of this paper, the most relevant evidence of deadlock risk is the existence of dependency cycles containing structural deadlock patterns in structural diagrams. If no risky cycle exists, no deadlock may occur. The characterization proposes a refinement of this parameter based on the sequence information available in the behavioral view of the architecture.

4.4. Deadlock risk evaluation of architectural models tool

The process we implemented in DREAM to assess deadlock risk can be summarized as follows:

1. Find all dependency cycles in architectural diagrams where two or more tasks are involved;
2. Search all the structural patterns present in the risky cycles previously identified;
3. Mark all the building elements involved in risky cycles with structural patterns as risky elements;
4. Assign a numeric value to the intrinsic deadlock risk: the amount of risky cycles containing structural patterns;
5. Search all the behavioral patterns present in the CFAs where the risky elements participate; and
6. Assign a numeric value to the behavioral deadlock risk: the amount of risky cycles containing behavioral patterns.

The first step of this process was implemented through the particularization of a cycle detection algorithm applicable to undirected graphs with a depth first search strategy. More details about this algorithm were presented in a previous paper [23].

The results from the cycle detection tool are:

- List of cycle sequences.
- List of elements involved in the cycles.

Once the cycles are identified, DREAM takes into account two additional contributions to deadlock risk: structural patterns and behavioral patterns. The structural patterns described in the previous section are searched in all the dependency cycles. Once a structural pattern is identified, the cycle, including all the elements participating in the pattern, is marked as risky. For the behavioral part, all the CFAs are scanned to search each of the patterns identified. This time only those elements considered risky from the structural point of view are considered in the search. Each time a behavioral pattern is found, the corresponding CFA is marked as risky. DREAM takes into account the overall behavioral deadlock risk if the elements participating in a risky cycle also participate in a behavioral pattern. In this case the cycle is also marked as risky from the behavioral point of view.

We used PPOOA-Visio tool [24], which is currently supported on top of Microsoft Visio. This tool is flexible enough to extend its functionality to support additional capabilities to assess the deadlock risk. The strategy selected was to use an XML export add-on to generate an intermediate file containing the dependencies and additional information necessary for DREAM to assess the models. This tool is conceived to help system architects assess the deadlock feasibility of their designs. However, perhaps the most important aspect is that it enables them to compare the relative deadlock risk of several design alternatives to better make and justify formal architectural decisions.

5. VALIDATION METHODOLOGY

In this section, we will explain the validation method used. During the development of the DREAM tool, traditional verification activities were carried out at different levels to demonstrate that it was error-free. The subject of this section is validation.

The validation was performed at two levels: first, corroborating that DREAM detects deadlock and, second, showing that the values of the selected parameters have physical meaning according to specific case studies. For the first validation level another tool was used (Cheddar [25]). For the second validation level three different case studies (containing known deadlocks) were analyzed, two of which are presented in this paper. In this context, the second case study is of particular interest for the validation of the proposed approach because it serves to illustrate the power of DREAM for detecting risk of deadlock and in addition shows that the results match the ones provided with a formal model assessment tool which requires much more detailed design information than DREAM.

Cheddar is not a tool specifically designed to detect deadlock. Its main function is to analyze the schedulability of an application's tasks, but it also incorporates several features to simulate real-time execution, and particularly deadlock.

Although the Cheddar (Figure 5) and PPOOA-Visio metamodels are different, there is a mapping between them [22] allowing the transformation of PPOOA models developed in Visio to be analyzed with Cheddar (through the exchange of XML files).

In Cheddar, a task is an element that represents a thread running on one processor. To define a task in Cheddar, it is first necessary to have defined a processor and a memory address space basis (the latter allowing the simulation of distributed and multiprocessor environments.)

A task in Cheddar is fully equivalent to a process in PPOOA-UML. The main caveat is that Cheddar requires tasks to be periodic to properly run the simulations. It is therefore necessary to make all tasks periodic in PPOOA-UML models before exporting them to Cheddar.

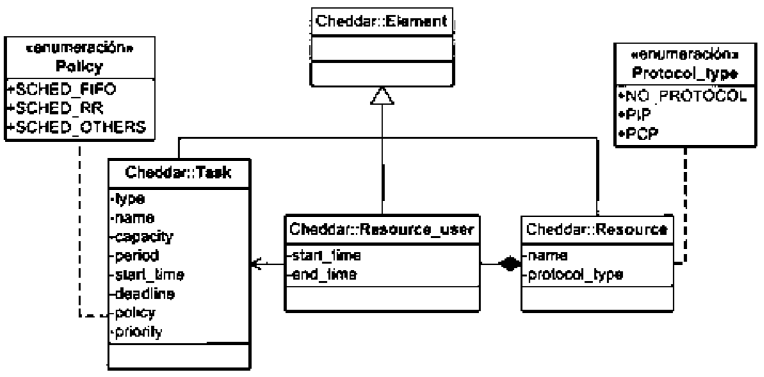


Figure 5. Cheddar metamodel [26].

Cheddar resources should be considered as shared resources. In Cheddar, pure coordination mechanisms (like semaphores/mutexes) are not considered. This coordination is implicit within the resource elements. Therefore, when a PPOOA-UML model contains a generic resource (domain component or structure) protected by a semaphore/mutex, its counterpart in Cheddar is represented by the corresponding resource it is protecting. The rest of resources are transformed directly from PPOOA-UML.

In the current mapping, PPOOA-UML to Cheddar buffers is mapped directly to the corresponding elements in Cheddar. This makes sense for analysis related to the buffers (e.g. feasibility analysis), but it represents an inconvenience for the characterization of deadlock proposed in this work. Cheddar buffers are transparent with respect to deadlock because the way it detects deadlock is associated only to shared resources. To provide the buffer elements of the PPOOA-UML models in Cheddar as potential risky elements regarding deadlock, it is necessary to assume them as shared resources in Cheddar.

The way of describing the interdependencies between tasks and resources in Cheddar is through the element type 'resource user'. Each resource can be used by one or more tasks. To represent that a task uses a resource is necessary to specify the access times of the task to the resource in units of time from the start of the task in its schedule. The access times of tasks to resources in Cheddar are defined by the start time and end time of resource use. The only condition imposed by Cheddar is that the finish time of use cannot be earlier than the start. The choice of these times is part of the detailed design of a model in Cheddar and crucially determines which tasks meet their deadlines. These inputs, along with the relative start on the tasks, are those that allow determining if a model contains deadlocks in Cheddar.

Cheddar detects deadlocks between tasks through the identification of task blockings during their wait for resources availability. To show a deadlock, it is necessary to first make a few preliminary assumptions:

- Planning policy of the processor is rate monotonic.
- All tasks have the same priority.
- There is no resource access protocol activated to explicitly avoid deadlock (resources have disabled the PCP).

It has been observed that these three conditions are necessary for Cheddar to detect deadlocks according to its detection mechanism.

Cheddar simulates the behavior of the model in terms of tasks accessing resources. If tasks are unable to complete their execution and the reason is that each one is blocking the other, the result is simulation halting. The outcome of this analysis is a report stating which tasks are blocked in which resources and the instant when simulation stops.

It is important to remark here an important limitation of Cheddar for deadlock detection. Once this tool detects a deadlock, the simulation stops and no additional deadlocks can be detected, which would have been revealed later in the simulation. This means that it is necessary to consider several scenarios with different access times to show all deadlocks present in a model.

The fundamental assumption followed at this point is that if there is deadlock risk in a structural model, it is possible to find values of timing parameters such that a simulation reveals the deadlock. At these values the risk of deadlock is shown. The failure to obtain a result of deadlock with a certain configuration of access times does not necessarily mean that the risk does not exist. In fact, most of the potential parameter settings are not deadlocked in the corresponding simulations and yet there is risk of deadlock. This circumstance can be considered a clear drawback of Cheddar as a means of detecting deadlock risk in front of DREAM.

6. AVIONICS SUBSYSTEM CASE STUDY

6.1. *Case study description*

A case study in the field of military avionics is presented here to illustrate the applicability of the proposed deadlock analysis and to validate DREAM.

One of the functions provided by the avionics embedded in military aircrafts is the automatic communications management. In particular, the function known as ‘automatic tuning of communication equipment’ (a.k.a. Autotuning) was selected to illustrate the approach of this paper. This function manages the frequencies of all on-board communication equipment (mainly radios and transceivers) to avoid unauthorized interception of communications by the enemy.

The architectural model for this function was split into three subsystems: autotuning plan (Figure 6), tuning configuration (Figure 7) and autotuning views management (this third subsystem was not relevant to this paper.)

The autotuning plan subsystem captures time, aircraft position and waypoint information from the avionics bus, represented in the architecture by the buffers B2, B3 and B4, respectively. Tasks

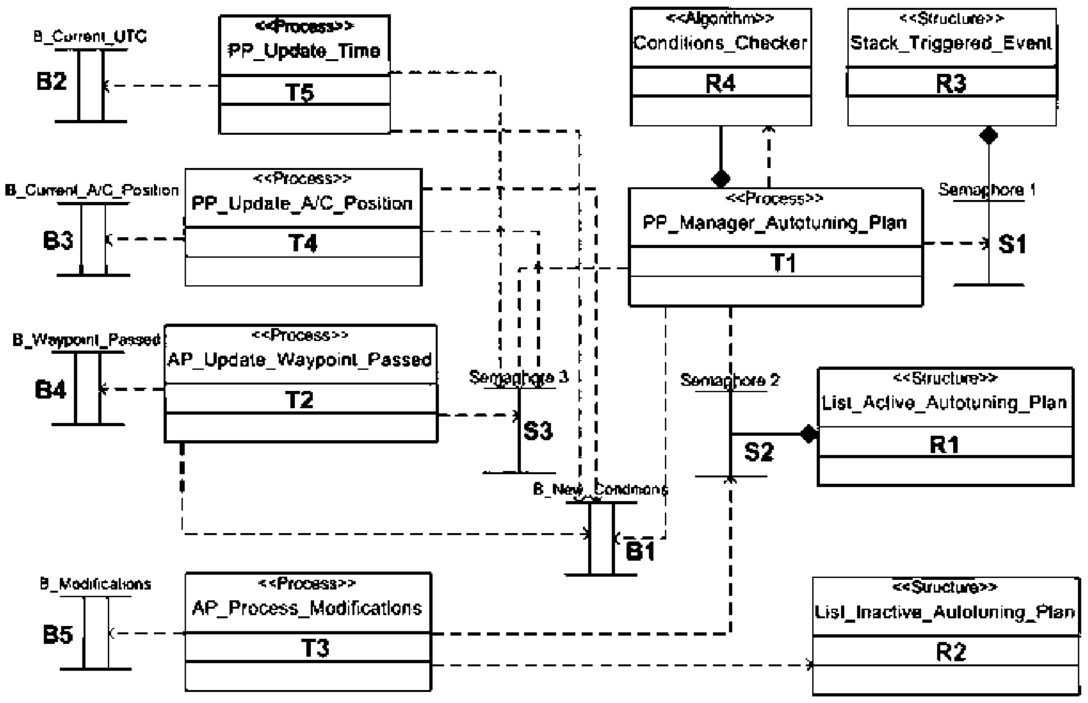


Figure 6. Autotuning plan subsystem.

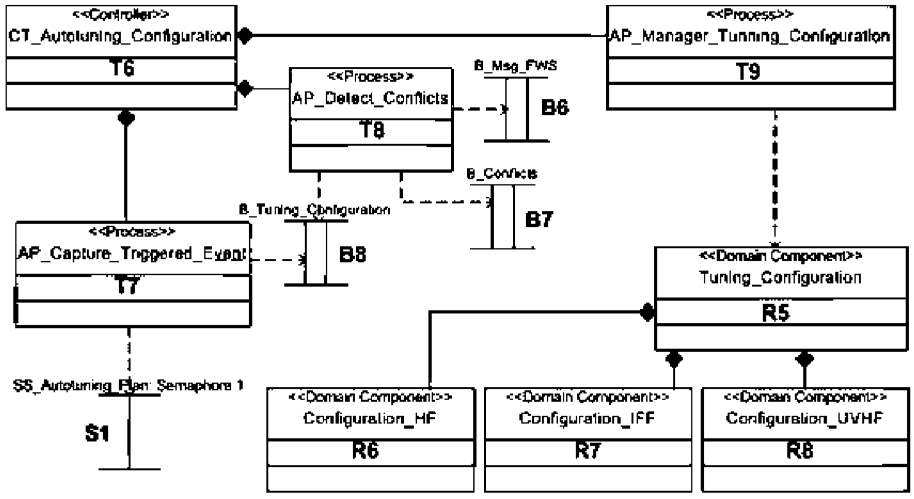


Figure 7. Tuning configuration subsystem.

T2, T4 and T5 put these data in the buffer B1. The periodic process T1 (management autotuning plan) is the main task of this subsystem and implements the execution of the autotuning plan. This task browses the list of planned events, implemented by resource R1, and compares them with the queue of events captured in real-time in the buffer B1. It is important to remark that B1 is combined with the binary semaphore/mutex S3 to implement a synchronous communication pattern to ensure that all the messages from the event handlers are received by the plan manager T1.

The tasks T1 and T3 can concurrently write on the shared resource R1. In terms of the problem domain, the pilot subsystem and the planning process can update the autotuning plan. For this reason, this resource is protected by a semaphore/mutex (S2).

Whenever the planning process T1 detects that a condition takes place as a result of periodic comparison, it pushes a new triggering event in the stack R3, which represents the command for the second subsystem to reconfigure the communication equipment frequencies.

The aperiodic task T7 captures the event from the protected stack R3 and the controller element autotuning configuration sends the command to set new tuning configurations to the communication equipment, represented in the diagram by resources R5, R6, R7 and R8. The process T8 detects conflicts in the configurations of different communication equipment and sends the warning messages through the corresponding bus ports represented by the buffer elements B6 and B7.

This architecture was selected to illustrate the handling of shared resources (it allows parallel execution of tasks), but it has some concurrency problems that shall be identified in the following section.

The behavioral part of the model is partially represented by the CFA 'triggered event' (Figure 8). This CFA can be interpreted as follows: at the arrival of the internal event 'new conditions' the sequence of activities executed within the system is:

1. T1 (autotuning plan manager) gets new conditions from B1;
2. As long as B1 is protected by S3, before accessing the buffer, the semaphore/mutex has to be acquired first. In execution time, T1 may be forced to remain waiting for the semaphore/mutex to be released;
3. Once the buffer is available, the message is received by T1;

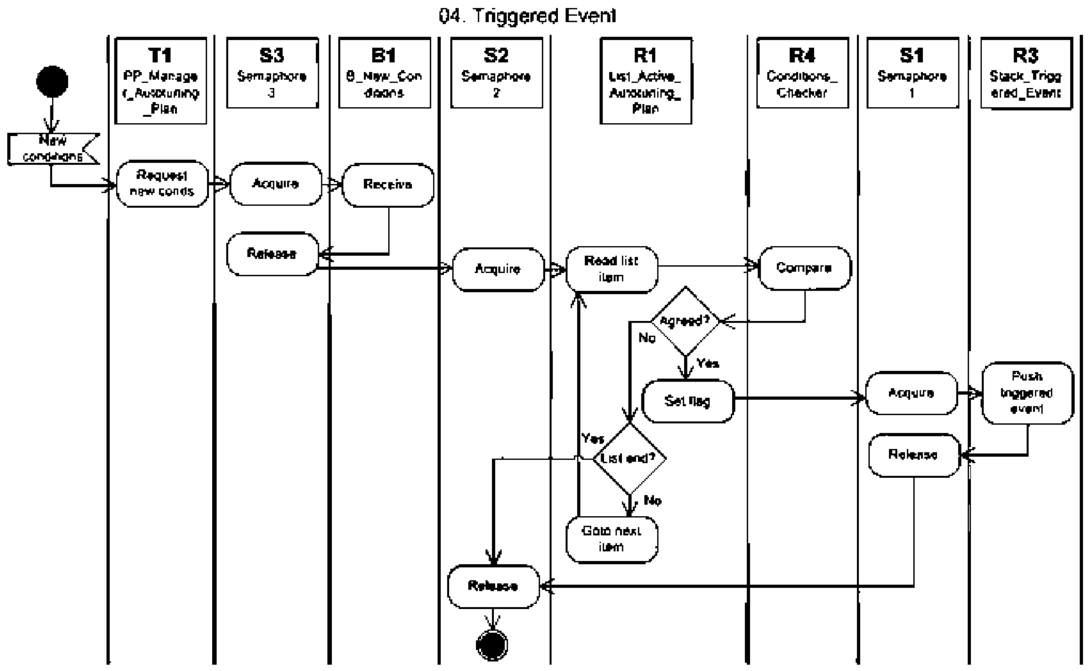


Figure 8. CFA triggered event.

4. Later, the semaphore/mutex is released and the list of events opened for reading the next event in it. This list is protected by the semaphore/mutex S2 and thus T1 can remain waiting until release;
5. T1 compares the information captured from B1 with the data in the list, transformed by the comparison algorithm R4 and if the result is positive, the event in the list is flagged and the command for the communication equipment reconfiguration is sent through the intermediate stack R3 (protected by the semaphore/mutex S1); and
6. Otherwise the next element in the list R1 is analyzed until the list ends.

6.2. Results discussion

The DREAM tool transforms the architectural diagrams described in the previous section into the aggregated graph shown in Figure 9. This undirected graph represents all the elements of the architecture and the relationships among them (regardless of PPOOA-UML stereotype).

This graph is used as an input by the cycle identification part of DREAM as a first step in the deadlock characterization. The results are shown in Table II.

According to this list, eight dependency cycles were detected. Out of them only six dependency cycles contain two or more tasks with structural deadlock patterns. Therefore, the structural deadlock risk (SDR) of this model is 6.

In Figure 9, the elements participating in the structural deadlock patterns are shown as rings. These elements shall be considered risky for deadlock. The risky elements detected are:

- Tasks:
  - PP_Manager_Autotuning_Plan (T1)
  - PP_Update_A/C_Position (T4)
  - AP_Update_Waypoint_Passed (T2)
  - PP_Update_Time (T5)
- Buffers: B_New_Conditions (B1)
- Semaphores/mutexes: Semaphore_3 (S3)

Once the information from structural diagrams is used, the rest of the information relevant to the deadlock analysis comes from the behavioral views or CFAs. In this case study, only three out of

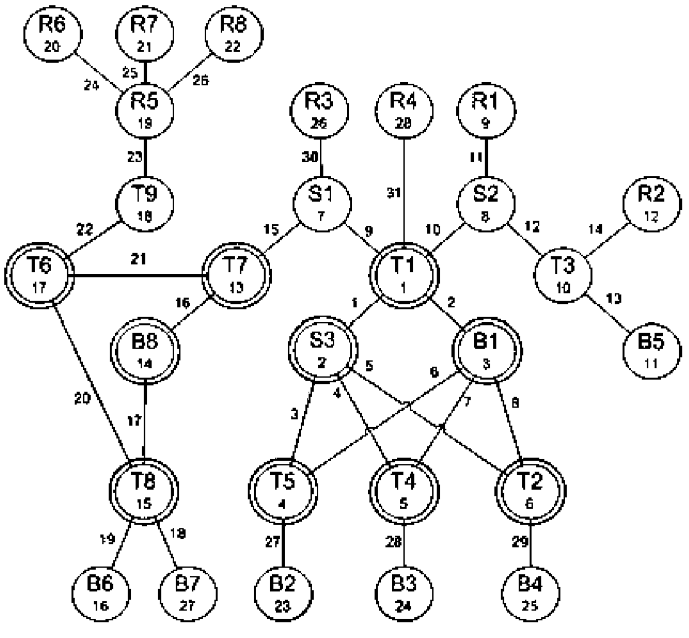


Figure 9. Structural graph.

seven autotuning CFAs included risky elements. The information generated by DREAM is shown in Table II.

From this information the following conclusions can be extracted:

- All the risky elements participate in the risky CFAs 1, 2 and 4.
- All the risky elements participate in at least one behavioral pattern in some risky CFA.
- All the cycles considered risky from the structural point of view are therefore also risky from the behavioral perspective. The dynamic deadlock risk (DDR) of this model is 6.

As a final summary, Table III shows the risky elements and the cycles and CFAs where they participate.

6.3. *Alternative design discussion and validation*

From the results of the previous section, the following conclusions can be extracted:

- The elements B_New_Conditions (B1) and Semaphore_3 (S3) are the most conflictive as long as both participate in all risky cycles and risky CFAs (Table IV).
- The rest of the risky elements each participate in three dependency cycles and one CFA, but all are related with the same pattern.
- The elements B1 and S3 are part of the synchronous communication pattern required to ensure that all the messages sent by the tasks T2, T4 and T5 are received by the plan manager task T1. This pattern is in fact the only source of deadlock risk identified in this case study.

A simple way to reduce the deadlock risk can be to change the current communication pattern among these tasks to an asynchronous one. This pattern involves removing the semaphore/mutex S3. The resulting model has no risky elements because the semaphore/mutex causing the risk is

Table II. List of dependency cycles detected.

Cycle	Elements	Deadlock risk?
1	PP_Manager_Autotuning_Plan, Semaphore_3, PP_Update_A/C_Position, B_New_Conditions	Yes
2	AP_Update_Waypoint_Passed, Semaphore_3, PP_Update_A/C_Position, B_New_Conditions	Yes
3	PP_Manager_Autotuning_Plan, Semaphore_3, AP_Update_Waypoint_Passed, B_New_Conditions	Yes
4	AP_Update_Waypoint_Passed, Semaphore_3, PP_Update_Time, B_New_Conditions	Yes
5	PP_Manager_Autotuning_Plan, Semaphore_3, PP_Update_Time, B_New_Conditions	Yes
6	PP_Update_A/C_Position, Semaphore_3, PP_Update_Time, B_New_Conditions	Yes
7	Conditions_Checker, PP_Manager_Autotuning_Plan	No
8	AP_Capture_Triggered_Event, B_Tuning_Configuration, AP_Detect_Conflicts, CT_Autotuning_Configuration	No

Table III. List of behavioral patterns detected.

#	Task-semaphore-buffer	T-B	T-S-R	**Split**
1	Analyse waypoint(AP_Update_Waypoint_Passed) -> Acquire(Semaphore_3) -> Send new condition(B_New_Conditions)	None	None	0
2	Analyse time(PP_Update_Time) -> Acquire(Semaphore_3) -> Send time(B_New_Conditions) Analyse position(PP_Update_A/C_Position) -> Acquire(Semaphore_3) -> Send position(B_New_Conditions)	None	None	1
4	Request new conds(PP_Manager_Autotuning_Plan) -> Acquire(Semaphore_3) -> Receive(B_New_Conditions)	None	None	0

Table IV. List of risky elements and their participation in risky cycles and CFAs.

		Risky cycles						Risky CFAs		
ID	**Risky element**	**1**	**2**	**3**	**4**	**5**	**6**	**CFA1**	**CFA2**	**CFA4**
1	PP_Manager_Autotuning_Plan (T1)	X		X		X				X
2	PP_Update_A/C_Position (T4)	X	X				X		X	
3	AP_Update_Waypoint_Passed (T2)		X	X	X			X		
4	PP_Update_Time (T5)				X	X	X		X	
5	B_New_Conditions (B1)	X	X	X	X	X	X	X	X	X
6	Semaphore 3 (S3)	X	X	X	X	X	X	X	X	X

missing. Nevertheless, this design decision is in conflict with the real-time requirement of ensuring that no message from the event managers is lost. For this reason, this alternative was discarded.

A second alternative design was proposed to fulfill two requirements: low deadlock risk and reliable message handling. The change consists in splitting the current buffer B_New_Conditions into three buffers, with each one communicating the pairs of tasks: T1–T2, T1–T4 and T1–T5. The results from DREAM show that no risk is present in this alternative model. Further analysis can be performed with temporal data to assess tasks overloaded with complementary tools like Cheddar.

The validation of the parameters proposed in this paper and the results of the different case studies used to derive them were performed with the aid of the schedulability analysis and simulation tool Cheddar. As mentioned above, Cheddar is interoperable with PPOOA–Visio [22]. A specifically developed Visio add-on implements the interoperability between PPOOA–Visio and Cheddar, and allows capturing the architecture model information generated with PPOOA as an XML file input to Cheddar. Execution time estimation was added to models to allow the simulation of execution in Cheddar showing when deadlock occurs.

6.4. *Validation with Cheddar*

Once this model is transformed to the XML file required by Cheddar, the simulation can be performed.

Given the characteristics of the model, it is necessary to perform several simulations to reveal all deadlocks present in the model. The reason is that four tasks (PP_Manager_Autotuning_Plan, PP_Update_A/C_Position, AP_Update_Waypoint_Passed, PP_Update_Time) are simultaneously accessing two resources (Semaphore_3, B_New_Conditions). According to the validation procedure described in the validation section, it is necessary to break down the test into six scenarios, each with two tasks accessing two resources, to demonstrate full deadlock detection coverage (containing the basic pattern of two tasks accessing two resources). In addition, it was necessary to consider the buffer 'B_New_Conditions' as a shared resource for the purpose of deadlock simulation with Cheddar. The results for the six scenarios are shown in Table V.

As can be seen from the results, the four conflicting tasks cannot complete their execution and six risky cycles are identified.

7. MARTIAN ROVER EXECUTIVE CASE STUDY

7.1. *Case study description*

This section describes an architectural design representing the control system of a planetary exploration rover. The system has been in operation in several NASA/JPL (National Aeronautics and Space Administration/Jet Propulsion Laboratory) missions and there is documented evidence of concurrency problems, such as priority inversion and deadlock. In [27], the authors explain how they used model checking to evaluate the occurrence of deadlock in this system. This concrete case study is considered particularly relevant because the detection of deadlock by DREAM with the same design solution is an explicit demonstration that it fits the purpose.

All the information used in this section has been gathered from the architectural solution described in several technical papers available on the website of the JPL [28–30] describing the K9 system.

Table V. Deadlock simulation results with Cheddar.

```
Scheduling simulation, Processor my_processor :
- Deadlock from simulation  :
   - Deadlock at time 2 :  PP_Update_AC_Position B_New_Conditions.
   - Deadlock at time 2 :  PP_Manager_Autotuning_Plan Semaphore_3.
```

```
Scheduling simulation, Processor my_processor :
- Deadlock from simulation  :
   - Deadlock at time 2 :  PP_Update_Time Semaphore_3.
   - Deadlock at time 2 :  PP_Update_AC_Position B_New_Conditions.
```

```
Scheduling simulation, Processor my_processor :
- Deadlock from simulation  :
   - Deadlock at time 2 :  AP_Update_Waypoint_Passed Semaphore_3.
   - Deadlock at time 2 :  PP_Update_AC_Position B_New_Conditions.
```

```
Scheduling simulation. Processor my_processor :
- Deadlock from simulation  :
   - Deadlock at time 2 :  PP_Update_Time Semaphore_3.
   - Deadlock at time 2 :  PP_Manager_Autotuning_Plan B_New_Conditions.
```

```
Scheduling simulation, Processor my_processor :
- Deadlock from simulation  :
   - Deadlock at time 2 :  AP_Update_Waypoint_Passed Semaphore_3.
   - Deadlock at time 2 :  PP_Manager_Autotuning_Plan B_New_Conditions.
```

```
Scheduling simulation. Processor my_processor :
- Deadlock from simulation  :
   - Deadlock at time 2 :  AP_Update_Waypoint_Passed Semaphore_3.
   - Deadlock at time 2 :  PP_Update_Time B_New_Conditions.
```

The K9 rover planetary exploration vehicle is a six-wheeled rocker bogey mounted on a chassis that incorporates the electronics and tools for scientific research and remote autonomous operation (not human operator dependent).

The architecture of the control system software of the rover consists of five types of modules (Figure 10): Device drivers, Resource managers, Processors and Data performance, Pilot subsystem and Executive.

Device drivers are the lowest level software elements. They directly control the rover hardware (engines, buses and cameras). Each device driver is controlled by a resource manager and cannot be commanded directly by the rover operator or by the Executive.

Resource managers act as intermediaries between the higher-level elements (Pilot subsystem and Executive) and device drivers. The components of this type are the base manager, the pan/tilt manager and the vision manager. Resource managers can be commanded directly by the operator using command messages or direct calls to component methods. The command messages are transmitted via a communications real-time bus, based on a datagram model.

Performance processors and data components include path planning, obstacle avoidance, visual tracking and stereo processing. These components cannot be directly commanded by the operator but only through the Pilot subsystem.

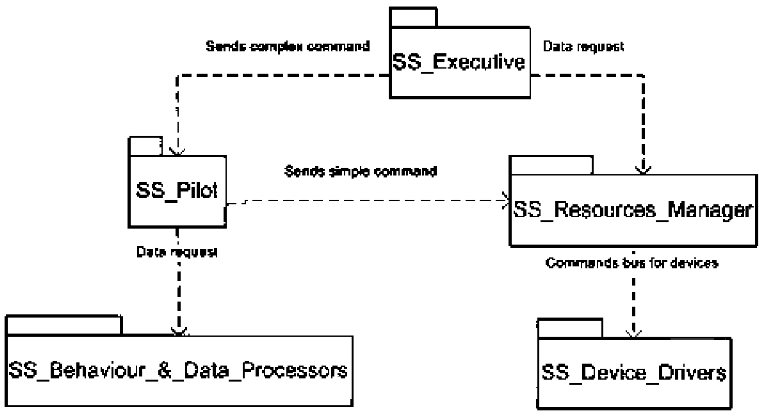


Figure 10. Subsystems of the rover control system.

The inputs to the obstacle avoidance module are a range map of the current environment, the desired target, the maximum allowed object size and the minimum spacing between obstacles. The output of this module is the safe driving direction.

The Pilot subsystem is a complex component that is responsible for decomposing/translating complex commands into specific commands to the resource managers and handling requests for information to parts of the behavior and data processor.

The conditional Executive subsystem is responsible for interpreting the command plans (described in a high-level language called CRL) coming from the ground control. The files written in this language are interpreted in a similar way as a compiler does.

The Executive checks the requirements on resources and their availability at runtime (from resources module), monitors the execution of the plan (checks that commands are fulfilled) and selects alternative branches of the plan according to the external conditions (the Executive subsystem can decide autonomously whether the situation is different from that originally planned).

The plan has a hierarchical structure. At each point of a plan branch, the Executive can have several selectable options. The Executive will choose the plan option with a higher value of expected utility. The utility value at each point of the plan is combined with a model of potential events to obtain the overall usefulness of each branch of the plan.

This decision mechanism works as a modern chess program does: it evaluates a branch of the tree and assigns a numeric value that represents how good the play would be chosen according to the possible combinations of estimated movements.

The initial estimate of the utility is carried out at the ground segment in accordance with the theoretical availability of time and resources. Such values are only estimates and actual data are known at runtime. To reduce uncertainties, the system is able to update the plan based on the actual information available regarding time and resources.

When the implementation of a plan fails in any of the actions, the Executive subsystem reacts by either ignoring the action or aborting and checking if there are alternative plans. In the second case, the Executive aborts the plan, leaving the rover in standby mode.

Because of the incomplete information available, only a partial view of the software architecture is provided. The architectural diagrams of the subsystems SS_Behaviour_&_Resource_Managers and SS_Executive are considered. The remaining parts of the system have been excluded from this analysis, but these have been taken into account through design decisions to avoid potential problems arising from the interaction with the missing parts (e.g. use of semaphores/mutexes to protect resources potentially available for other elements not considered).

The SS_Executive subsystem (Figure 11) is responsible for reading the command plan and interpreting complex commands to be translated into simple commands to be sent to the physical

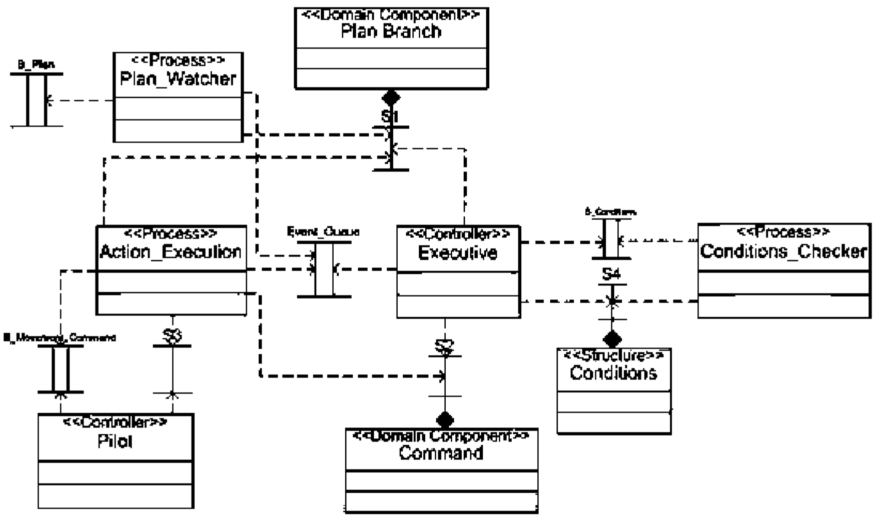


Figure 11. SS_Executive subsystem architecture.

device drivers. In the figure, the task in charge to receive the plan is Plan_Watcher. The plan is essentially a complex list of instructions in a particular format (instruction tree in high-level language CRL), which is represented in the model through a domain component. The reason for this selection (instead of a structure component) is that it can save the internal state (utility). The plan is an element that can be changed at runtime depending on the conditions and the utility factor. For this reason, it is protected by a semaphore/mutex (S1) to avoid simultaneous writing by the Executive and Action_Execution tasks. The plan is read sequentially by Plan_Watcher, which is performing the browsing of the branches of the plan. Each instruction is placed in the Event_Queue as a new command that is in turn read by the Executive, which is the main component of this subsystem. This task takes the last command in the Event_Queue and checks the available resources (represented by the structure element named Conditions), assesses whether they are sufficient for running the corresponding command and finally breaks down the complex command into simple commands (represented by the domain component Command in Figure 11). The reception of resource data must be conducted synchronously (messages cannot be missed because otherwise the Executive could be considering conditions different from the current ones and therefore its decisions may result in damage to the rover). For this reason it is necessary to use the synchronous communication pattern (Figure 3(c)) for the tasks Executive and Conditions_Checker (semaphore/mutex S4 and buffer B_Conditions). The domain component Command is protected by a semaphore/mutex to prevent other tasks (not shown in this partial view of the architecture) from writing on it simultaneously. The task Action_Execution is responsible for monitoring the implementation of simple commands generated by the task Executive. This monitoring task can read the plan and the Event_Queue, but its main functionality is to take simple commands generated by the Executive, send them to physical devices (represented in the diagram by the complex task Pilot) and check that these have been executed satisfactorily. Because messages between tasks Action_Execution and Pilot cannot be lost (which could mean, e.g. not performing obstacle avoidance actions), the synchronous communication pattern is used again (represented by the semaphore/mutex S3 and the buffer B_Movement_Command). Once execution confirmation of a simple command is received, the task Action_Execution updates the status of the command and instructs the Executive through the Event_Queue.

The pilot is a subsystem not described in this paper, but it interacts with the subsystem SS_Behaviour_ & _Resource_Managers shown in Figure 12. In this diagram, the pilot subsystem is represented by the complex task Pilot. The pilot subsystem receives simple commands from the

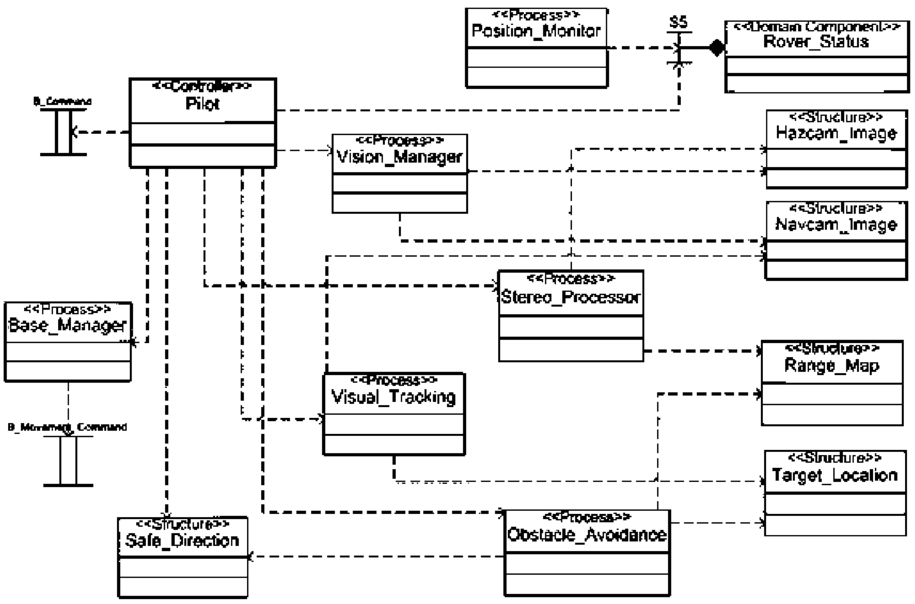


Figure 12. SS_Behaviour_ & _Resource_Managers.

SS_Executive and translates them in a coordinated manner to different physical devices. A part of the operation of this subsystem is described in Figure 13, but in this section, we justify the selection of the building elements and coordination mechanisms.

The Pilot subsystem has to compare the current rover position with the visual information received by the video capture devices. For this reason it must be coordinated with the position monitoring task (Position_Monitor). In this case we have chosen the semaphore/mutex S5 to prevent simultaneous writing on the domain component Rover_Status, which holds among other things, the rover geographic position. Vision_Manager task is responsible for the management of navigation cameras (NavCams) and alert cameras (Hazcams). The cameras are actually duplicated to provide stereoscopic images, which are represented in the model by the structures Hazcam_Image and Navcam_Image. The Stereo_Processor task processes the Hazcam_Image for the Obstacle_Avoidance task, responsible in turn for avoiding obstacles. The Visual_Tracking task is in charge of processing the Navcam_Image to generate the Target_Location structure, used by the Obstacle_Avoidance task together with the Range_Map to generate a safe direction (Safe_Direction) used by the Pilot subsystem to command the physical devices of movement (motor and wheels) represented in Figure 12 by the Base_Manager element.

The dynamic view of the architecture is represented by three CFA diagrams. Each of these CFAs represents the response that is triggered by the particular events within the system.

The CFA 1 (Figure 13) describes the behavior of the system in response to the arrival of a new command plan and describes the behavior needed to transform complex commands into simple ones for the subsystem Pilot. When the operator sends a command plan to the rover, the Plan Watcher reads the plan (previous acquisition of the semaphore/mutex S1) and sends its current branch to the Executive. The Executive reads the plan branch and reads the resource Conditions. If the resources are sufficient then the utility of the branch is calculated and the next command is processed.

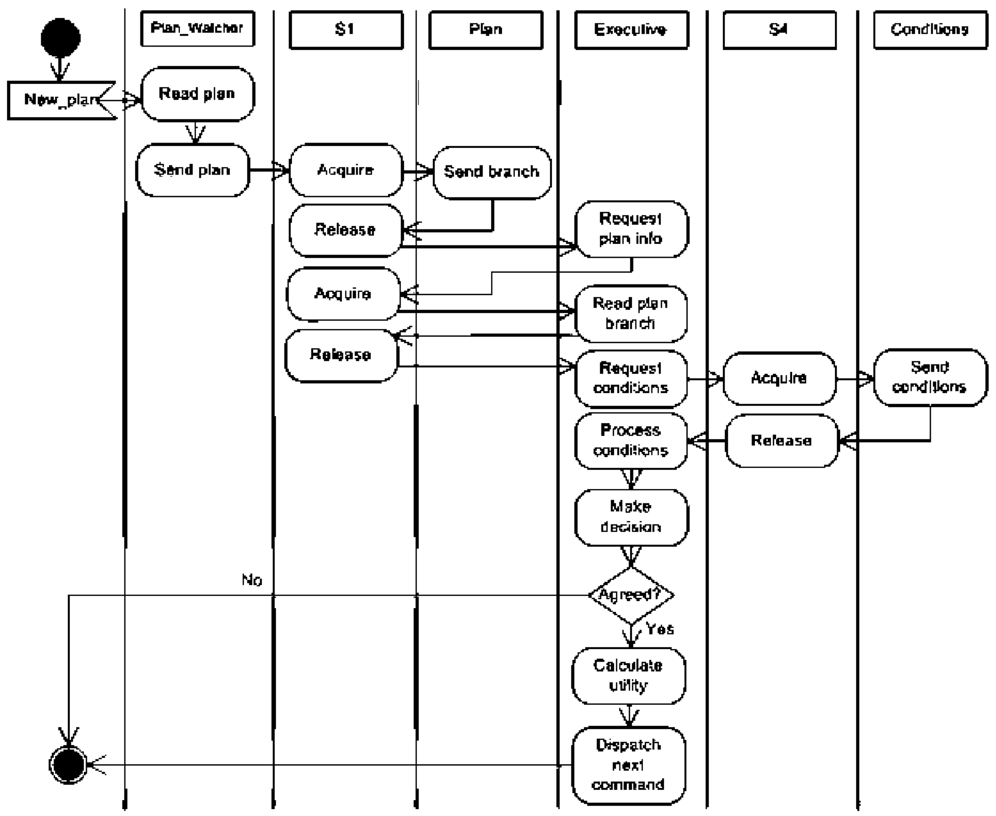


Figure 13. Transforming complex in simple commands.

The CFA (Figure 14) describes the behavior of the system responding to a particular command that is part of the basic operation of the rover: to go to a certain position while avoiding obstacles in the path.

The sequence of actions triggered by the command is described as follows:

1. The Pilot subsystem processes the current position of the rover, which is available in the Rover_Status element (which in turn is updated regularly by various tasks, such as Position_Monitor).
2. In the event that the current position does not correspond to the destination, then the rover continues moving to the destination.
3. The Pilot subsystem requests the Vision Manager to acquire two stereoscopic images of the front cameras (hazcams) and two more from the navcams.
4. Then the Pilot subsystem passes the hazcam images to the Stereo Processing element and it returns a ground level map.
5. Simultaneously, the Pilot subsystem passes the navcam images and an image of the destination to the Visual Tracking element.
6. The Visual Tracking returns the position of the destination, which in turn is used by the Obstacle Avoidance component together with the ground level map.
7. The Obstacle Avoidance element returns the safe direction, which is used by the Pilot subsystem to send the appropriate commands to the Base Manager to move the rover.

This last operation involves transformations to translate the direction into motion instructions for the actuators (rover control laws). This process is repeated periodically (internal periodic event) until the rover reaches the destination safely.

7.2. Results discussion

The results generated by DREAM can be summarized as follows:

- Twenty-eight dependency cycles containing two or more tasks have been detected.
- Only 8 out of the 28 contain some deadlock patterns. Therefore, the value for the static deadlock risk is SDR = 8.
- The risky elements identified are:

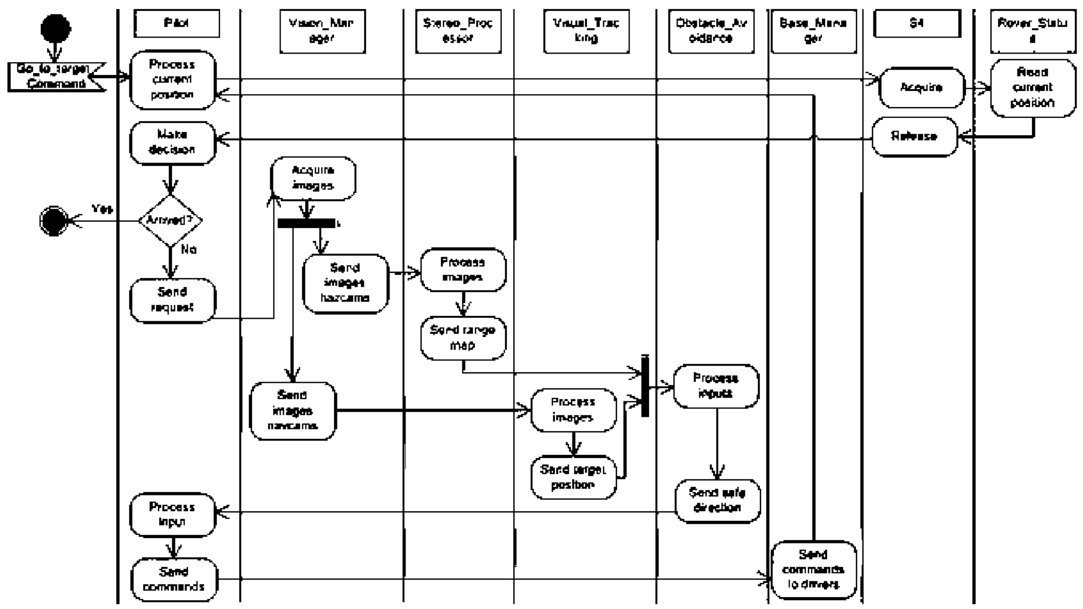


Figure 14. Go to a certain position while avoiding obstacles.

  - Tasks: Executive, Action Execution, Plan Watcher, Conditions Checker, Pilot
  - Buffers: Event_Queue, B_Conditions, B_Movement_Command
  - Semaphores/mutexes: S1* (protecting resource), S2* (protecting resource), S3, S4* (protecting resource)
  - Resources: Plan, Command, Conditions
- The risky elements detected from the static point of view are involved in CFAs 1 and 2. The CFA3 does not include any risky elements and therefore it does not contribute to the dynamic deadlock risk.
- Only the elements Executive, Plan Watcher, S1, S2, S4, Plan, Command and Conditions are involved in dynamic patterns. The remaining elements cannot be considered risky from the dynamic point of view.
- Only 7 out of the 8 cycles considered risky from the static point of view can be considered also risky from the dynamic point of view because they are composed of elements involved in dynamic deadlock patterns. The value of dynamic deadlock risk is therefore DDR = 7.
- The total number of dynamic patterns detected is 3.

In light of the results obtained, the following conclusions can be extracted:

- The most problematic elements of the model are Executive and Action Execution as they are involved in 6 out of 8 risky cycles. In addition the 'Executive' is involved in the two risky CFAs.
- The resource 'Plan', protected by the semaphore/mutex S1, is the next item on the list of problematic elements as it participates in 5 risky cycles. It is also involved in the dynamic patterns within the CFA1.
- The elements Plan Watcher, Event_Queue and Command are involved in 3 risky cycles.
- The remaining risky elements are less relevant.

7.3. Comparison versus formal methods

Giannakopoulou [27] showed the results of applying formal verification techniques to the specific case of the K9 rover executive. In the first section, they presented the software architecture of the rover executive and architectural decisions taken to improve its preliminary design. The main change in the modeling is the way to represent the system behavior; changing an approach based on state machines to another based on events responses. To improve the communication among the Executive subsystem and other elements of the architecture, they proposed the use of a FIFO queue of events (Event_Queue). Additionally, the designers decided to enable the simultaneous execution of threads for the monitoring task Action_Execution. These improvements introduce, however, a number of concurrency issues that are brought to light through the formal verification technique used by the authors.

They used the tool Labeled Transition System Analyzer (LTSA) [16] to carry out the verification of the improved model. This tool supports modular verification (compositional reachability analysis) [15] to analyze the characteristics of the system, modeled with Kripke structures (in this case using the formal language FSP) and incrementally applied to parts of the system. This technique significantly reduces the number of states and transitions to be checked with respect to the monolithic application of traditional model checking techniques (as evidenced in the comparative tables of the paper's discussion).

They proposed the evaluation of a series of system properties that are relevant to assessing the quality of models, among which are deadlocks and race conditions.

From the incremental evaluation of the first property ('Absence of local and global deadlocks') they reported the detection of a deadlock involving the tasks and Action_Execution and Executive. The counterexample derived from the formal technique (assume-warrantee) was a behavior in which the Executive actively tried to stop the action without knowing whether or not it was already completed, while Action_Execution was waiting to start a new action (deadlock situation). They proposed a solution to tackle this problem consistent with the level of design detail they were using.

From the results shown in Table VI, the first conclusion is that the most problematic elements are Executive and Action_Execution. This result corresponds to the information reported by the authors in [27]. This can be considered as evidence that our approach works correctly and is able to identify deadlock risk where other much complex tools handling more implementation details have yielded positive results in this regard. Both elements, Action_Execution and Executive, participate in 4 out of the 8 risky cycles detected by DREAM.

The remaining deadlock risks identified by DREAM (4 additional risky cycles) can be considered as deadlock potentials that have not materialized. The fact that they were not detected by the LTSA tool does not mean that they are false positives but only potential risks to be taken into account when refining the model with the data available at this level of detail. In this sense, it can be considered that DREAM is a better tool for detecting deadlock threats than LTSA when limited design information is available.

8. CHARACTERIZATION RATIONALE

The objective of the deadlock characterization proposed in this paper is to provide a metric that is easily adopted by practitioners to make architectural decisions with respect to deadlock. With this aim in mind, this section explains the process followed to decide the final characterization described in Section 4. Table VII shows the comparative results of a set of primitive metrics applied to three different case studies and a fourth alternative design of the first one in order to illustrate the arguments proposed in this section. The case studies (CS) shown in this paper correspond, respectively, to the CS2 and CS3 in this table.

Table VI. Correlation matrix between risky elements, cycles and CFAs.

		Risky cycles								**Risky CFAs**	
ID	**Risky element**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**CFA1**	**CFA2**
1	Executive	X	X	X	X	X	X			X	X
2	Action Execution	X	X	X			X	X	X		
3	Plan Watcher				X		X		X	X	
4	Conditions Checker					X					
5	Pilot							X			
6	Event Queue	X		X	X						
7	B_Conditions					X					
8	B_Movement_Command							X			
9	S1*	X	X		X		X		X	X	
10	S2*		X	X			X				X
11	S3							X			
12	S4*					X				X	
13	Plan	X	X		X		X		X	X	
14	Command		X	X			X				X
15	Conditions					X				X	

Table VII. Comparative results of case studies.

	CS1	**CS2**	**CS3**	**CS4**	**CS4** vs **CS1**	**CS2** vs **CS1**
Number of elements	21	34	34	20	-5%	62%
Number of arcs	24	31	46	22	-8%	29%
Relational complexity	1,1	0,9	1,4	1,1	0%	–18%
Number of dependency cycles	7	8	28	4	-43%	14%
Static deadlock risk	5	6	8	4	-20%	20%
Dynamic deadlock risk	5	6	7	4	-20%	20%
Number of risky elements	11	6	15	10	-9%	–45%
Risky elements ratio	52%	18%	44%	50%	-4%	–65%
Number of static patterns	5	6	8	4	-20%	20%
Number of dynamic patterns	8	4	3	8	0%	–50%

8.1. Comparison based on percentages

The first approach proposed to evaluate the risk of deadlock [23] was a relative metric comparing the number of risky elements with the total number of elements of each kind (tasks, semaphores/mutexes, buffers and generic resources). This metric was intended to be interpreted in probabilistic terms: the higher its value, the higher the deadlock risk. Nevertheless, this option was discarded because it was observed that changes in the values of the intermediate parameters did not involve proportional increases of the risk (as discussed in [14]).

8.2. Comparison based on dependency cycles

The second approach was also based on a relative metric that compared the number of risky cycles with respect to the total number of dependency cycles present in a model. This metric was also promising, but its applicability to properly characterize deadlock risk was again considered questionable. The reason for this was that reducing the number of risky cycles did not involve reducing the value of this metric. If a model holds 7 dependency cycles and 5 of them are risky, its relative value is 71% (see CS1 in Table VII). The alternative design CS4 holds only 4 risky cycles, but it also has 4 dependency cycles and therefore the relative value of this parameter is 100%, which is higher than the value with a higher deadlock risk (which is contradictory).

An alternative to this metric can be the ratio of risky cycles with respect to the model size (number of building elements). For the same CS1, the metric value is 33% and for the alternative design CS4, it is 20%, which seems to be consistent with the expectations. The first drawback of this parameter is its physical interpretation: it is difficult to understand the concept of risky cycles compared with respect to a number of building elements as they are heterogeneous elements. However, the main issue had to do with the lack of proportionality of this metric when changing the primitive metrics. Let us consider a model holding 10 elements and just one risky cycle. The overall value of risk should be 10% (1/10). Now if we double the number of elements to 20 keeping the risky cycle, the value is now 5% (1/20). It looks like the risk is half in the second case, but intuitively the risk in fact remains the same.

8.3. Comparison based on model size

The third approach consists of using model size as a parameter to characterize the risk of deadlock. Model size can be defined as the number of building elements present in the model. The underlying idea is assuming that the higher the size of a model, the higher its deadlock risk. This approach is clearly refuted according to the results shown in Table VII. In CS1 the number of elements is 21 and the number of risky elements is 11. The value for the parameter based on size is therefore 52% (half of the elements are risky). In the CS2, this ratio is only 18%. This result can be consistent with the idea that bigger size involves higher risk, but it is in contradiction with the fact that the CS1 contains less risky cycles (5) than CS2 (6).

Alternative to the model size, the model complexity could be considered as a potential candidate to characterize the deadlock risk, with the same idea of higher complexity involving higher risk. The model complexity can be defined as the number of dependencies among building elements present in the model. However, for similar reasons, this characterization also fails. The model complexity value of CS1 is 1.1 and for CS2 it is 0.9. Once again, a lower complexity should involve lower risk, but this is in contradiction with the risky cycle parameter.

8.4. Final considerations

According to the previous observations, the main conclusion extracted is that the better parameter representing the intrinsic risk of deadlock present in a model is the number of risky cycles [14,31]. This parameter can be defined as the number of dependency cycles containing two or more tasks and at least one of the static deadlock patterns described in Section 4.

As discussed in Section 2, the number of dependency cycles was the main parameter used by the authors in [4,17], but standalone was considered not precise enough to characterize deadlock. The refinements proposed in this paper involve, first, the consideration of static deadlock patterns within the risky cycles, and second, the contribution of the behavioral part of an architecture to the overall deadlock risk present in a model. This refinement is described in detail in Section 4.

This approach is based on a refinement process, which reduces the number of cycles successively based on the model information with increasingly specific criteria to characterize the risk of deadlock. The approach provides a decreasing sequence of three numbers: number of dependency cycles, static deadlock risk and dynamic deadlock risk (e.g. 7, 5 and 2), where the really significant concerning deadlock are the last two.

Those models holding higher intrinsic risks (SDR) are more risky with respect to deadlock than others with lower values. Those models in which the intrinsic risk is similar should be compared according to the dynamic part of the risk (DDR), which is always equal to or lower than the static risk. This process permits the comprehensive comparison of models with respect to deadlock (as highlighted in Table VII).

9. CONCLUSIONS AND FUTURE WORK

This paper proposes a complementary approach to deadlock prevention, avoidance and detection techniques. The automated analysis of the basic properties of an architectural model allows for the characterization of complex problems, such as the overall deadlock risk of an RTS architecture with preliminary design information. The model information used in this characterization was:

- Cyclic dependency of tasks and resources.
- Structural patterns in architectural diagrams.
- Behavioral patterns in activity diagrams.

The appropriate combination of these three inputs provides two factors (SDR and behavioral deadlock risk), which can be used to compute the potential risk of a design to have deadlocks and, in addition, to compare alternative designs to choose the most appropriate with respect to deadlock.

This analysis has been validated by the application of DREAM to several case studies, some presented in this paper.

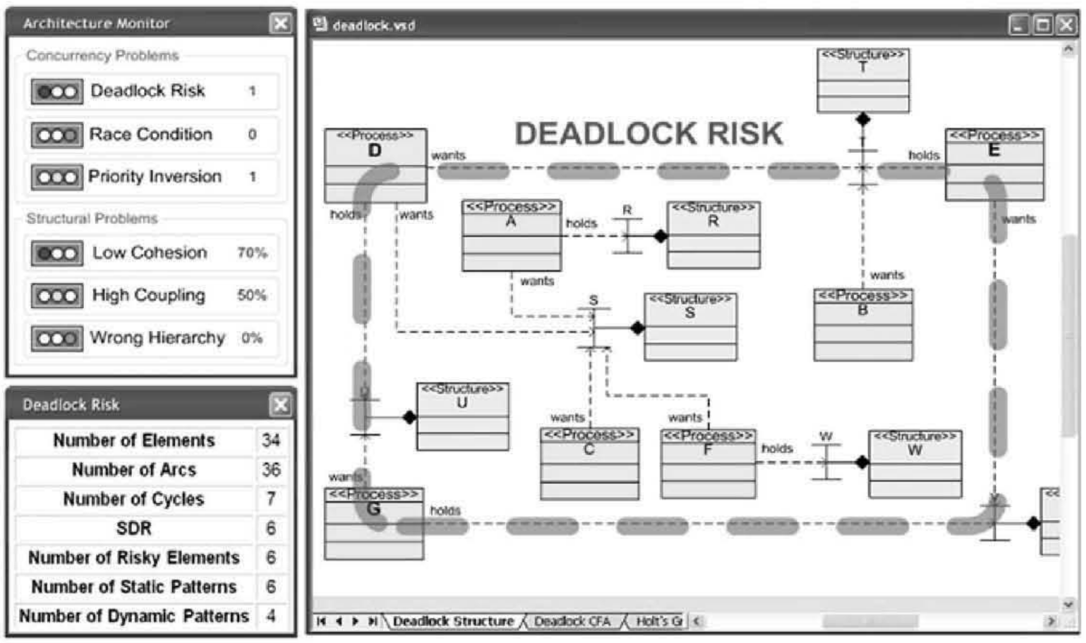
Although DREAM was developed as an add-on on top of PPOOA–Visio, the analysis proposed in this paper can be extended to architectural designs created with any other architectural description language representing concurrency. The conclusions obtained are of general applicability and were considered relevant in practice to make architectural decisions at early stages of the development of an avionics system.

The results of this research activity are the core of a doctoral thesis [14] presented with honors in 2010 at the Technical University of Madrid. The prototype of the tool DREAM was developed as part of an engineering degree thesis presented in 2009 at the same university.

Finally, it should be mentioned that the approach proposed in this paper can be considered as a partial view of a more complete architecture assessment tool. Additional concurrency and architectural problems in general (e.g. race conditions) could also be addressed with similar approaches. Ideally, an engineering dashboard with different indicators of system properties could be created to support architectural decisions and trade-offs (Figure 15).

ACRONYMS

CASE:	Computer Aided Software Engineering
CFA:	Causal Flow of Activities
COTS:	Component of the Shelf
DREAM:	Deadlock Risk Evaluation of Architectural Models (tool)
FIFO:	First-In First-Out
HLP:	Highest Locker Protocol
HMI:	Human Machine Interface
MBSE:	Model Based Systems Engineering
MDA:	Model Driven Architecture
PCP:	Priority Ceiling Protocol



Concurrency Problems	
Deadlock Risk	1
Race Condition	0
Priority Inversion	1

Structural Problems	
Low Cohesion	70%
High Coupling	50%
Wrong Hierarchy	0%

Deadlock Risk

Number of Elements	34
Number of Arcs	36
Number of Cycles	7
SDR	6
Number of Risky Elements	6
Number of Static Patterns	6
Number of Dynamic Patterns	4

Figure 15. Architectural dashboard.

OMG:	Object Management Group
PPOOA:	Pipelines of Processes in Object Oriented Architectures
RMA:	Rate Monotonic Analysis
RTOS:	Real-Time Operating System
RTS:	Real-Time System
UML:	Unified Modeling Language
XML:	Extensible Mark-up Language

ACKNOWLEDGEMENTS

We would like to send our special thanks to Ágatha Puigdueta for her contribution in the description of the avionics problem domain. She works as SW Architect in the SW Avionics Department of Systems Center of Competency of Airbus Military.

For confidentiality reasons no express references have been made to specific project information. All the documentation is under both industrial confidentiality and military security constraints.

None of the figures or tables contained in this document is under Airbus Military copyright. All the figures, tables and data shown in this paper were created specifically for the purposes of this paper.

REFERENCES

1. Fernandez JL. An Architectural Style for Object-Oriented Real-Time Systems. *Fifth International Conference on Software Reuse*, IEEE, 1998.
2. OMG MBSE Methodology. www.omgwiki.org/MBSE/doku.php?id=mbse:methodology.
3. Adve SV. Data Races are Evil with No Exceptions. *Communications ACM* 2010; **53**(11):84.
4. Agarwal R, Bensalem S, Farchi E, Havelund K, Nir-Buchbinder Y, Stoller SD, Ur S, Wang L. Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development* 2010; **54**(5):3.
5. Bensalem S, Bozga M, Nguyen T, Sifakis J. *D-Finder: A Tool for Compositional Deadlock Detection and Verification*, 21st ICCAV. LNCS, Vol. 5643. Springer-Verlag, Grenoble: France, 2009. 614–619.
6. Chaki S, Sinha N. Assume-Guarantee Reasoning for Deadlock. *SEI Technical Note*, 2006. CMU/SEI-2006-TN-028.
7. Sutter H. The Many Faces of Deadlock. *Dr. Dobbs Journal* 2008; **33**(8):53–54.
8. Clarke E, Grumberg O, Peled D. *Model Checking*. MIT Press: Cambridge, MA, USA, 1999.
9. Sha L, Rajkumar R, Lehoczky J. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers* 1990; **39**(9):1175–1185.

10. Briand L, Roy D. Meeting Deadlines in Hard Real-Time Systems - The Rate Monotonic Approach. IEEE Computer Society Press: Los Alamitos, CA, USA, 1999.
11. ARTiSAN Software Tools, Inc., Real-Time Studio. http://www.artisansw.com.
12. IBM Rational, Rhapsody System Designer. http://www-01.ibm.com/software/awdtools/rhapsody.
13. Emanuelsson P, Nilsson U. A Comparative Study of Industrial Static Analysis Tools. Technical Reports in Computer and Information Science. *Report number 2008:3*, 2008. Linköping University.
14. Monzon A. Técnicas para el Análisis de la Consistencia de Modelos en el Desarrollo de Software Embarcado (Evaluación del Riesgo de Interbloqueo). *Doctoral Thesis*, ETSI de Telecomunicación, Universidad Politécnica de Madrid (UPM), 2010.
15. Cheung SC, Kramer J. Checking Safety Properties using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology* 1999; **8**(1):49–78.
16. Magee J, Kramer J. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
17. Balarin F, Watanabe Y, Hsieh H, Lavagno L, Passerone C, Sangiovanni-Vincentelli A. Metropolis: An Integrated Electronic System Design Environment. IEEE Computer Society Press: Los Alamitos, CA, USA, 2003; **36**(4):45–52.
18. Chen X, Davare A, Hsieh H, Sangiovanni-Vincentelli A, Watanabe Y. Simulation based deadlock analysis for system level designs. *Proceedings of the 42nd Design Automation Conference, DAC 2005*, San Diego, CA, USA, June 13-17, 2005; 260–265.
19. Monzon A, Fernandez-Sanchez JL. An Ontological Representation of the Characteristic Problems of Real-Time Systems. *Embedded Real-Time Software International Conference*, Toulouse, 2008.
20. Fernandez JL, Monzon A. Extending UML for Real-Time Component Based Architectures. *International Conference on Software & Systems Engineering*, Paris, 2001.
21. Coffman EG, Elphick MJ, Shoshani A. System deadlocks. *Computing Surveys* 1971; **3**(2):67–78.
22. Fernandez JL, Marmol G. Modelling and Evaluating Real-Time Software Architectures. In *Ada-Europe 2009. LNCS*, Vol. 5570. Springer-Verlag, Brest: France, 2009. 164–176.
23. Monzon A, Fernandez JL. Deadlock Risk Assessment in Architectural Models of Real-Time Systems. In *IEEE Symposium on Industrial Embedded Systems (SIES 2009)*. IEEE Computer Society Press: Lausanne, Switzerland, 2009; 181–190. (July 8-10, 2009).
24. PPOOA-Visio. http://www.ppooa.com.es.
25. Singhoff F, Legrand J, Nana L, Marcé L. Cheddar: A flexible real-time scheduling framework. *ACM SIGAda Ada Letters* 2004; **24**(4):1–8.
26. Ovaska E, Balogh A, Campos S, Noguero A, Pataricza A, Tiensyrjä K, Vicedo J. Model and Quality Driven Embedded Systems Engineering, VTT Publications 705, GENESYS (GENeric Embedded SYStem platform, FP7-213322) Project Report, 2009.
27. Giannakopoulou D, Pasareanu CS, Lowry M, Washington R. Lifecycle Verification of the NASA Ames K9 Rover Executive. *ICAPS'05 Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*, Monterey, California, 2005.
28. Bresina JL, Bualat M, Fair M, Washington R, Wright A. The K9 On-Board Rover Architecture. *European Space Agency (ESA) Workshop on On-Board Autonomy*, 2001.
29. Volpe R, Nesnas IAD, Estlin T, Mutz D, Petras R, Das H. The CLARAty Architecture for Robotic Autonomy. *Proceedings of the 2001 IEEE Aerospace Conference*, Big Sky Montana, 2001.
30. Washington R, Golden K, Bresina J, Smith DE, Anderson C, Smith T. Autonomous rovers for Mars exploration. *Proc. of the 1999 IEEE Aerospace Conference*.
31. Monzon A, Fernandez-Sanchez JL, Ruiz-de-Castañeda J. *Applying Deadlock Risk Assessment in Architectural Models of Real-Time Systems Embedded Real-Time Software International Conference*, Toulouse, 2010.