AFRL-VA-WP-TR-2005-3039

ACES: ASPECT-ORIENTED COMPOSITION OF EMBEDDED SYSTEMS

Marcel Becker
Douglas R. Smith

Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

FEBRUARY 2004

Final Report for 05 July 2000 – 20 February 2004

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

AIR VEHICLES DIRECTORATE
AIR FORCE MATERIEL COMMAND
AIR FORCE RESEARCH LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site Public Affairs Office (AFRL/WS) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

/s/

VINCENT W. CRUM
Program Manager
Applications Branch

/s/

MICHAEL P. CAMDEN
Chief, Control Systems Development and
Air Force Research Laboratory

/s/

BRIAN W. VAN VLIET, Chief
Control Sciences Division
Air Vehicles Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
February 2004	Final	07/05/2000 – 02/20/2004

4. TITLE AND SUBTITLE
ACES: ASPECT-ORIENTED COMPOSITION OF EMBEDDED SYSTEMS

5a. CONTRACT NUMBER
F33615-00-C-3050

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0609199

6. AUTHOR(S)
Marcel Becker
Douglas R. Smith

5d. PROJECT NUMBER
ARPF

5e. TASK NUMBER
04

5f. WORK UNIT NUMBER
ST

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Vehicles Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7542

DARPA

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/VACC

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-VA-WP-TR-2005-3039

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT

Resource management is the key component of any real-time application. Traditional approaches to the development of embedded applications usually restrict their resource management capabilities to the level of the real-time scheduler implemented as an admissibility policy associated with a myopic dispatcher mechanism. The main problem with this approach is that they have very limited visibility of the actual levels of resource availability in the system. The non-functional aspect aspects of the architecture are completely ignored by these approaches. To overcome these limitations, system developers manually optimize the architectures for the particular application under consideration. These optimizations are not reusable and, once a new application is needed, a new architecture needs to be created.

15. SUBJECT TERMS
Distributed embedded systems, aspects, prescriptive aspects, software composition tools, software analysis tools, component based design

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Vincent W. Crum
Unclassified	Unclassified	Unclassified	SAR	100	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-8428

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

1. Executive Summary

This report describes our research on the Program Composition for Embedded Systems. Our project is based on the automatic generation of resource management components for embedded applications, and on the use of static analysis techniques for code verification.

Resource management is the key component of any real-time application. Traditional approaches to the development of embedded applications usually restrict their resource management capabilities to the level of the real-time scheduler implemented as an admissibility policy associated with a myopic dispatcher mechanism. The main problem with this approach is that they have very limited visibility of the actual levels of resource availability in the system. The non-functional aspect aspects of the architecture are completely ignored by these approaches. To overcome these limitations, system developers manually optimize the architectures for the particular application under consideration. These optimizations are not reusable and, once a new application is needed, a new architecture needs to be created.

Kestrel is implementing high-performance real-time scheduling algorithms using automatic software synthesis technology. In this approach, a high-level description of a particular real-time scheduling problem is translated into an algorithmic model. This model is then further refined into a very optimized lower-level implementation that targets the execution platform and middleware architecture. The two main benefits of this modeling and synthesis approach are: (1) the clear separation between functional requirements and lower level implementation details; and (2) the support for automatic translation of the functional specification into a product family of real-time scheduling services.

The generator was implemented as a component of Planware, a software synthesis tool specialized to the domain of resource planning and scheduling. Planware was initially designed as a generator for off-line schedulers that operate in batch mode. The goal of the project was to implement a generator for real-time schedulers that could be integrated in the Boeing OEP.

There are two main differences between generating real-time schedulers and offline schedulers: awareness and adaptivity. The first difference is that real-time schedulers must be aware of the actual level of resources available for its own execution, and for the execution of the tasks it is scheduling. These schedulers should be able of not only allocating task for execution, but also of monitoring different levels of resource availability in the system. In response to feedback from the environment, the scheduler should be able to estimate QoS levels and decide, based on the required QoS levels specified, how to execute the next set of tasks.

The second main difference is that real-time schedulers must dynamically adapt its allocation or dispatch strategy based on the difference between nominal or required QoS levels, and actual or estimated values. The scheduling algorithms must identify that the situation has changed and that maybe a different scheduling strategy is needed. For example, if a fighter is performing a patrol mission and suddenly needs to engage the enemy, the scheduling strategy for on-board systems should also adapt to give higher priority to task related to combat mode

as opposed to patrol or recognizance mode.

The work on the generator involves writing Planware program schemas, templates for program generation, capable of implementing a number of different real-time scheduling strategies: ranging from static, offline scheduler to pure dispatch-based, dynamic scheduling algorithms. In addition to designing and implementing the generator, the models describing the problem to be solved also must be described. This report will focus on the latter: The modeling approach and the resource models used to develop end-to-end resource management capabilities for embedded applications.

In addition to the resource management problem modeling, this project also performed a defect analysis of the software infrastructure used as the experimental platform: The Tao ORB used by Boeing's *BoldStroke* architecture. The result was that a reduced number of non-critical defects were detected in thousands of different files.

2. Introduction

This document addresses the problem of correctly and efficiently managing resource usage over time in real-time, embedded applications. The main goal of this research was to investigate how real-time performance requirements specified at system design time can be used to synthesize high-performance scheduling algorithms that guarantee desired QoS levels during system execution. To accomplish that, we built a prototype software development environment that supports the precise specification of scheduling requirements for real-time systems, and automatically synthesizes and integrates highly specialized and optimized services for resource management into a real-time application architecture.

The motivation behind our approach is the critical role scheduling services play in supporting QoS requirements for real-time systems [5], and the ad-hoc nature of the design process currently used for these types of services. Section 3 below describes in more details the specific real-time compositional problem we are addressing and its importance. Section 4 discusses how our work relates to current research in this area, and emphasizes the differences between Kestrel's algorithm generation approach and recent work on adaptive scheduling algorithms like RT-ARM [8]. Section 6 describes how Planware addresses the technical challenges in the PCES program. Section 5 summarizes Planware modeling approach. Section 7 describes the models created using Planware to describe the Sensor to Decision Maker to Shooter/Weapon problem. Section 9 described our implementation of the C code generator capable of generating C code from our Kestrel's high-level specification language.

In section 8 we summarize the results of a static analysis of the TAO ORB code performed by Reasoning Inc. The full report of the analysis is provided as an appendix to this report.

3. Automatic Synthesis Of Real-Time Scheduling Algorithms

The design and development of real-time, mission-critical applications is a complex activity that cuts across several different disciplines. In addition to the complexity associated with traditional development of software artifacts, real-time systems are required to correctly manage and coordinate the execution of a number of different tasks competing for resource usage under very stringent timing constraints. Correctly dealing with time and correctly managing resource usage are probably the two most important requirements imposed on the design and implementation of real-time systems: The way in which the system handles time and resources characterizes the system temporal behavior and its correctness [12]. The key issue here is how to play the trade off between the processing overhead of the scheduling algorithm, and the quality of service (QoS) resulting from the resource allocation while guaranteeing overall system correctness and reliability.

By the inherent nature of the problem described in the previous paragraph, implementations of middleware scheduling services for embedded, real-time applications need to be highly optimized for both the lower-level platform executing the tasks, and for the high-level domain level application. Historically, this optimization has been performed manually, and using ad-hoc design approaches. The result of this process is costly maintenance and almost no re-use of previously implemented services. Furthermore, timing and scheduling decisions are usually implicitly embedded in the code, and made without a solid theoretical foundation that guarantees correctness results. This severely limits the ability of predicting system behavior under different operating conditions [15]. Correctness claims about the system are then made based on the result of exhaustive, and expensive, testing.

To address some of these problems, several formalisms have been proposed to allow system designers to explicitly represent timing and resource constraints. These formalisms also facilitate the automatic verification and validation of specifications. Despite the benefits of using formal methods, very few approaches provide mechanisms to translate specifications into efficient executable code that preserve the properties explicitly represented in the specification. Kestrel's software synthesis environment, Specware [20] and related technologies, is one of such approaches.

One of the domains in which Kestrel's correct-by-construction software synthesis approach has been particularly successful is scheduling. Extremely efficient scheduling algorithms for a number of different practical applications have been successfully synthesized using a prototype software generator similar to the one discussed here [18, 1]. The new aspect is the nature of the real-time scheduling problem, and the set of additional requirements concerning the algorithm processing time. Previous efforts at Kestrel were concerned with the generation of off-line scheduling systems for target domains where very complex constraints must be taken into account like, for example, military logistics and transportation networks. These algorithms are very complex and deal with large networks of constraints. Real-time resource management imposes a slightly different set of requirements on the characteristics of the algorithms generated. Since the scheduler is also competing for resource usage, implementations of real-time schedulers trade complexity for efficiency and correctness. The scheduler must be

able to allocate time to its own execution, and to precisely compute or estimate its required resource usage. The integration of the service into a middleware framework, and required optimizations along three dimensions, the higher-level application, the lower level OS, and the middleware system, has not previously been investigated.

As current mission-critical applications evolve into highly distributed, heterogeneous systems, the more critical and complex the issue of correctly managing time and resources becomes. Our synthesis approach will allow a high level of optimization for each application-platform pair with a minimal amount of effort while preserving timing properties.

4. Related Work

Real-time scheduling problems range from static, single-processor problems in which all tasks and respective timing constraints and parameters are known in advance, to dynamic, multiple processors where very little is known about the tasks and their temporal characteristics. Purely static problems are less interesting since they can usually be solved by an off-line scheduler. Before system execution, the scheduler checks schedulability conditions for the set of tasks, and defines a certain scheduling policy to be followed by the run-time kernel while executing the tasks. For this class of problems, classical scheduling theory provides techniques that allow optimal or nearly optimal resource utilization and task processing [21]. Although the scheduling problem in this case is not very complex, guaranteeing timing constraints at design time and separating timing from functional requirements are still open issues.

For static problems, the algorithms implementing the scheduling decisions are implicitly embedded in the implementation of the application. In other words, the scheduling decisions are part of the application's logic. Maintenance and modifications to system implemented in this fashion are very expensive. The generator approach we are proposing allows different sets of requirements to be treated individually, represented explicitly, and automatically combined, or integrated, during system synthesis.

The quality of solutions provided by purely static analysis starts to degrade as uncertainty about task's processing and arrival times increases. Dynamic scheduling problems are particularly challenging due to the lack of theoretical results capable of guaranteeing that timing properties will be satisfied during execution. Traditionally, dynamic real-time systems have been statically scheduled using rate monotonic scheduling techniques [10]. Worst-case assumptions are used to compensate for the uncertainty on task's parameters [6].

To overcome the limitation of purely static scheduling techniques applied to dynamic problems, recent research is considering the use of hybrid allocation schemes that add dynamic dispatching capabilities to the task executor [2, 7, 8]. In these approaches, a static analysis is initially performed on the set of tasks to be executed. This analysis determines the schedulability of the set of tasks, and computes some basic metrics to guide the run-time executor. During run-time, the executor uses the result of the static analysis plus values computed using one or more scheduling heuristics (e.g., earliest deadline first or minimum laxity first) to perform the real-time dispatching of tasks. These types of services are called adaptive scheduling, or adaptive resource management [17]. Associated with the notion of adaptive scheduling is the concept of Quality of Service (QoS) [9]. QoS requirements translate abstract metrics of user perceived system quality into concrete metrics on resource utilization. Dynamically allocating, maintaining, and negotiating QoS levels along multiple dimensions during system execution is the main motivation behind adaptive resource management.

Notice that adaptation as described above refers to the use of a flexible resource allocation policy or scheduling framework capable of compensating only for the uncertainties on resource requirements during application execution. Although the use of an adaptive scheduling policy facilitates the customization of the software functionality after initial development, and has the potential to "improve versatility and decrease lifecycle maintenance

costs for embedded real-time systems, [2]" it does not directly account for uncertainties on the computational environment the application will run (e.g., hardware and software platforms, network connectivity, etc) and still preserves the two main problems of the ad-hoc design nature previously discussed: to demonstrate the correctness of the resource allocation policy, empirical results and extensive tests are still required; and to optimize the system performance for each application-middleware platform triple, handcrafted code is still necessary. Generic scheduling services optimized and tested for only one of the dimensions in isolation-application, platform, or middleware-provides limited improvement. For example, a very efficient, middleware-optimized dispatcher would always compute the heuristic metrics and add events to an event queue even when the platform application pair is such that direct channels could be established for certain classes of events.

In our approach, adaptation is achieved at both the software architecture and resource management level. The scheduling services generated are parameterized by scheduling policy, hardware and software platform, middleware, and higher-level application. Using composition and refinement, the generator will automatically synthesize efficient scheduling services that are not only tailored to the execution environment, but are also guaranteed to preserve the timing properties explicitly described as higher-level requirements. A very efficient compilation and propagation mechanism is used to maintain and update the multi-dimensional QoS requirements as a network of constraints. As the network evolves, new scheduling algorithms can be dynamically synthesized and seamlessly integrated into the overall system architecture without disturbance of the application. The need for testing of modified versions of the original algorithm is minimized since the generation approach is less likely to introduce errors in the new implementation than handcrafted code.

Given the high performance of automatically generated algorithms, additional scheduling techniques like reactive or proactive strategies can also be explored. Reactive and proactive scheduling strategies that respond to changes in the environment and dynamically maintain and update statically generated schedules are usually very expensive to be used in real-time applications. Dispatch heuristics are used because of their little computational overhead. Heuristic methods are known to be myopic, that is, given their limited view of the overall problem, they provide only local optimality. Better overall system performance can potentially be achieved by the use of global scheduling methods.

5. Planware

Planware system is Kestrel's domain-specific generator of high-performance planners & schedulers.

Planware provides an answer to the question of how to help automate the acquisition of requirements from the user, and how to assemble a formal requirement specification for the user. The key idea is to focus on a narrow, well-defined class of problems and programs, and to build a precise, abstract, domain-specific description formalism that covers this class.

From a software development perspective, interaction with the user is only required in order to obtain the refinement from the abstract specification to a description of the concrete requirements of a particular problem instance.

Kestrel's research has focused for years on automating the generation of correct programs from formal specifications. Scheduling has been a prime application domain for testing out the generation technology. Previous scheduler generators implemented by Kestrel like KIDS [18], and some previous versions of Planware [1] were criticized because of the complexity of the specification language, and their inability of considering problems with multiple resources. Planware's design focused on addressing these concerns: It is simple to use, and can handle the coordination of multiple resources. Planware's main capabilities are:

Simple Modeling Language: The complex logical description used by KIDS was substituted with a simple, intuitive formalism that describes the behavior of a resource by explicitly describing the set of activities it must perform.

Multiple Resource Coordination: The synchronized use of multiple resources is obtained by the use of compact descriptions of resources capabilities represented as services (see section 5.1.5).

Configurable Problem-Solving Strategy: The modeler can select the scheduling strategy to be used by the generated scheduling application just by choosing among a number of different search-based implementations available to the code generator.

Integrated Development Environment: The modeling, scheduler generation, and schedule computation is performed in a uniform development environment that supports all phases of the development process.

5.1. Planware Modeling Primitives

5.1.1. Planware in a Nutshell

One of the most important requirements driving the design and implementation of the new version of Planware was the ability to represent the synchronization of multiple resources without using complex mathematical or logical formalisms. To achieve this goal, Planware uses Abstract State Machines (ASM) [16] to model the behavior of tasks and resources.

To handle the interactions involved in multi-resource problems, Planware uses a service matching theory in which resources can offer and/or require different types of services. The scheduler is responsible for matching providers and requesters in a consistent fashion. For example, a transportation organization might want a scheduler to simultaneously handle its aircraft, crews, fuel, and airport load/unload facilities. Each resource has its own internal required patterns of behavior and may have dependencies on other resources.

The semantics of a resource is the set of possible behaviors that it can exhibit. We treat these behaviors as (temporal) sequences of activities modeled as ASM modes (abstract states). Each activity is characterized by a set of mode variables (e.g. start-time and duration), the set of services that it offers (e.g. the flying mode of an aircraft offers transportation service), and the set of services that it requires (e.g. the flying mode of an aircraft requires the services of a crew). A formal theory of a resource should have as models exactly the physically feasible behaviors of the resource. The axioms serve to constrain the values it can exhibit. A formal theory constrains the values that mode variables can take on in states (e.g. the weight of cargo cannot exceed a maximum value during the flying mode of an aircraft). The transitions serve to constrain the evolution of the mode variables (e.g. the finish time of one activity must occur no later than the start time of the next activity; a take-off activity can only be followed by a fly activity, etc.).

All modes have variables for the start-time, finish-time, and duration of the corresponding activity, plus related constraints. They may also have other variables and constraints that further define the resource behavior, and better describe the mode. A mode may also provide and/or require services. Only the temporal resource constraints of the activities are relevant, not their nature. Thus the same mechanism can be used to model activities as diverse as transportation, computation, allocating a logical design to a hardware/software platform, personnel assignments, workflow, manufacturing, and so on.

A task is also expressed formally as an ASM. The main difference between a task and a resource is that a task offers no service – it only requires services of resources. For example, a fighter mission may require fuel, crew, and weapons resources.

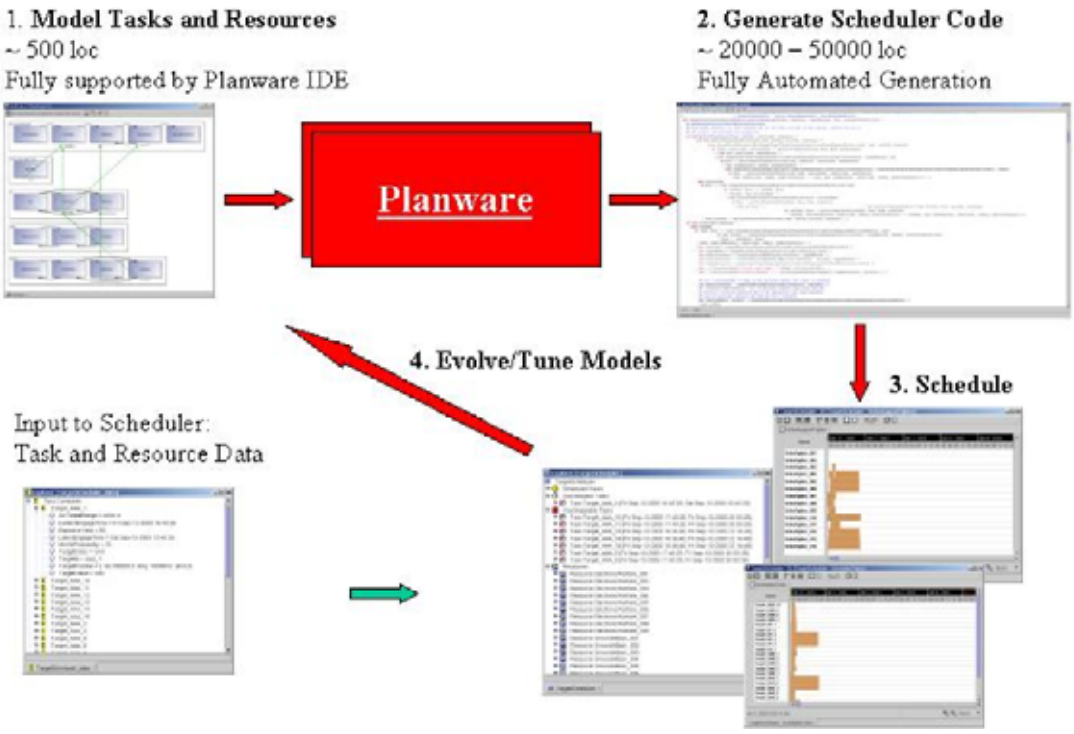


Figure 1: Planware Generation Process

Figure 1 summarizes the modeling, code generation, and application test process. A more detailed explanation of these concepts is provided in the following paragraphs.

5.1.2. Representing Resources as Activity Machines

To allow users to specify complex multi-resource problems we started by trying to identify a language that would naturally represent the basic concepts in the domain. The first approximation was to look for planning and scheduling ontologies.

Smith & Becker in [19] describe an ontology for planning and scheduling systems. The five top-level entities in this ontology are tasks or demands, activities, resources, services, and constraints. Using this ontology, the role of a scheduling or planning system can be described as the prescription of a sequence of activities that a set of resources must perform over time to perform the services required by a task. Based on this description, it is clear that any formalism for describing a scheduling problem domain must be able to represent tasks, resources, activities, and services, plus associated constraints.

If we consider the processing of an activity by a resource as a possible "state" or "mode" the resource entity can assume, we can think of a resource model as the description of all the valid sequences of activities it can perform. For example, a transportation aircraft might have the following sequence of activities:

Prepare → Fly → Fly → Unload → Refuel → Fly → and so on.

Or a strike fighter might execute the following sequence of activities while executing a mission:

Rearm → Position → EngageTarget → Position → EngageTarget → and so on.

Each legal sequence of activities is called a behavior. A resource is characterized by a potentially infinite collection of behaviors. A convenient and intuitive model for concisely representing an infinite collection of behaviors is a state machine. Our modeling approach is based on state machines. Since the term ”state” is somewhat misleading we prefer to refer to our models as activity machines, as exemplified in Figure 2.

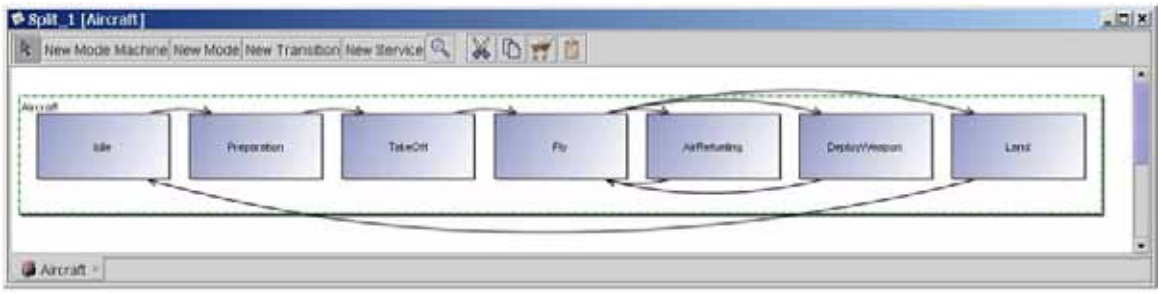


Figure 2: Activity Machine

In the activity machine diagram, we refer to the boxes/states as activities. The arrows are called transitions and indicate which activities can legally follow one another.

5.1.3. Representing Activities as Sets of State Variables

The activity machine per se gives us general information about the legal sequences of activities for a resource, but little information about the activities. Clearly in scheduling we need to model the timing of activities, as well as capacity of a resource to do useful work, and other physical constraints. The next step in modeling resources is to represent information about an activity via activity variables, simply called variables. The set of variables defined for a particular resource model represents the state descriptor used to describe all activities performed by this resource. For example, a Fly activity of a transportation resource might be characterized by variables that model its start time, finish time, origin, destination, and others. Planware uses the same collection of variables to characterize all activities of a particular activity machine. This collection is sometimes called the signature of the machine (See Figure 3). Technically, each activity is treated as a first-order theory, with the signature providing the vocabulary of the theory, and the axioms specifying constraints on the meaning of the variables (and other vocabulary).

```
mode-machine StrikeFighter is
    constant baseLoc                                  : Location
    constant fuelBurnRate                             : BurnRate
    constant maxFuelCapacity                          : Capacity
    constant engagementDuration                       : Duration
    constant rearmingDuration                         : Duration
    input-variable targetLoc                          : Location
    input-variable munitionType                       : Munition
    input-variable targetYield                        : Capacity
    internal-variable origin, dest, airRefTrack       : Location
    external-variable st, et, duration                : Time
end-mode-machine
```

Figure 3: Activity Machine Signature for a Strike Fighter Resource

The variables defining the signature of an activity machine are further divided into four groups: constants, internal variables, external variables, and input variables.

Constants: are the fixed parameters used to characterize invariant aspects of the resource. For example, the size of the cargo hold of an aircraft, its maximum speed, the maximum amount of fuel it can carry are parameters that can be constants for a given aircraft type. The set of constants define the input values that should be provided to the scheduling system to create concrete instances of resources at schedule computation time.

Internal Variables are auxiliary variables used by the scheduler for bookkeeping purposes. For example, if there is a constraint that states than an aircraft needs to go through preventive maintenance after a certain number of hours flown, an internal variable can be used to maintain the number of hours flown since last maintenance.

External Variables represent the values that are modified by the scheduler engine while computing the schedule. For example, the start and end time of an aircraft Fly activity will be set by the scheduler based on the availability of additional resources like Crews and Airports. External variable values depend not only upon the evolution of the resource machine but also upon the scheduling decisions made by the problem-solving control strategy.

Input variables Input variables serve as a mechanism for passing parameters between a resource requesting a service and a resource providing it. For example, the origin and destination of a Fly activity may be specified as input variables whose values get assigned each time a transportation task needs to be satisfied. Input variables act as constant variables for a given fragment of the resource behavior. The role of the input variables and external variables will be further discussed when we present the concept of services and service matches.

After defining the variables that can take on possibly new values at each activity, we can express more information about a behavior:

Most variables of interest hold their values for the duration of an activity, and only change with the transition between activities. In the syntax construct used to describe valid transitions, the variables whose values change as a result of the transition are explicitly represented. Three external variables are pre-defined for all activity machines, and do not need to be explicitly represented in the assignments field of a transition: start-time, end-time, and duration. Implicit in the structure of the activity machine is the constraint that states that the start-time of an activity is always greater than or equal to the end-time of the preceding activity; and that the end-time of an activity is equal to or greater than the sum of its start-time and its duration. Additional properties about the possible values a variable can take on can be stated through the use of constraint expressions.

5.1.4. Defining Behavior using Constraints

Not all combinations of values for the variables are physically possible. For example, if the maximum munition capacity of a fighter is 100 tons, then an activity in which the munitionRequired variable has a value exceeding 100 tons does not model a realizable situation. To rule out such impossible situations, each activity has axioms that express constraints on the values that variables can take on in an activity. The generated scheduler uses the constraints expressed in the model as pruning conditions to drive the expansion of the search tree.

Furthermore, it is necessary to put constraints on transitions, to model the physically realizable evolution of variables between activities. For example, the transition from *Preparation* to *TakeOff* in Figure 2 specifies that the end time of activity *Preparation* should less than or equal to the start time of activity *TakeOff*. The constraints labeling a transition must refer to the values of variables in both the before and after modes. The usual notation is to refer to the value of a variable **x** in the after state by priming it: **x'**. So the constraint **endTime' <= startTime** means that the finish time of the before activity must be no later than the start of the after activity. As previously mentioned, a number of temporal constraints between modes do not need to be explicitly represented since the structure of the machine already assumes temporal dependencies between the sequence of valid activities. The constraints on the transitions are called guards. Guards are used to guide the scheduler to expand the correct sequence of activities needed to satisfy the request.

Similar to the guards on the transition, the assignment of values to variables can also be used to express or enforce constraints. Conditional expressions can be used in the definition of variable assignments to drive the valid sequence of activities expanded. Figure 4 shows an example of a simplified Planware model for a strike fighter with four activities: Idle, Rearming, Positioning, Refueling, and EngagingTarget, and some of the valid transitions between the different modes. In the transition from Rearming to Positioning we determine if a Refueling activity is needed or not: If there is enough fuel to engage the target and return to the home base, no refueling is needed; otherwise, the value of the variable airRefTrack is set to the location of the air refueling track. The variable airRefTrack is used in the guard of the transition from Positioning to Refueling to force the scheduling algorithm to include an air refueling activity in the activity sequence if necessary.

```
mode-machine StrikeFighter is

    mode Idle has
    end-mode
    mode Rearming has
      required-invariant loadMunition (st, et, rearmingDuration, munitionType)
    end-mode
    mode Positioning has
    end-mode
    mode EngagingTarget has
      provided-invariant engageTarget(st, et, targetLoc, munitionType, targetYield)
    end-mode
    transition from Idle to Rearming when {} is
    { duration := rearmingDuration }
    transition from Rearming to Positioning when {} is {
        dest := (if  (consumedFuel(baseLoc, targetLoc, fuelBurnRate) <= maxFuelCapacity)
                 then findAirTrackLocation(baseLoc, targetLoc)
                 else targetLoc),
        airRefTrack := (if (consumedFuel(baseLoc, targetLoc, fuelBurnRate) <= axFuelCapacity)
                        then dest
                        else zeroLoc)}
    transition from Positioning to Refueling when { airRefTrack != zeroLoc } is {}
    transition from Positioning to Engaging when { dest = targetLoc } is {}

end-mode-machine
```

Figure 4: Activity Machine for Strike Fighter Resource

5.1.5. Coordinating Resources Using Services

The modes or activities, the variables, the transitions, and the constraints are sufficient to represent the behavior of an individual resource. The key missing element of this formalism is how to connect resources to tasks, and how to coordinate the usage of several resources to accomplish complex tasks. For example, the transportation of certain amount of cargo between two locations may involve the usage of a number of different aircraft, airports, crews, fuel, ground control personnel, diplomatic clearances, etc. Engaging a given target may involve the coordination of air refueling tankers, escort aircraft, air patrol, jammers, etc.

We need to provide modeling constructs that allow the explicit representation of these dependencies. The missing modeling construct is the service: The service is the element used to coordinate and synchronize the execution of activities across different resources. Each activity machine may specify required and/or provided services. Machines that only request services define the top-level tasks that drive the entire scheduling process. Resources are machines that provide one or more services. Resources can also request additional services. For example, to provide transportation service to a transportation request, the aircraft may need services from one or more crew resources. Figure 4 gives an example of a resource that offers an *engageTarget* service and requires a *loadMunition* service to be able to engage the target. In this case, the resource plays the dual role of provider and requester. The concept of requested and provided services allows tasks and resources to be represented using a uniform formalism.

As illustrated in figure 4, a service is specified by a predicate associated with a mode together with an indication of whether it is a provided or required service. For example, the engage target service may be represented by the predicate *engageTarget(startTime, endTime, targetLoc, targetMunitionType)* which specifies a certain target located at coordinates specified as *targetLoc* to be engaged some time during the time interval defined by the values of *startTime* and *endTime*.

The requester resource specifies the service as a required condition. The provider specifies it as a provided condition. For temporal synchronization, a service can be specified as a pre-condition, a post-condition, or an invariant. If the service is specified as an invariant, both activities, the requesting and the providing, should start and end at the same time. For the other types, there are set of rules to establish the appropriate synchronization depending on the characteristics of the provider and requester. For example, if the requesting service is a pre-condition and the providing service is a post-condition, the providing activity should finish before the requesting activity can start.

5.1.6. Passing parameters through Service Descriptions

There is also a set of rules governing the assignment of values to the parameters specified in the service description. For the requesting resource, external variables present in the service predicate will have their values set by the scheduler after an appropriate provider has been identified. All other variables will not change value. For the providing resource, input variables present in the service predicate will act as constants for the purpose of finding a valid sequence of activities to satisfy the request. External variables will be unified with external variables coming from the requester. At scheduling time, any constraints imposed on the external variables of the requester, will be translated to the corresponding variables of the provider. In our *engageTarget* example, the variables *targetLoc* and *targetMunitionType* are defined as input variables. Their values will be passed by the requesting target and will be treated as constant for the duration of that particular mission.

Before we go on to discuss how services between machines are linked up, we note that the introduction of services in mode machines allows us to treat tasks as a special case of resource. Tasks are the drivers of a planning or scheduling problem. The overall nature of the scheduling problem is to carry out a set of tasks subject to the constraints imposed by the available resources. In terms of the activity machine model, a task can be modeled as a resource that requires service, but offers none. Figure 5 shows an air strike task modeled as a simple machine with just one mode. Its *engagingTarget* mode requires the service *engageTarget*. The task model also specifies the values to be used in the service request: target position and *munition* type are defined as constant parameters and should be provided as input to the provider resource. Note that the *engageTarget* mode has two axioms expressing constraints on the start and finish time of the activity – the start time must occur no earlier than the *earliestTimeOnTarget* and the finish time must occur before the *latestTimeOnTarget*. Figure 5 represents a typical example of a task model.

```
mode-machine Target is

constant targetClass : TargetClass
constant minKillProbability : Probability
constant targetLoc : Location
constant earliestTimeOnTarget, latestTimeOnTarget : Time
constant munitionType : Munition
external-variable st, et, duration : Time
mode EngagingTarget has
  required-invariant engageTarget(st, et, targetLoc, munitionType)
  constraint st >= earliestTimeOnTarget
  constraint et <= latestTimeOnTarget
end-mode
end-mode-machine
```

Figure 5: Target Task Model

We have modeled the component tasks and resources individually, and now we need to model the composite system. To model a complex resource system we focus on the interactions of the components, which are specified by the services. We use the service match formula schema below to express the conditions under which the service provided by resource Prov satisfies the service required by resource Req:

$$\forall\,(\text{constants (Req)}, \text{input-vars (Req)}, \text{constants (Prov)})$$
$$\exists\,(\text{ext-vars (Req)}, \text{internal-vars (Req)}, \text{input-vars (Prov)}, \text{ext-vars (Prov)}, \text{internal-vars (Prov)})$$
$$(\text{Provided Conditions (Req)} \wedge \text{ProvidedConditions(Prov)}$$
$$\rightarrow$$
$$\text{ReqConditions(Req)} \wedge \text{Constraints (Req)} \wedge \text{ReqConditions(Prov)} \wedge \text{Constraints (Prov))}$$

We expect two kinds of information from reasoning about the formula. First, we get witnesses for the existentials, meaning that for each existentially quantified variable, we extract a term over the preceding universally quantified variables. Second, we gather up any of the conjuncts in the consequent of the formula that cannot be proved. These gathered constraints are the aggregated constraints of the composite resource Req - Prov. While they are not provable at design time, they will be treated as constraints to enforce at run-time (i.e. schedule-computation-time), either via pruning or constraint propagation.

In Planware the actual ground constraints are determined dynamically, and depend on the input data, together with the dynamics of the scheduling process (the current state of the process). This is in contrast to many Operations Research and Constraint Programming systems in which a static set of constraints is passed to a generic solver. Planware not only generates a customized solver for each problem, but the solver works on a dynamic constraint problem.

5.1.7. Scheduler Code Generation

From the activity machine models described in the previous section, Planware automatically generates a fully operational scheduling application. The key component of the generated code is a search-based scheduling algorithm, and a constraint propagation mechanism.

In addition to the search algorithm implementation, support code is also generated to represent resources and activities, and to produce I/O for the application. In the following paragraphs we will explain in more detail the code generation process, and the different components created by Planware.

5.1.8. Scheduling implemented as a Bidding Process

Planware generates search-based scheduling algorithms implementing a bidding process as its main control cycle. In this process, entities requiring services post tasks, or requests for bids. Provider resources capable of performing the type of service specified in a task respond with their best bid according to their own internal strategy. The requesters then collect the bids, rank them according to the requester's objective function, select the best bid, and notify the selected bidders. Constraint propagation is triggered every time a bid is accepted. The propagation updates the internal state of the resources involved in the bidding. The rejected bids are discarded, and no additional work is needed.

5.1.9. Algorithms Generated by Composing Program Schemas

The concrete implementation of the scheduling algorithm used in a particular application is obtained by instantiating and composing program schemes. A program schema is a parameterized fragment of algorithmic logic that gets instantiated for each service match between a given provider and requester. Different program schemes are used to allow a number of different bidding generation and bidding selection mechanisms to be combined in the implementation of an application. For example, a program schema could be used to implement a bidding mechanism in which the first feasible bid is accepted; a different one could collect up to n bids, and select the one that can finish the service with the minimum amount of time; a third one could generate all possible bids and select the one with earliest start-time.

5.1.10. Schema Composition Driven by Service Tree

The program schemes are composed in a tree-like structure reflecting the structure of service matches defined by the activity machine models. As described in the previous section, an activity machine can request, and/or provide services. For each required service in the model, the generator code will search for a matching provider – a resource providing a service that matches the signature of the required service. As a provider for a given service can request additional services from other resources, the service matches define a directed acyclic graph we refer to as the service-match tree. The code generator traverses this service tree creating the appropriate code to formulate the task, generate the bids, and select the best bid.

5.1.11. Scheduling Strategy Selected through Service Match

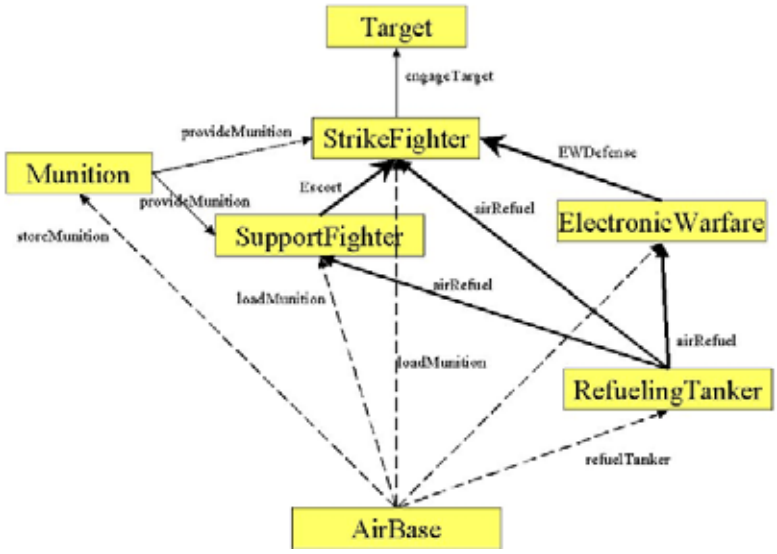


Figure 6: Service Tree for Target Scheduler

The service-match tree is an auxiliary data-structure automatically generated by Planware before the actual application generation. By exposing the structure of the service-match tree to the user through the graphical interface, a greater level of control over the code generation can be obtained. An advanced user can configure the search schemes to be used by the generator for each service match in the tree. Figure 6 shows a possible service tree for a model in which, to engage a target, Strike Fighter may require escort, EW defense, air refueling, and munition. Each of the required resource may also require additional resources. In this example, the munition resource requires some storage space at the AirBase. Through the GUI, the user can select the search strategy used to satisfy the requests, as well as the sequence in which the services are satisfied. Once a provider and a requester for a given service are specified, the system automatically generates and inserts in the model source file a textual representation describing the service tree containing all possible matches between the different resources. Figure 7 shows the service match between the target task and the strike fighter. In this example, there is only one type of resource that can provide the engageTarget service. In general, several different resources could offer the same service.

5.1.12. Constraint Propagation Implemented by Arc Consistency Algorithm

The constraint propagation code used to update the resources is not automatically synthesized. A standard implementation of an arc consistency algorithm that propagates temporal constraints on a simple temporal network is used. All schedulers generated share the same implementation.

```
service-match Target EngagingTarget engageTarget is
service-requester Target
requester-mode EngagingTarget
requested-service required-invariant engageTarget
search-strategy BreadthFirst
service-providers has {
      provider-record StrikeFighter Engaging engageTarget is
      service-provider StrikeFighter
      provider-mode Engaging
      provided-service
            provided-invariant engageTarget
       search-strategy BreadthFirst
       end-provider-record
       }
end-service-match
```

Figure 7: Service Match generated for Target task and StrikeFighter

The constraint propagation is responsible for maintaining consistent start times for all scheduled activities. Each activity has a time bound representing the earliest and latest time the activity can start executing. Each activity time bound defines a node in the constraint network. The scheduler adds temporal constraints (arcs) between time bounds (nodes) as the problem solving process evolves. If constraint violations are detected, scheduling decisions are retracted, and the search backtracks to the last decision point before the violation.

5.1.13. Resource Represented as Capacity Profile

Activities and resources are closely related. Resources are represented by a capacity profile: A temporal sequence of activities representing the resource reservations performed by the scheduler. The profile represents a trace of the activity machine defined in the abstract model. The data structure used to represent the profile must be optimized for lookup and update. During the bid creation phase of the scheduling algorithm, the providers inspect their capacity profile searching for feasible intervals capable of feasibly performing the requested service. Once the requester accepts a bid, the selected provider updates its own profile to reflect the new reservation. Planware uses a binary tree implementation optimized for the particular type of resource.

The representation of the activities in the resource profile is generated from the set of variables defined by the activity machine model. All activities created for a given resource instance share the same set of constants. Activities are defined as a record structure with a field for each variable described in the model. The code to access and set each one of these fields is automatically generated. Additional code to print and display individual activities, and activities sequences is also generated to facilitate debugging, testing, and schedule visualization. A number of different output formats are supported: plain text, XML, etc.

5.1.14. Activity Sequences computed Dynamically

Activities are dynamically created at schedule computation time. Activity sequences are created by the bidding mechanism previously discussed. The generation of the bid creation mechanism is one of the most complex components of Planware. It involves the generation of code capable of querying the resource profile for feasible intervals, expanding the sequence of activities the resource must execute, and enforcing the constraints imposed on the service by both the requester and by the provider resource.

The bid creation mechanism is implemented as a 3-step process: Identify feasible capacity intervals, expand activity sequence for selected intervals, propagate temporal constraints. If these three steps generate a feasible activity sequence, a bid containing this sequence is sent to the tasker resource. If the bid is accepted, then the resource profile is updated to include the new sequence of activities.

After a bid has been accepted, and the resources appropriately updated, the search algorithm change its focus to schedule additional pending service requirements. Depending on the configuration provided by the user through the service match, the search proceeds by scheduling the requirements of the current bidder, or goes back to the level of the previous requester, and schedule its next task. The sequence of services scheduled is also determined by the service match structure.

In terms of the global behavior of the application, the execution of the generated scheduler starts by reading a file describing all the top-level tasks, and all concrete resources available. The scheduling algorithm cycles through the top-level tasks, and expands the search following the structure defined by the service-match tree. If a top-level task cannot be satisfied using the available resources, it is marked “unschedulable” and discarded. Once there are no more pending tasks in the system, the scheduling algorithm finishes its execution and, if instructed to do so, outputs the schedule in text form, writes the schedule to an XML file, and/or displays the schedule on the GUI.

5.1.15. Planware Implementation – IDE for Domain Modeling and Planning

Planware’s implementation can be divided into two main components: the graphical interface, and the code generator.

Planware Interface implemented as a NetBeans Module

The interface component is written in Java and uses the open source configurable IDE platform NetBeans, which is an extensible integrated development environment designed to support multiple programming languages and formalisms. Additional capabilities are added to the NetBeans platform by writing modules using its customization API. A NetBeans module is just a JAR file (collection of compressed Java class files) that can be “installed” in the platform. A module can implement a number of different capabilities like syntax sensitive source code edition, compilation, execution, and debugging among others. The Planware module provides

1. An outline editor for editing activity machines based on a hierarchical representation of the models.

2. A graphical editor that allows the visualization and fast specification of activity machines.

3. A source code editor for more detailed specification of the models.

4. 4. Visualization tools for inspecting the results of executing the generated code on test data.

Developing resource models in this environment requires very little knowledge of Planware syntax. The set of syntax constructs is small and most of the model creation activity can be accomplished using just the outline and graphical editors.

A typical Planware resource model has less than a hundred lines of code, and can be created in a matter of minutes. A complete application model can be defined in few hours using a highly interactive environment.

The advantage of using an extensible platform like NetBeans is that the full application development and execution can take place in the same environment, and using the same interaction paradigm. Defining models, generating and compiling code, and executing the scheduler are all defined using the same basic set of actions and gestures.

Planware Code Generator implemented as a Specware Application

Planware code generator is implemented as an application layer on top of Specware, Kestrel's software synthesis platform. The Planware code generator translates the activity machines, and service match structure into an implementation of the algorithms and auxiliary data-structures described in section 5. Planware first generates an intermediate representation of the algorithms in MetaSlang, the specification language used internally by Specware. This representation is then further refined, optimized, and composed with appropriate library code to generate a highly optimized implementation of the scheduling application in some programming language.

For a problem model with five different resource types (approximately 500 lines of code), Planware generates an intermediate representation with around 10,000 lines of code. The size of final code in the target language usually increases by a factor of 3 or 4 in comparison with the generated code since all the library code used is included as part of the target implementation. The total synthesis time is on the order of 1 or 2 minutes for average size models – 4 or 5 different types of resources.

In terms of run-time performance of the generated schedulers, without any special heuristics added, models with four resource types running on data sets with thousands of tasks, and around 20 resource instances for each resource type, generate schedules in a matter of seconds. The runtime performance of the generated code was around 20% faster than the performance provided by scheduling applications previously developed manually by the authors for the domain of logistical deployment.

5.1.16. Planware Application Development Process

The application development process currently supported focuses on the generation of centralized, offline algorithms. The Planware domain analysis and application development process has the following steps:

1. **Requirement Acquisition –** The user interactively develops a model of the scheduling problem using the primitives previously discussed. This model describes the kinds of tasks and resources that are of concern. Figure 8 shows the graphical representation of a problem model.

   The problem model is formalized into a specification that can be read abstractly as follows: Given a collection of task instances and a collection of resource instances, find a schedule that accomplishes as many of the tasks as possible (or (approximately) optimizes the given cost function), subject to all the constraints of the resource models, and using only the given resources.

   The required and offered services of a resource express the dependencies between resource classes. The arrows between the resources in figure 8 represent the services required and provided. Planware analyzes the task and resource models to determine a hierarchy of service matches (service required matched with service offered) that is rooted in a task model.

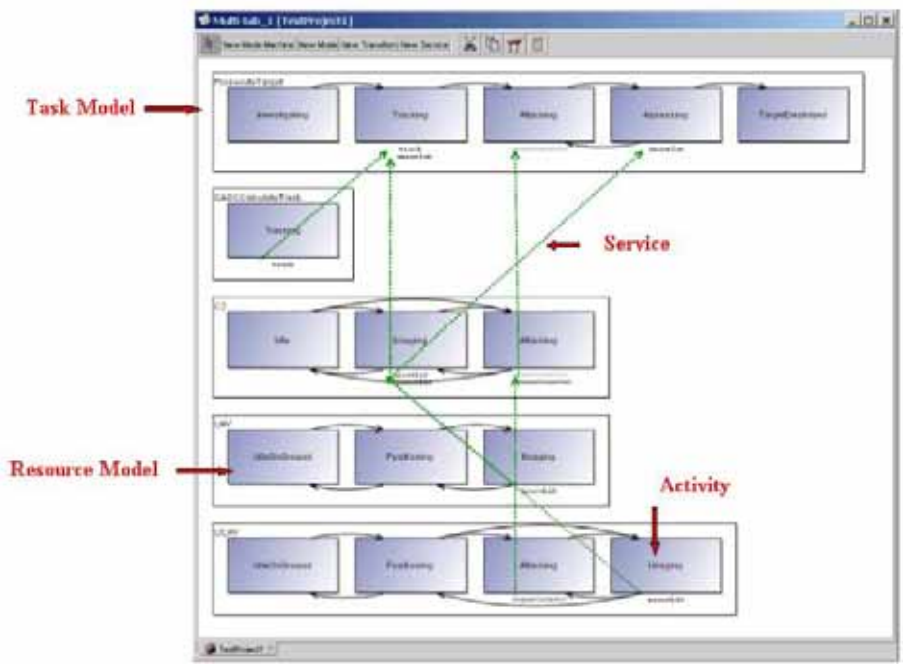


Figure 8: Planware Model for Sensor to Decision Maker to Shooter Scheduler

2. **Algorithm Design –** The problem specification is used to automatically instantiate program schemes that embody abstract algorithmic knowledge about global search and constraint propagation. The algorithm generation process follows the structure of the service hierarchy, resulting in a nested structure of instantiated search schemes.

3. **Datatype Refinement and Optimization –** Program schemes used by the generator are described in terms of abstract datatypes. After the generation of the scheduling algorithm, abstract datatypes are refined to concrete programming language types. Additional program transformations to provide further optimizations can then be manually or automatically applied to the resulting code.

4. **Code generation –** Finally code in a programming language (currently CommonLisp) is generated. In one recent example, we developed formal models for air cargo packages, cargo aircraft, air crews, and port facilities. In about one second Planware generates about 19000 LOC of source code comprising over 1780 definitions.

6. Addressing PCES Technical Challenges

Kestrel's efforts focused on the SDMS/W Challenge problems. The motivation and justification for this choice is the high level of automation software synthesis, and Planware in particular, can provide to the development of systems addressing these problems.

Planware is a generator for planning and scheduling applications. It automatically generates fully executable scheduling systems from high-level descriptions of tasks and resources. The generator takes as input a set of resource models and outputs the implementation of a planning and scheduling algorithm specialized to the resources and constraints specified in the model. The generated code takes as input data representing the parameters defining concrete instances of tasks and resources, and computes a set of resource reservations to perform all activities required by the tasks.

The algorithms generated by Planware implement a control cycle that decomposes high-level tasks into lower level tasks directly associated with resource consumption. The same concept is used in Hierarchical Task Network Planning [3, 4], and seems to be the principle behind the task network architecture [13, 14].

In PCES, our goal was to cast the program composition and system configuration problem as an extended version of a planning and scheduling problem. This allows us to model and reason about heterogeneous sets of resources and tasks in a uniform and integrated way. We have identified at least three classes of tasks and resources: operational, related to theater level activities and resources (e.g., aircraft, targets, UAVs, UCAVs, C2 nodes etc); configuration, related to set of software components needed by each operation resource to perform the operational activities (e.g., GPS, Weapon System, ATR, etc); and embedded or real-time, related to the lower level resources and tasks required to implement the functionality provided by each software component (e.g., CPU, Memory, bandwidth). In this formulation, software components are modeled as resources whose role is to manage the use of lower level resources to satisfy the high-level operational goals. The focus of the generated application is to manage resource utilization.

The generated scheduler can potentially handle any functional or non-functional component property that can be modeled as a resource utilization problem.

The concrete plan we proposed was to show that Planware can provide end-to-end resource management capabilities by providing a functionality equivalent or complementary to the one currently provided by the Task Network Architecture.

In the following paragraphs we address in more details Kestrel's contribution to the challenge questions.

6.1. Sensor to Decision Maker to Shooter/Weapon (SDMS/W) Challenges

Platform-level Resource Management

[QPR01] Accurately modeling platform capabilities (e.g., sensors, weapons, maneuverability)

Planware models represent platform capabilities in terms of activities and services they can provide, and resources it may use. The emphasis is on resource consumption.

[QPR02] Performing tradeoffs between platform tasking alternatives Planware models can represent several alternative plans to execute a given task (e.g., two imaging sensors that require different amounts of CPU and network resources). The scheduler algorithm generated is responsible for performing the tradeoffs among different alternatives. The user can specify in the models how to select among a number of different alternatives at different levels of the task decomposition process.

[QPR03] Determining and specifying relationships among activities performed across coordinated segments

Coordination is achieved through the decomposition of high-level tasks into lower level ones while guaranteeing that certain constraints are satisfied. For example, the requirement that a UAV starts transmitting an image to the C2 node when the UCAV is at a certain distance of the target can be accomplished by a high-level task that requires the UAV to be at a certain mode when the UCAV approaches the target. The scheduler will be responsible for coordinating the actions and guaranteeing that the lower level resources needed are available at the same time in both platforms. The implementation of coordination of different resources in real-time is out of our scope. The generated scheduler only guarantees that the coordination is feasible at configuration time. As execution evolves, a real-time coordination mechanism may require a complete different set of models that need to be integrated into the platform.

[QPR04] Assessing task granularity and specifying loadable task networks

The scheduler will be responsible to decomposing the task network into its lower level resource requirements. The scheduler will identify if a given task network can be supported in the current configuration, or if the configuration required can be feasibly implemented given the existing configuration. I am not sure what “task granularity” means in this context.

[QPR05] Effectively and dynamically scheduling CPU resources for platform tasking needs

In this phase, we are focusing our efforts on the modeling and decomposition of tasks into its resource requirements, and on the generation of priority policies that would guarantee the feasibility of the required activities at design or configuration time. We would first rely on the currently available dispatching techniques to execute our schedule. In the next phase, once we understand the task decomposition and resource usage, we are planning to incorporate dynamic schedulers generated by Planware into the OEP platform.

Multi-Network Resource Management

[QMR01] Reasoning about, quantizing, and prioritizing cross-network bandwidth needs and importance

The main strength of Kestrel's generated schedulers is the ability of scheduling multiple resources in distributed and coordinated fashion. Planware models, and schedulers, can represent multiple network types, and implement a number of different allocation policies. Each network class or instance can have its own allocation policy, and the scheduler will manage bandwidth utilization according to the specified policies.

[QMR02] Adaptive allocation of resources to mitigate contention while preserving QoS

Adaptive resource allocation is related to dynamic scheduling. The schedulers generated implement reactive capabilities that can be used to perform load balancing and other resource management capabilities. Similar to the comments on dynamic scheduling, we would not address these problems in the first phase.

Dynamic Tactical Link Management

[QDT01] Data management strategies to route high priority data through the network in a timely manner

[QDT02] Dynamic timeslot allocation to reallocate network time slots based upon changing mission modes.

Planware schedulers can naturally address both of these problems. In a first phase, we will assume that we each mission mode has a pre-defined requirement for network usage. In the future, we will explore more dynamic mechanisms that would allow the timeslot allocation to be sensitive to other environment factors in addition to mission mode.

6.2. System-wide QoS Management Challenges

Kestrel's solution is aiming at "combining individual QoS solutions in a seamless manner to get end-to-end, dynamic QoS management among competing elements is the end challenge".

End-to-end and Roundtrip Sensor Data Delivery, Processing, and Control:

[QER02] Provide accurate and timely delivery of control signals from C2 to shooter/weapon

[QER03] Compose these into a round-trip SDMS/W system

[QER04] Mediate the contending requirements of multiple participating nodes

Our goal to address all these questions is to develop rich resource and task models representing all the relevant entities that need to be managed to guarantee end-to-end QoS. Our focus is on the "timely" delivery of data and control signal while naturally managing contention. The "accurate" and "sufficient" are outside the scope of the scheduler, unless, of course, they can be represented as some kind of measurable resource capacity.

Rapid Reconfiguration and Reaction to Dynamic Conditions and Changing Missions:

[QRR01] Support rapid mission mode changes

[QRR02] Operate effectively (by mission standards) in hostile, changing environments

[QRR03] Support changing numbers and types of participants with shared resources

[QRR04] Proactively anticipate mission mode transitions, predict reconfigurations, and adapting

As previously mentioned, we de-emphasized the dynamic aspects of the problem by proactively trying to anticipate change as much as possible. We investigated how to provide different schedulers for different mission modes, and how to transition between them as changes occur. We still need more experience with changes in mission modes to be able to adequately address these questions. Our plan is to move into the more dynamic scenarios as our task networks models mature.

6.3. System Evolution Challenges

[SEV01] A means by which architectural aspects can be expressed and mapped from one architecture (legacy) to another (future).

[SEV02] Model translators that migrate legacy models to a new paradigm based on the mapping expressed above.

[SEV02] Code translators that migrate legacy code to a new paradigm based on the mapping expressed above.

Kestrel's synthesis approach addresses some evolution challenges different from the ones listed above. The generator approach allows the problem specification, and the underlying technology used to solve the problem to evolve separately. Since we expect the problem specification to change faster than the underlying technology, the models defining this specification can be modified in a matter of minutes, and a complete new application, obeying the same architectural constraints, can be generated almost immediately. As the technology evolves (e.g., new algorithms, new languages, new paradigms, new modeling formalisms), the generators need to be adapted, and previously generated systems must be re-generated from existing models. The time to adapt the generator engine can be on the order of weeks or months depending on the scale of change.

7. Sensor to Decision Maker to Shooter/Weapon (SDMS/W)

This section presents models of high-level tasks and resources for an example scenario based on the proposed PCES Sensor to Decision Maker to Shooter/Weapon challenge problem.

The approach proposed by Kestrel for end-to-end resource management was to use the Planware modeling and synthesis tool to model tasks and resources at different levels of abstraction, starting with high-level operational models, and mapping them to real-time tasks and resources. The resulting hierarchical models can be used not only for the automatic generation of operational schedulers, but also for the generation of real-time resource managers.

The steps that are needed to obtain these hierarchical models are the following:

1. Model operational tasks and resources (e.g. C2, UAVs) as state or activity machines, determining operational resource allocation and defining the task network for target prosecution.

2. Identify the system components and behavior required in each mode (e.g. video delivered to the C2 node at a specified rate and quality when the UAV is sensing the target) and add them to the model as a second layer, determining the configuration of each resource in each mode and decomposing the task network for target prosecution into a modes/capabilities model.

3. Map the required components and behavior to embedded resources that are added to the model as a third layer, decomposing the task network for system configuration in each mode into a real-time schedule.

   The generation of an embedded resource manager for a demonstration on the Boeing OEP required the following additional steps that were not completed:

4. Enable a Planware scheduler to generate schedules in a format that can be used in the OEP.

5. Enable a scheduler in the OEP to be configured with the schedules generated by Planware for each mode, and reconfigured at each mode transition.

This document addresses the modeling steps. The generation of an embedded resource manager will be discussed in a separate context. Section 7.1 and Section 7.2 present an example operational scenario and high-level models. Section 7.3 and Section 7.4 discuss the decomposition of the activities in the high-level model and the mapping to embedded resources.

7.1. Example Operational Scenario

The example scenario for the end-to-end resource management model is based on the Boeing BBN UAV OEP scenario. The operational resources included in this scenario are CAOC, UAV C2 (UAC2), UCAV C2 (UCC2), and a number of UAVs and UCAVs.

The demo scenario proposed by Boeing includes the following steps: *Track, Identify, Deal with Failures (Links, Visibility, Equipment), Attack, (Battle Damage Assessment (BDA), Re-attack* and *Re-BDA*.

The sequence of actions are described as:

1. Ground sensor does initial detection of mobile target and notifies CAOC.
2. CAOC sends UAV to investigate.
3. UAV provides video (perhaps streaming) to UAV C2.
4. UAV C2 digitizes and creates images and sends them to CAOC.
5. 5. CAOC performs ATR to turn image into a track.
6. Automatic Target Recognition component (ATR) is re-locatable and could happen on any of the C2 or AV nodes.
7. CAOC creates VTF-like entity using the track and a link to the UAV image stream.
8. CAOC tasks UCAVs with VTF.
9. UCAV C2 receives tasking from CAOC and assigns UCAV 1 and sensor and shooter and provides the track to UCAV 1.
10. UCAV 1 starts to prosecute the track and then loses comm with UCAV C2.
11. UCAV 2 establishes comm with UCAV C2 and relays comm to UCAV 1.
12. UCAV 1 prosecutes target.
13. UAV provides BDA to CAOC (could be UCAV).
14. CAOC determines that target not destroyed and tasks UCAVs with re-attack.
15. UCAV 2 prosecutes target.
16. UAV provides BDA to CAOC (could be UCAV). 17. CAOC determines target destroyed.

The main task of the CAOC in the scenario is to prosecute a target once that target is detected (by a sensor external to the scenario). The UAVs and UCAVs provide services or capabilities (through the UAC2 and UCC2) to support that task:

The UAVs provide sensor products (video) that are converted to images at the UAC2 and sent to the CAOC.

The UCAVs may also provide sensor products that are converted to images at the UCC2 and sent to the CAOC.

The UCAVs are tasked (through the UCC2) to engage the targets.

Both UAVs and UCAVs may provide Battle Damage Assessment (BDA) to the CAOC after each engagement.

The first three services correspond to the AV roles of UAV Sensor (UAS), UCAV Sensor (UCS), and UCAV Weapon (UCW).

BDA is assumed to be a service of the CAOC or C2 nodes: The C2 nodes receive sensor products from the AVs and either process them locally or send them to the CAOC as images. The AVs in this case perform the role of UAS/UCS.

7.2. High-Level Models

Figure 9 shows Planware activity-machine models of the high-level tasks and resources for the UAV scenario, including the top-level task for target prosecution (*ProsecuteTarget*) and the top-level resources (CAOC, UAC2, UCC2, UAV, UCAV).

7.2.1. High-Level Tasks

Figure 10 shows the source code for a task model. The complete source code for this model is included in Appendix A. The top-level task *ProsecuteTarget* is modeled as an activity machine with a single activity, which requires a service from the top-level resources:

- Prosecuting is the single activity of the *ProsecuteTarget* activity-machine.
    - caocProsecuteTarget(targetLocation) is the service required by the task from the CAOC in that activity. The service request passes the location of the target to the CAOC.

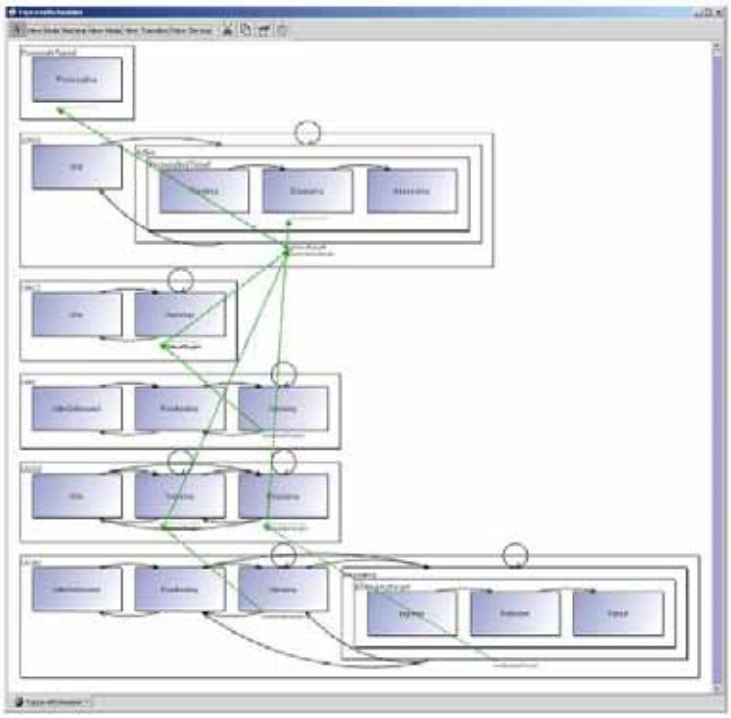


Figure 9: Operational Models for Target Prosecution

7.2.2. Resources

CAOC: The CAOC has two main activities or modes:

Idle is the initial state of the CAOC, and does not provide or require any service.

Active is an activity that can be decomposed into three sub-activities: Tracking, Engaging, and Assessing. The CAOC requires and provides the following services when in Active mode:

- Provides *caocProsecuteTarget(targetLocation)* to the task.

- Requires *c2SenseTarget(targetLocation, avSensorId).* The location of the target is passed in the variable targetLocation from the requesting task to the provider of the service, which assigns an AV sensor to investigate. The service returns the identity of the AV sensor when the sensor is in position. We assume the CAOC has access to images derived from the AV sensor during all the time that it is in one of the sub-modes of Active mode.

The Active mode has three sub-modes:

Tracking: in this mode, the CAOC, using the images derived from the AV sensor, calculates the target track, and therefore the current target location. This mode requires no service.

Engaging: in this mode, the CAOC tasks a C2 node to assign a UCAV to engage the target.

The engaging activity requires the service *c2EngageTarget(currentTargetLocation, avSensorId, avWeaponId)*. The CAOC passes the current location of the target and the previously assigned *avSensorId* to the provider of the service. The service returns the identity of the UCAV assigned to engage the target, when it is in position.

Assessing: this mode is for battle damage assessment. The CAOC uses images derived from the AV sensor (again, with the previously assigned avSensorId) to determine if the target has been destroyed.

The CAOC goes through the three sub-modes and then has the option of transitioning to Idle (if the target is destroyed and there is no other target to prosecute) or back to Active.

```
mode-machine Target is
   constant targetLocation         : Location
   constant earliestEngageTime : Time
   constant latestEngageTime : Time
   external-variable startTime : Time
   external-variable endTime : Time
   external-variable duration : Duration
   internal-variable requiredEngageTargetCapacity: Capacity
   initial final mode Targeting has
   required-invariant engageTarget(startTime, endTime, duration, targetLocation)
      constraint startTime >= earliestEngageTime
      constraint endTime <= latestEngageTime
   end-mode
   transition from Initialization to Targeting when {} is
   { startTime                          := timeZero,
     endTime                            := timeInfinite,
     duration                           := oneTimeUnit,
     requiredEngageTargetCapacity   := unitCapacity}
end-mode-machine
```

Figure 10: Source Code for Task Model

The assignment of the same AV sensor for the three sub-modes is a choice that may or may not be desirable: The AV sensor being used for Tracking will already be in position and sensing the target when the CAOC moves into Engaging and Assessing. On the other hand, the assignment of the same AV sensor for the three sub-modes prevents the UCAV that engages the target from being used for tracking or damage assessment. Moreover, the AV sensor might not be able to stay with the target long enough because of fuel limitations.

The assignment of independent sensors for the three sub-activities would enable the UCAV that engages the target to perform tracking or damage assessment as well. On the other hand, constraints would be needed to prevent the scheduler from allocating four AVs to each target when only two are needed.

The sub-modes Tracking and Assessing require no service from other operational resources, but will require service from lower-level resources (ATR and BDA modules) when these are included in the model.

UAC2/UCC2: The UAC2 has two modes, Idle and Sensing.

Idle: is the initial mode of the UAC2.

Sensing: in this mode, the UAC2 assigns a UAV to sense the target, receives video from the UAV, and sends images to the CAOC.

- Requires *uasSenseTarget(targetLocation, avSensorId)*. The target location is passed to the provider of the service, and the service returns the identity of the provider (UAV sensor).

- Provides *c2SenseTarget(targetLocation, avSensorId)* to the CAOC. The UAC2 receives the target location from the requester of the service, and returns the identity of the UAV sensor.

The UCC2 Idle and Sensing modes are identical to the UAC2 modes, except that the required service is renamed ucsSenseTarget(targetLocation, avSensorId) to reflect the fact that the UCC2 controls UCAVs and not UAVs. The UCC2 adds a third mode:

Engaging: in this mode, the UCC2 assigns a UCAV to engage the target.

- Requires *ucwEngageTarget(targetLocation, avSensorId, avWeaponId).* The UCC2 passes the target location and identity of the AV sensor to the UCAV, and receives back the identity of the UCAV.

- Provides *c2EngageTarget(targetLocation, avSensorId, avWeapontId).* The UCC2 receives the target location and identity of the AV sensor from the CAOC, and passes back the identity of the UCAV.

UAV/UCAV: The UAV and UCAV require a more elaborate model than the other resources to account for movement from the base to the target and vice versa. The model for the UAV must include a unique identifier that can be passed to the consumers of the sensor products it generates, and also parameters to calculate the flight time to the target and the maximum range of the UAV:

- *avId*: unique identifier for UAV.
- *baseLocation*: location of the base, assumed to be fixed.
- *maxSpeed:* maximum speed of the UAV.
- *maxFuelLevel*: amount of fuel that the UAV can carry (assumed to be replenished at the base).
- *burnRatePerDistance*: amount of fuel consumed per distance traveled (when traveling to the target and back).
- *burnRatePerDuration*: amount of fuel consumed per time unit when sensing the target.
- *maxMunitionCapacity*: number of targets that the UCAV can engage before rearming. The UCAV, in addition, must have a munitions capacity that limits how many targets it can engage before returning to the base to rearm:

The actual locations of origin and destination for each mode, fuel level, and munition capacity (for the UCAV) are kept as internal mode variables in the model. The fuel level and munition capacity must be non-negative at all times.

The UAV has the following modes:

IdleOnGround: this is the initial mode for the UAV. The origin and destination must be equal to the base location. The fuel level is replenished while the UAV is in this mode. There may be a minimum duration for each stay on the ground.

Positioning: this is the mode in which the UAV is flying to the target or back. The appropriate amount of fuel is subtracted from the fuel level to account for the distance. The duration of the flight must be at least the time it takes to fly from baseLocation to targetLocation without exceeding maxSpeed.

Sensing: this is the mode in which the UAV is sensing the target. The origin and destination must be equal to the target location. The fuel level is decreased to account for the time the UAV stays in this mode.

- Provides *uasSenseTarget(targetLocation, avId)*. The UAC2 passes the target location to the UAV and receives back the identity of the UAV.

The UCAV has the same IdleOnGround, Positioning, and Sense modes, with the munition being replenished (as well as the fuel level) while IdleOnGround, and the sensing service renamed to ucsSenseTarget(targetLocation, avId). The UCAV has one additional mode Engaging:

Engaging: this is the mode in which the UCAV engages the target.

- Provides *ucwEngageTarget(targetLocation, avSensorId, avId)*. The UCAV receives the target location and the sensor identity from the UCC2, and returns its own identity.

- The mode Engaging has three sub-modes:

  Ingress: Representing the approach to the target.

  Release: Representing the deployment of the weapon or munition.

  Flyout*:* Representing the activity of leaving the target area.

  The duration of these modes is assumed to be constant for each one. Fuel level is adjusted according to duration. Munition is decremented in activity Release. After Flyout, the UCAV may go to Positioning – to fly back to the base, or to Sensing – for battle damage assessment, or back to Engaging – if there are more targets at the same location.

7.3. System Configuration Models

The next step after defining the models for the operational scenario is to identify the system components that are active in each mode for each resource, determining the configuration of the components for each mode of the resource. Figure 11 shows and example of this decomposition of each mode into components and behaviors for the UCAV in the UCW role. The components that are active or inactive in the different modes are shown together with their different priorities in each mode, indicating a change in behavior from one mode to the next. The mode sequence in Figure 11 corresponds in our model to the following mode sequence:

Positioning → Ingress → Release → Flyout → Sensing

The assignment of priorities to each component in each mode in Figure 11 needs to be clarified. Changes in behavior may be more significant than the change of priority indicated in the figure: the Recon component in Figure 11 may be capable of capturing video and sending it to the UCC2 with various settings of quality and data compression. The system configuration for each mode should include a list of the possible settings for each component and the requirements for each mode, e.g., higher frame rate when navigating, greater image resolution when doing damage assessment.

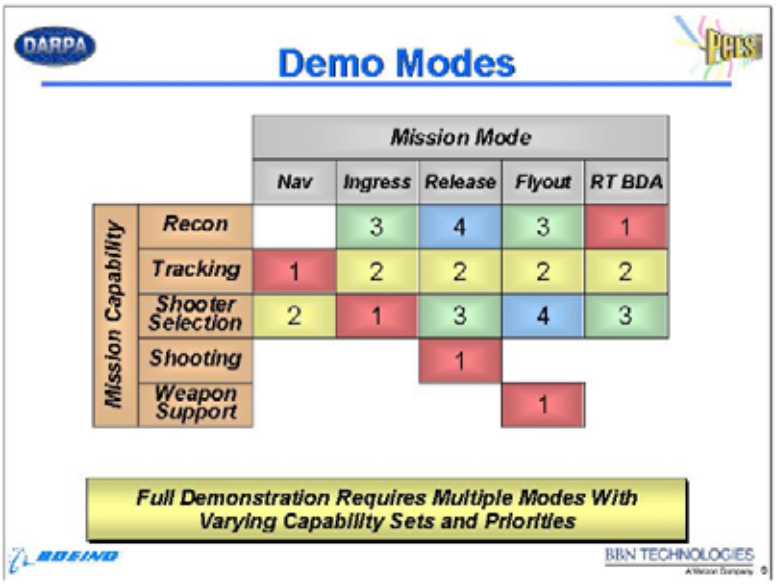


Figure 11: System Configuration for UCAV in UCW Role

These components are added to the Planware model as resources that provide a service for each mode (e.g., a resource Recon that provides a recon service to the UCAV in Ingress, Release, Flyout, and Sensing modes, and in turn requires services from the embedded resources).

7.4. Real-Time Resource Management

The final step after the decomposition of the modes into a component configuration is to map each component to real-time resources that they require. For instance, the Recon component might require an image to be sent from an on-board sensor at a certain minimum rate.

The real-time resources may also be modeled as resources that provide a service to the components. Figure 12 shows a configuration resource Recon that provides the recon (priority,. . .) service to the UCAV and in turn requires a service sense at rate(rate, quality, . . .) from the embedded resource OnBoardSensor.

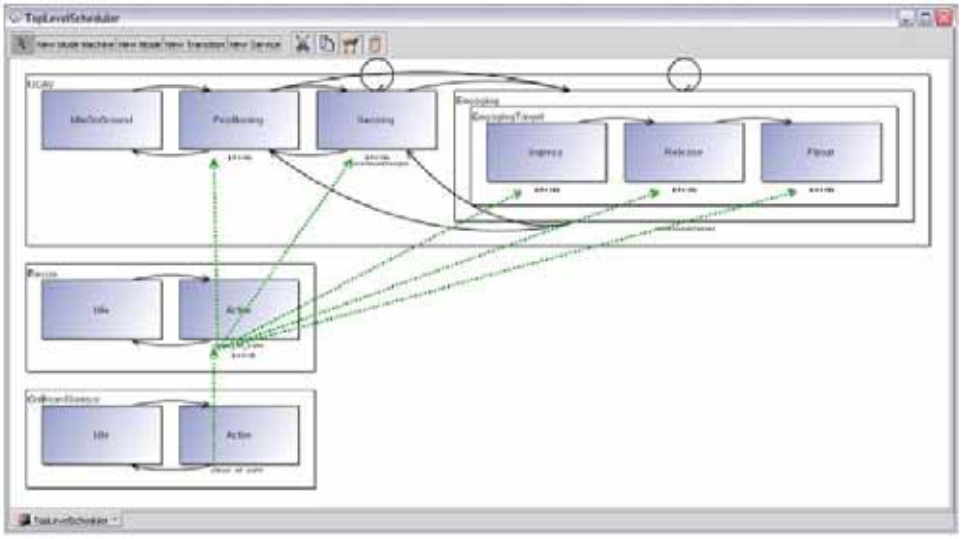


Figure 12: Configuration resource (Recon) and real-time resource (OnBoardSensor)

Figure 12 illustrates the role of the configuration resource in mapping the requirements of the operational model (such as the mode-dependent priority parameter of the recon service) onto suitable configurations of the embedded resources (such as the rate and quality parameters of the sense at rate service).

One configuration resource may require service from multiple embedded resources, for instance a network link might be required to transmit the data from the on-board sensor back to the C2. Figure 13 shows a model representing how the different mission modes described in Figure 11 could be mapped to the lower-level embedded resources needed.

The model in Figure 13 has a state machine representing the different mission modes described in Figure 11: Navigation, Ingress, Release, Flyout, and BDA. The mission resource behaves as an abstract resource whose role is to provide task decomposition into the lower level functionality required.

Each mission activity require services from state machines representing the diffeent mission capabilities: *MissionRecon, MissionTrack, MissionShooterSelection, MissionShoot,* and *MissionWeaponSupport*. These are also abstract resources.

The mission capabilities required by each mission mode and their priorities may change from mode to mode. Nav mode requires service from MissionRecon, MissionTrack and MissionShoot erSelection. The priority of each service can be set by changing the order in which they appear in the service match.

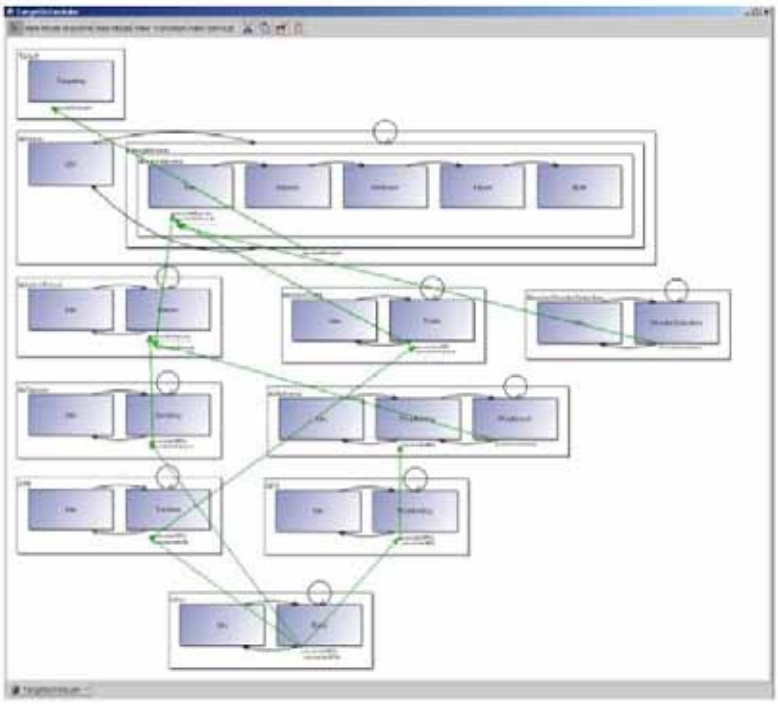


Figure 13: Planware Model Representing Embedded Resources

The mission capabilities require service from concrete resources that correspond to software components that may reside on an embedded platform (UAV, UCAV) or on a command center (UAC2, UCC2). The resources included in the application include AVSensor, AVAirframe, ATR, and GPS. These components are comparable to the software components in the OEP BasicTNA scenario (OEP Scenario 1.11).

The software components in turn require service from lower-level resources such as CPU and network links. The example shows the CPU resource.

Notice that each platform (UAV, UCAV, UAC2, UCC2) is represented by a collection of resources (e.g., a UAV will consist of an AVSensor, AVAirframe, GPS, and a CPU). The platforms are not resources in themselves, but appear as a homeId constant that is the same for each resource belonging to that platform (see appendix A). This arrangement allows services to be relocated from one platform to another or constrained to a single platform as appropriate, e.g.:

- *MissionTrack* capability requires service from an ATR. The scheduler is free to choose an ATR module where it is available, either on a UAV or on a UAC2.
- *MissionRecon* capability requires service from an AVSensor to locate and track the target, and an AVAirframe to be positioned at the target location. These two resources must reside on the same UAV platform. The scheduler must choose an AVSensor and an AVAirframe with matching homeIds.

Each software component will require service from a lower-level resource on the same platform (CPU, in our example). The homeId of the software component is passed to the lower-level resource as an input variable and required to match the local homeId.

8. Comments on Reasoning's ACE/TAO Analysis Results

Reasoning performed a static analysis of the ACE/TAO code base using Illuma, an automated software inspection tool capable of detecting structural defects in software. Illuma detected 8 defects that could potentially affect the behavior and quality of applications using the ACE/TAO framework. Considering the large number of files, 736 files, and the large number of lines of code, over 140,000 LOC, composing the ACE/TAO distribution, the fact that Illuma was only capable of detecting only 8 structural defects in 7 of the files, lead us to the conclusion that the ACE/TAO implementation is a well engineered, mature software system that has been extensively and thoroughly tested.

The detailed report provided by Reasoning is provided as an appendix to this report. This section describes the possible implications or impact the identified defects would have in an application developed using the ACE/TAO orb infrastructure and services. Please refer to Reasoning's Illuma Defect Data for ACE/TAO report for a more detailed description of the defects. Reasoning's comments on the defect analysis address the generic impact of the defects in a software system. The current document addresses the defects in the context of the potential use of the affected code and its implications.

Defect Number 1:

Location: ace/Configuration.cpp @ ACE Configuration Win32Registry::resolve key : line 1059

Defect type: Memory leak

Analysis: The variable identified as the source of the memory leak, temp path, is a pointer to a fixed-length character array and is allocated no more than once per invocation of the function. It would appear that this could indeed be a small source of persistent memory leaks, but the functions defined in the affected file are almost exclusively used at application startup time for a single-pass configuration stage.

Impact: Minor

Defect Number 2:

Location: ace/Log Msg.cpp @ ACE Log Msg::instance : line 34 Defect type: Memory leak

Analysis: The variable identified as the source of the memory leak, tss log msg, is held within thread-local storage, which is automatically reclaimed when the lightweight process using it exits. In this case, a singleton logging thread holds the only reference to this assigned memory, and continues to utilize the variable as long as the application is running.

Impact: None

Defect Number 3:

Location: ace/TP Reactor.cpp @ ACE TP Reactor::remove handler : line 275

Defect type: Memory leak

Analysis: The TP Reactor class is a multi-threaded event callback handler for event driven code. The method affected, remove handler(ACE Handle Set, ...), removes all correlations between events and handlers in a TP Reactor instance, and so, is unlikely to be used outside of application shutdown code. For normal event handler maintenance, an alternative method exists which removes only one event handler from the TP Reactor's active set.

Impact: Minor

Defect Number 4:

Location: ace/MEM Acceptor.cpp @ ACE MEM Acceptor::accept : line 104

Defect type: Null pointer dereference

Analysis: The defect analysis appears to have mistakenly flagged a null pointer dereference when a parameter is being written to, not read from. That is, a null pointer is passed into a constructor call. The constructor then stores the pointer location in the newly-assigned object instance, but does not dereference it. Obviously, simply storing (or passing as a method argument) a null pointer does not necessarily indicate a program error. (Note: Our understanding of the internals of the class whose constructor is called is limited, but, since this possible error is in an extremely heavily utilized portion of the network socket-handling code, it seems highly unlikely that a null pointer error would have gone unnoticed.)

Impact: Probably none

Defect Number 5:

Location: ace/SString.cpp @ ACE SString::ACE SString : line 394

Defect type: Null pointer dereference

Analysis: The basic potential error reported here could only occur in cases where a single SString class instance was shared among multiple active exectuing threads - basically, there is an instance variable set to 0 only two lines above the potential null pointer bug which will prevent the error from happening unless another thread preempts the one calling the method, changes the value, and immediately allows the first thread to resume. That case is highly unlikely, as the ACE documentation for this class states

that it is a special-purpose simple string representation used solely for string-to-integer mapping tables requiring specific properties for their key type, and should be avoided in favor of the CString class for general use.

Impact: Minor to none.

Defect Number 6:

Location: ace/Svc Conf y.cpp @ ace get module : line 1535

Defect type: Null pointer dereference

Analysis: This entire method is a little bit confusing since the source file in question was automatically generated from a YACC grammar, and not hand-written. It would appear, however, that in cases where no runtime exception support exists in the C++ compiler used, it might be possible to give sufficiently skewed input to this function in order to generate a null pointer error. However, as this is a purely internal function not visible or used outside this particular source file, the triggering of any such problem seems highly unlikely.

Impact: Minor

Defect Number 7:

Location: ace/Configuration.cpp @ operator = = : line 362

Defect type: Uninitialized variable

Analysis: The variable in question here is of type u int, which is a typedef alias for unsigned int. Modern C and C++ compilers implicitly initialize statically allocated primitive variables like this one to a value of 0. The assumption here is that the check for that case failed here because of the aliased type name, or some other minor syntactic issue that confused the analysis.

Impact: None, except with an extremely buggy compiler

Defect Number 8:

Location: ace/Configuration.cpp @ operator= = : line 386

Defect type: Uninitialized variable

Analysis: See above – this is an almost identical case, within the same method as the above issue.

Impact: None

9. C Code Generator

This section describes the Metaslang-to-C-Generator, the task of the which is to produce C-code from a Metaslang specification as an alternative to the Lisp generation functionality that is already part of the Specware system.

In the current version of the C-Generator, the C-code that is generated from a Metaslang specification is given in a purely functional style. That means that the functional Metaslang code is not analyzed wrt. possible optimizations involving e.g. destructive updates of data structures. The transformations from the Metaslang code to the resulting C-code mainly removes all ”functional features” (e.g. local and anonymous functions, currying, pattern matching) and translates the representation of data structures to corresponding C data structures.

The generated C-Code uses a public-domain garbage collector as storage management tool. Garbage is produced whenever a data structure such as a product, co-product, record, or closure type is created in the C code.

The C code generation for a Metaslang specification directly succeeds the type-checking step. Several steps are performed prior to the actual C-code generation in order to transform the features that cannot directly be mapped to C into those for which it is possible. Among these steps are pattern matching compilation, lambda-lifting, and arity normalization.

Metaslang sorts and operators undergo a number of transformations before they are actually mapped into C code. Among them are:

- Type variables are completely removed from the sort definitions, which means that e.g. a List of Nat has the same representation as a List of Strings in the resulting C code.

- Metaslang sorts without a definition are mapped to the sort Any, which is translated to the C type void*.

- Subsorts and quotients are identified with their base sort; no distinction is made in the generated C code between the subsort or quotient sort and the corresponding base sort.

- Sorts are ”flattened” meaning that a transformed sort only refers to base sorts in its definition, where a base sort is a reference to another sort definition. That means, for instance, that if in the Metaslang source a sort is defined to be a co-product of different products, it will be transformed to a co-product of newly introduced base sorts, each of which defining one of the original product sorts. For example, if the original sort definition has been

```
sort BTree = Empty | Node (BTree * Nat * BTree)
```

then a a new sort definition for BTree * Nat * BTree will be created resulting in the following representation:

```
sort Product_1 = BTree * Nat * BTree
sort BTree = Empty | Node Product_1
```

Similar transformations are carried out for products containing co-products, and all other combinations. These transformations are performed on all sorts in the input specification regardless whether they occur in sort definitions, in signatures of operator definitions, or at any other place where sorts are allowed.

- Arrow sorts, i.e. sorts representing function signatures are all mapped to the same C type named ”Closure”, which is defined in $SPECWARE2000/c-lib/meta-slang.h. A closure is a structure containing a reference to the function to be called as well as information about the environment in which it should run. In case of Metaslang, the environment is given Metaslang Sort C Type Literals by a tuple of all free variables in the function definition term and their current values. A function represented by this closure type can be called using the built-in ”applyClosure” C function, which is also defined in the above mentioned file.

- The primitive sorts Char, Boolean, Nat, Integer, and String are translated to corresponding types in C:

MetaSlang Sorts	**C Type**	**Literals**
Boolean.Boolean	unsigned int	TRUE, FALSE
Char.Char	char	C character constants
Nat.Nat	unsigned int	C int constants
Nat.PosNat	unsigned int	C int constants
Integer.Integer	int	C int constants
String.String	char*	C string constants
Any	void*	
()	Void	

In the context of the translation of operator signatures we distinguish between operators representing functions and those representing terms. In Metaslang this means that in the first case the operator has a sort that can be reduced to an arrow sort “domsrt ! codomsrt”; in the latter case, the sort of the operator is different from an arrow sort. We refer to these different cases as function operators and constant operators. Function operators are translated into C

functions, while constant operators are translated to global C variables. Depending on the type of the variable, either the initialization is done directly at the place the variable is introduced, or a separate initialization function is generated in case the type of the variable does not correspond to a primitive C type.

The C code generator can also be used to refine sorts and operators that do not have definitions in Metaslang. This is a especially very useful in integrating the Metaslang specifications into existing C code, as is has been done in the context of the PCES project.

10. Concluding remarks and future work

During this program we developed a detailed model for the challenge problem sensor to decision maker to shooter provided by the join Boeing-BBN OEP. This model can provide the functionality to substitute or complement the task network module develope by Boeing.

The final integration of the generated scheduler with the OEP platform and experimentation with the provided OEP scenarios was not finalized as a result of resource limitations. We would like to explore a closer integration of the Planware generated scheduler with the Event Channel component currently implemented in the OEP platform. Ideally, the generated schedulers could provide an integrated component that would substitute both the task network and the event channel component. The rationale for using a generator to provide a scheduling component was to include high-level, domain knowledge into the real-time scheduling decision loop.

We also performed an extensive and thorough static analysis of the TAO ORB source code: Very few problems have been detected, and most of the identified problems will have little or no effect on the runtime behavior of the system.

References

[1] Blaine, L., Gilham, L., Liu, J., Smith, D., and Westfold, S. Planware: Domainspecific synthesis of high-performance schedulers. In Thirteenth Automated Software Engineering Conference (Los Alamitos, CA, October 1998), IEEE Computer Society Press, pp. 270–80.

[2] Doerr, B., Venturella, T., Jha, R., Gill, C., and Schmidt, D. Adaptive scheduling for real-time, embedded information systems. In Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC) (St. Louis, Missouri, October 1999).

[3] Erol, K., Hendler, J., and Nau, D. S. Semantics for HTN planning. Tech. Rep. CS-TR-3239, Carnegie Mellon University, 1994.

[4] Erol, K., Hendler, J. A., and Nau, D. S. Complexity results for HTN planning. Annals of Mathematics and Artificial Intelligence 18, 1 (1996), 69–93.

[5] Gill, C., Schmidt, D., Loyall, J., Schantz, R., Levine, D., and Kuhns, F. Applying adaptive middleware to manage end-to-end qos for next-generation distributed applications. submitted to the Special Issue of Computer Communications on QoS-Sensitive Network Applications and Systems (2002).

[6] Gill, C. D., Levine, D. L., and Schmidt, D. C. The design and performance of a real-time CORBA scheduling service. Real-Time Systems 20, 2 (2001), 117–154.

[7] Huang, J., Wang, Y., and Cao, F. On developing distributed middleware services for qos- and criticality-based resource negotiation and adaptation. Real-Time Systems 16, 2-3 (1999), 187–221.

[8] Huang, J. J., Jha, R., Muhammad, M., Lauzac, S., Kannikeswaran, B., Schwan, K., Zhao, W., and Bettati, R. RT-ARM: a real-time adaptive resource management system for distributed mission-critical applications. In IEEE Workshop on Middleware for Distributed Real-time Systems and Services (San Francisco, CA, December 1997).

[9]Hutchison, D., Coulson, G., and Campbell, A. Quality of service management in distributed systems, 1994.

[10] Klein, M., Ralya, T., Pollak, B., Obenza, R., Gonza, M., and Harbour, L. A Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real Time Systems. Kluwer Academic Publishers, 1993.

[11] Krsti/’c, S., Launchbury, J., and Pavlovi´c, D. Categories of processes enriched in final coalgebras. In Proceedings of FoSSaCS 2001 (2001), F. Honsell, Ed., vol. 2030 of LNCS, Springer Verlag, pp. 303–317. [12] Levi, S., and Agrawala, A. Real Time System Design. McGraw Hill Co., 1990.

[13] McBryan, B., and Joy, M. Rotorcraft pilot’s associate hierarchical planning. In Proceedings of the 55th AHS Annual Forum (Montreal, Canada, May 1999), AHS.

[14] McBryan, B., and Joy, M. Rotorcraft pilot’s associate shared memory task network architecture. In Proceedings of the 55th AHS Annual Forum (Montreal, Canada, May

1999), AHS.

[15] Mok, A. Towards mechanization of real time system design. In Foundations of Real Time Computing: Formal Specifications and Methods, A. Tilborg and G. Koob, Eds. Kluwer Academics Press, Boston, MA, 1991.

[16] Pavlovic, D., and Smith, D. R. Composition and refinement of behavioral specifications. In Proceedings of Sixteenth International Conference on Automated Software Engineering (2001), IEEE Computer Society Press, pp. 157–165.

[17] Rosu, D., Schwan, K., Yalamanchili, S., and Jha, R. On adaptive resource allocation for complex real-time applications. Tech. Rep. GIT-CC-97-26, Georgia Institute of Technology, 1997.

[18] Smith, D. R. KIDS: A knowledge-based software development system. In Automating Software Design, M. Lowry and R. McCartney, Eds. MIT Press, Menlo Park, 1991, pp. 483– 514.

[19] Smith, S., and Becker, M. An ontology for constructing scheduling systems. In Proceedings of the AAAI Spring Symposium on Ontological Engineering (Palo Alto, CA, April 1997), pp. 120–129.

[20] Srinivas, Y. V., and Jullig, R. Specware: Formal support for composing software. In Mathematics of Program Construction (July 1995), pp. 399–422.

[21] Stankovic, J. A., Spuri, M., Natale, M. D., and Buttazzo, G. C. Implications of classical scheduling results for real-time systems. IEEE Computer 28, 6 (1995), 16–25.

Appendix A Planware Model for SDMS/W Problem

```
Target =
mode-machine Target is
constant targetLocation : Location
constant earliestEngageTime : Time
constant latestEngageTime : Time
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
internal-variable requiredEngageTargetCapacity: Capacity
initial final mode Targeting has
required-invariant engageTarget(startTime, endTime, duration, targetLocation)
constraint startTime >= earliestEngageTime
constraint endTime <= latestEngageTime
end-mode
transition from Initialization to Targeting when {} is {
startTime := timeZero,
endTime := timeInfinite,
duration := oneTimeUnit,
requiredEngageTargetCapacity := unitCapacity}
end-mode-machine

Mission =
mode-machine Mission is
external-variable sensorId : NodeId
external-variable trackerId : NodeId
external-variable shooterId : NodeId
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
input-variable targetLocation : Location
internal-variable availableEngageTargetCapacity : Capacity
internal-variable requiredProvideReconCapacity : Capacity
internal-variable requiredProvideTrackCapacity : Capacity
internal-variable requiredProvideShootCapacity : Capacity
internal-variable requiredWeaponSupportCapacity : Capacity
internal-variable requiredProvideShooterSelectionCapacity: Capacity
initial final mode Idle has
end-mode
mode FlyingMission has
provided-invariant engageTarget(startTime, endTime, duration, targetLocation)
mode-machine MissionModes is
initial mode Nav has
required-invariant provideRecon(startTime, endTime, duration, targetLocation, sensorId)
required-invariant provideTrack(startTime, endTime, duration, sensorId, trackerId)
required-invariant provideShooterSelection(startTime, endTime, trackerId, shooterId)
end-mode
mode Ingress has
end-mode
mode Release has
end-mode
mode Flyout has
end-mode
final mode BDA has
end-mode
transition from Nav to Ingress when {} is {}
transition from Ingress to Release when {} is {}
transition from Release to Flyout when {} is {}
transition from Flyout to BDA when {} is {}
```

```
end-mode-machine
end-mode
transition from Idle to FlyingMission when {} is {}
transition from FlyingMission to Idle when {} is {}
transition from FlyingMission to FlyingMission when {} is {}
transition from Initialization to Idle when {} is {
targetLocation := zeroLocation,
sensorId := noNodeId,
trackerId := noNodeId,
shooterId := noNodeId,
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
availableEngageTargetCapacity := unitCapacity,
requiredProvideReconCapacity := unitCapacity,
requiredProvideTrackCapacity := unitCapacity,
requiredProvideShooterSelectionCapacity := unitCapacity,
requiredProvideShootCapacity := unitCapacity,
requiredWeaponSupportCapacity := unitCapacity}
end-mode-machine

MissionRecon =
mode-machine MissionRecon is
external-variable sensorId : NodeId
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
input-variable targetLocation: Location
internal-variable availableProvideReconCapacity : Capacity
internal-variable requiredProvideSensorCapacity : Capacity
internal-variable requiredProvideAirframeCapacity: Capacity
internal-variable requiredProvidePositionedAirframeCapacity: Capacity
initial mode Idle has
end-mode
mode Recon has
provided-invariant
provideRecon(startTime, endTime, duration, targetLocation, sensorId)
required-invariant
provideSensor(startTime, endTime, duration, sensorId)
required-invariant
providePositionedAirframe(startTime, endTime, duration, targetLocation, sensorId)
end-mode
transition from Idle to Recon when {} is {}
transition from Recon to Idle when {} is {}
transition from Recon to Recon when {} is {}
transition from Initialization to Idle when {} is {
targetLocation := zeroLocation,
sensorId := noNodeId,
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
availableProvideReconCapacity := unitCapacity,
requiredProvideSensorCapacity := unitCapacity,
requiredProvideAirframeCapacity := unitCapacity,
requiredProvidePositionedAirframeCapacity := unitCapacity}
end-mode-machine

MissionTrack =
mode-machine MissionTrack is
external-variable trackerId : NodeId
external-variable startTime : Time
external-variable endTime : Time
```

```
external-variable duration : Duration
input-variable sensorId : NodeId
internal-variable availableProvideTrackCapacity: Capacity
internal-variable requiredProvideATRCapacity : Capacity
initial mode Idle has
end-mode
mode Track has
provided-invariant provideTrack(startTime, endTime, duration, sensorId, trackerId)
required-invariant provideATR(startTime, endTime, duration, trackerId)
end-mode
transition from Idle to Track when {} is {}
transition from Track to Idle when {} is {}
transition from Track to Track when {} is {}
transition from Initialization to Idle when {} is {
trackerId := noNodeId,
sensorId := noNodeId,
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
availableProvideTrackCapacity := unitCapacity,
requiredProvideATRCapacity := unitCapacity}
end-mode-machine

MissionShooterSelection =
mode-machine MissionShooterSelection is
external-variable sensorId : NodeId
external-variable trackerId : NodeId
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
internal-variable availableProvideShooterSelectionCapacity: Capacity
initial mode Idle has
end-mode
mode ShooterSelection has
provided-invariant provideShooterSelection(startTime, endTime, sensorId, trackerId)
end-mode
transition from Idle to ShooterSelection when {} is {}
transition from ShooterSelection to Idle when {} is {}
transition from ShooterSelection to ShooterSelection when {} is {}
transition from Initialization to Idle when {} is {
sensorId := noNodeId,
trackerId := noNodeId,
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
availableProvideShooterSelectionCapacity := unitCapacity}
end-mode-machine

AVSensor =
mode-machine AVSensor is
constant homeId : NodeId
constant requiredCPU : Capacity
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
input-variable inHomeId : NodeId
internal-variable availableProvideSensorCapacity : Capacity
internal-variable requiredProvideCPUCapacity : Capacity
internal-variable requiredProvideCPU1Capacity : Capacity
initial mode Idle has
end-mode
mode Sensing has
```

```
provided-invariant provideSensor(startTime, endTime, duration, inHomeId)
required-invariant provideCPU1(homeId, requiredCPU)
constraint (inHomeId = homeId) or (inHomeId = noNodeId)
end-mode
transition from Idle to Sensing when {} is {
inHomeId := homeId
}
transition from Sensing to Idle when {} is {}
transition from Sensing to Sensing when {} is {
inHomeId := homeId
}
transition from Initialization to Idle when {} is {
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
inHomeId := homeId,
availableProvideSensorCapacity := unitCapacity,
requiredProvideCPUCapacity := unitCapacity,
requiredProvideCPU1Capacity := unitCapacity}
end-mode-machine

AVAirframe =
mode-machine AVAirframe is
constant homeId : NodeId
constant baseLocation : Location
constant maxFuelCapacity : Capacity
constant maxSpeed : Integer
constant fuelDistanceBurnRate : CapacityPerDistance
constant fuelTimeBurnRate : CapacityPerTime
input-variable inHomeId : NodeId
input-variable targetLocation : Location
input-variable engagementDuration : Duration
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
internal-variable origin : Location
internal-variable destination : Location
internal-variable fuelLevel : Capacity
internal-variable availableProvidePositionedAirframeCapacity: Capacity
internal-variable requiredProvideGPSCapacity : Capacity
initial final mode Idle has
end-mode
mode Positioning has
required-invariant provideGPS(startTime, endTime, duration, inHomeId)
end-mode
rest mode Positioned has
provided-invariant
providePositionedAirframe(startTime, endTime, engagementDuration, targetLocation, inHomeId)
constraint (inHomeId = homeId) or (inHomeId = noNodeId)
end-mode
transition from Idle to Positioning
when {~(targetLocation = baseLocation) &
(fuelLevel >=
consumedFuelForDistance(baseLocation, targetLocation, fuelDistanceBurnRate))}
is {
origin := baseLocation,
destination := targetLocation,
fuelLevel := fuelLevel -
consumedFuelForDistance(baseLocation, targetLocation, fuelDistanceBurnRate),
duration := computeFlightDuration(baseLocation, targetLocation, maxSpeed)}
transition from Positioning to Positioned when {
(destination = targetLocation) &
```

```
(fuelLevel >= consumedFuelForTime(fuelTimeBurnRate, engagementDuration))}
is {
inHomeId := homeId,
origin := destination,
destination := targetLocation,
fuelLevel := fuelLevel - consumedFuelForTime(fuelTimeBurnRate, engagementDuration)}
transition from Positioned to Positioning when {((destination = targetLocation) &
(fuelLevel >= consumedFuelForDistance(baseLocation, targetLocation, fuelDistanceBurnRate))) or
(~(destination = targetLocation) &
 (fuelLevel >= consumedFuelForDistance(destination, targetLocation, fuelDistanceBurnRate)))}
is {
origin := destination,
destination := if (destination = targetLocation)
then baseLocation
else targetLocation,
fuelLevel := fuelLevel -
consumedFuelForDistance(origin, destination, fuelDistanceBurnRate),
duration := computeFlightDuration(origin, destination, maxSpeed)}
transition from Positioning to Idle when {destination = baseLocation} is {
origin := baseLocation,
destination := baseLocation,
fuelLevel := maxFuelCapacity,
duration := oneTimeUnit}
transition from Positioned to Positioned when {(destination = targetLocation) &
(fuelLevel >= (fuelLevel - consumedFuelForTime(fuelTimeBurnRate, engagementDuration)))} is {
inHomeId := homeId,
fuelLevel := fuelLevel - consumedFuelForTime(fuelTimeBurnRate, engagementDuration),
duration := engagementDuration}
transition from Initialization to Idle when {} is {
inHomeId := homeId,
targetLocation := baseLocation,
engagementDuration := oneTimeUnit,
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
origin := baseLocation,
destination := baseLocation,
fuelLevel := maxFuelCapacity,
availableProvidePositionedAirframeCapacity := unitCapacity,
requiredProvideGPSCapacity := unitCapacity}
end-mode-machine

ATR =
mode-machine ATR is
constant homeId : NodeId
constant requiredCPU : Capacity
external-variable startTime: Time
external-variable endTime : Time
external-variable duration : Duration
input-variable inHomeId : NodeId
internal-variable availableProvideATRCapacity: Capacity
internal-variable requiredProvideCPU1Capacity: Capacity
initial mode Idle has
end-mode
mode Tracking has
provided-invariant provideATR(startTime, endTime, duration, inHomeId)
required-invariant provideCPU1(homeId, requiredCPU)
constraint (inHomeId = homeId) or (inHomeId = noNodeId)
end-mode
transition from Idle to Tracking when {} is {
inHomeId := homeId
}
```

```
transition from Tracking to Idle when {} is {}
transition from Tracking to Tracking when {} is {
inHomeId := homeId
}
transition from Initialization to Idle when {} is {
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
inHomeId := homeId,
availableProvideATRCapacity := unitCapacity,
requiredProvideCPU1Capacity := unitCapacity}
end-mode-machine

GPS =
mode-machine GPS is
constant homeId : NodeId
constant requiredCPU : Capacity
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
input-variable inHomeId : NodeId
internal-variable availableProvideGPSCapacity : Capacity
internal-variable requiredProvideCPU1Capacity : Capacity
initial mode Idle has
end-mode
mode Positioning has
provided-invariant provideGPS(startTime, endTime, duration, inHomeId)
required-invariant provideCPU1(homeId, requiredCPU)
constraint (inHomeId = homeId) or (inHomeId = noNodeId)
end-mode
transition from Idle to Positioning when {} is {
inHomeId := homeId
}
transition from Positioning to Idle when {} is {}
transition from Positioning to Positioning when {} is {
inHomeId := homeId
}
transition from Initialization to Idle when {} is {
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
inHomeId := homeId,
availableProvideGPSCapacity := unitCapacity,
requiredProvideCPU1Capacity := unitCapacity}
end-mode-machine

CPU =
mode-machine CPU is
constant homeId : NodeId
constant nodeCPUCapacity : Capacity
input-variable processingHomeId : NodeId
input-variable requiredCPU : Capacity
external-variable startTime : Time
external-variable endTime : Time
external-variable duration : Duration
internal-variable availableProvideCPU1Capacity : Capacity
internal-variable availableProvideCPU2Capacity : Capacity
initial final mode Idle has
end-mode
mode Busy has
provided-invariant provideCPU1(processingHomeId, requiredCPU)
provided-invariant provideCPU2(processingHomeId, requiredCPU)
```

```
constraint processingHomeId = homeId
end-mode
transition from Idle to Busy when {} is {
duration := computeProcessingDuration(requiredCPU, nodeCPUCapacity)}
transition from Busy to Idle when {serviceProvided?} is {}
transition from Busy to Busy when {~serviceProvided?} is {
duration := computeProcessingDuration(requiredCPU, nodeCPUCapacity)}
transition from Initialization to Idle when {} is {
processingHomeId := homeId,
requiredCPU := unitCapacity,
startTime := timeZero,
endTime := timeInfinite,
duration := infiniteDuration,
availableProvideCPU1Capacity := unitCapacity,
availableProvideCPU2Capacity := unitCapacity}
```
end-mode-machine

Appendix B. ACE/TAO Defect Analysis

Illuma℠ Defect Data *for*

ACE/TAO

from Kestrel Institute

12/13/2002

Illuma SERVICES PROVIDED BY

2440 West El Camino Real
Mountain View, CA 94040
1-650-316-4400 · www.reasoning.com

B.1. INTRODUCTION

Illuma[SM]

Illuma is an automated software inspection service developed by Reasoning that rapidly detects critical structural defects in software. It is an important complement to functional testing, because it detects defects before testing and provides metrics that assist in risk-assessment.

Deliverables

Results from an Illuma service inspection are provided in two reports. The Illuma Defect Metrics report provides high-level metrics results, and the Illuma Defect Data report includes detail for each individual defect. This document is the Illuma Defect Data report.

B.2. SUMMARY DEFECT REPORT

B.2.1. Inventory Summary

Reasoning inspected a complete application.

Total Number of Source Files:	268
Number of User Include Files:	468
Total Number of User Files Processed:	**736**
Total LOC of Source Files:	79,876
Number LOC User Include Files:	60,816
Total LOC in Project:	**140,692**

B.2.2. Defect Summary

The column Defect Instances in the table below details, per defect class, how many defects there are in the application.

The column Files Affected details, per defect class, the number of files in the application that have one or more defects.

Inspection Class	Defect Instances	Files Affected
Memory Leak Reference to allocated memory is lost	3	3
NULL Pointer Dereference Expression dereferences a NULL pointer	3	3
Bad Deallocation Deallocation is inappropriate for type of data	0	0
Out of Bounds Array Access Expression accesses a value beyond the array	0	0
Uninitialized Variable Variable is not initialized prior to use	2	1
Total Defect Instances	**8**	**---**

B.3. DETAILED DEFECT REPORT

DEFECT CLASS: Memory Leak Defect Number: 1

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\Configuration.cpp : 1059

DESCRIPTION:

Local variable temp_path, declared on line 1012, is assigned a pointer to a block of memory allocated by new[] on line 1013(176(OS_Memory.h)). No other pointer refers to this memory block, so it is inaccessible (still allocated, but unreachable) once temp_path goes out of scope after line 1059. A similar error can be found on line 1067.

PRECONDITIONS:

The conditional expression (::RegOpenKey (hKey, 0, &result) != ERROR_SUCCESS) on line 1007 evaluates to false AND

The conditional expression (POINTER == 0) on line 1013(177(OS_Memory.h)) evaluates to false AND

The for loop on line 1022 is executed with temp != 0 evaluates to true AND

The conditional expression (ACE_TEXT_RegOpenKey (result, temp, &subkey) != ERROR_SUCCESS) on line 1036 evaluates to true AND

The conditional expression (!create || ACE_TEXT_RegCreateKeyEx (result, temp, 0, 0, 0, KEY_ALL_ACCESS, 0, &subkey, #if defined (__MINGW32__) (PDWORD) 0 #else 0 #endif) != ERROR_SUCCESS) on line 1042 evaluates to true.

IMPACT:

Depending on how long the application runs, how frequently the leak occurs, and the amount of available (virtual) memory, memory leaks will sooner or later cause performance degradation of the application, and potentially of the entire system. Eventually, the performance degradation may lead to a fatal out-of-memory condition. This condition may be encountered by an application unrelated to the one that caused the memory leak.

CODE FRAGMENT:

```
997 HKEY
998 ACE_Configuration_Win32Registry::resolve_key (HKEY hKey,
999 const ACE_TCHAR* path,
1000 int create)
1001 {
1002 HKEY result = 0;
1003 // Make a copy of hKey
1004 #if defined (ACE_HAS_WINCE)
1005 if (::RegOpenKeyEx (hKey, 0, 0, 0, &result) != ERROR_SUCCESS)
1006 #else
1007 if (::RegOpenKey (hKey, 0, &result) != ERROR_SUCCESS)
1008 #endif // ACE_HAS_WINCE
1009 return 0;
1010
1011 // recurse through the path
1012 ACE_TCHAR *temp_path = 0;
1013 ACE_NEW_RETURN (temp_path,
1014 ACE_TCHAR[ACE_OS::strlen (path) + 1],
1015 0);
1016 ACE_Auto_Basic_Array_Ptr<ACE_TCHAR> pData (temp_path);
1017 ACE_OS::strcpy (pData.get (), path);
1018 ACE_Tokenizer parser (pData.get ());
1019 parser.delimiter_replace ('\\', '\0');
1020 parser.delimiter_replace ('/', '\0');
1021
1022 for (ACE_TCHAR *temp = parser.next ();
1023 temp != 0;
1024 temp = parser.next ())
1025 {
1026 // Open the key
1027 HKEY subkey;
1028
1029 #if defined (ACE_HAS_WINCE)
1030 if (ACE_TEXT_RegOpenKeyEx (result,
1031 temp,
1032 0,
1033 0,
1034 &subkey) != ERROR_SUCCESS)
1035 #else
1036 if (ACE_TEXT_RegOpenKey (result,
1037 temp,
1038 &subkey) != ERROR_SUCCESS)
1039 #endif // ACE_HAS_WINCE
1040 {
1041 // try creating it
1042 if (!create || ACE_TEXT_RegCreateKeyEx (result,
1043 temp,
1044 0,
1045 0,
1046 0,
1047 KEY_ALL_ACCESS,
1048 0,
1049 &subkey,
```

```
1050 #if defined (__MINGW32__)
1051 (PDWORD) 0
1052 #else
1053 0
1054 #endif /* __MINGW32__ */
1055 ) != ERROR_SUCCESS)
1056 {
1057 // error
1058 ::RegCloseKey (result);
1059 return 0;
1060 }
1061 }
1062 // release our open key handle
1063 ::RegCloseKey (result);
1064 result = subkey;
1065 }
1066
1067 return result;
1068 }
```

DEFECT CLASS: Memory Leak Defect Number: 2

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\Log_Msg.cpp : 334

DESCRIPTION:

Local variable tss_log_msg, declared on line 304, is assigned a pointer to a block of memory allocated by new on line 324(176(OS_Memory.h)). No other pointer refers to this memory block, so it is inaccessible (still allocated, but unreachable) once tss_log_msg goes out of scope after line 334.

PRECONDITIONS:

The conditional expression (key_created_ == 0) on line 256 evaluates to false AND

The conditional expression (ACE_Thread::getspecific (log_msg_tss_key_, ACE_reinterpret_cast (void **, &tss_log_msg)) == -1) on line 307 evaluates to false AND

The conditional expression (tss_log_msg == 0) on line 313 evaluates to true AND

The conditional expression (POINTER == 0) on line 324(177(OS_Memory.h)) evaluates to false AND

The conditional expression (ACE_Thread::setspecific (log_msg_tss_key_, ACE_reinterpret_cast (void *, tss_log_msg)) != 0) on line 331 evaluates to true.

IMPACT:

Depending on how long the application runs, how frequently the leak occurs, and the amount of available (virtual) memory, memory leaks will sooner or later cause performance degradation of the application, and potentially of the entire system. Eventually, the performance degradation may lead to a fatal out-of-memory condition. This condition may be encountered by an application unrelated to the one that caused the memory leak.

CODE FRAGMENT:

```
248 ACE_Log_Msg *
249 ACE_Log_Msg::instance (void)
250 {
251 #if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
252 # if defined (ACE_HAS_THREAD_SPECIFIC_STORAGE) || \
253 defined (ACE_HAS_TSS_EMULATION)
```

```
254 // TSS Singleton implementation.
255
256 if (key_created_ == 0)
257 {
258 ACE_thread_mutex_t *lock =
259 ACE_reinterpret_cast (ACE_thread_mutex_t *,
260 ACE_OS_Object_Manager::preallocated_object
261 [ACE_OS_Object_Manager::ACE_LOG_MSG_INSTANCE_LOCK]);
262
263 if (1 == ACE_OS_Object_Manager::starting_up())
...
267 ;
268 else
269 ACE_OS::thread_mutex_lock (lock);
270
271 if (key_created_ == 0)
272 {
273 // Allocate the Singleton lock.
274 ACE_Log_Msg_Manager::get_lock ();
275
276 {
277 ACE_NO_HEAP_CHECK;
278 if (ACE_Thread::keycreate (&log_msg_tss_key_,
279 &ACE_TSS_cleanup) != 0)
280 {
281 if (1 == ACE_OS_Object_Manager::starting_up())
...
285 ;
286 else
287 ACE_OS::thread_mutex_unlock (lock);
288 return 0; // Major problems, this should *never* happen!
289 }
290 }
291
292 key_created_ = 1;
293 }
294
295 if (1 == ACE_OS_Object_Manager::starting_up())
...
299 ;
300 else
301 ACE_OS::thread_mutex_unlock (lock);
302 }
303
304 ACE_Log_Msg *tss_log_msg = 0;
305
306 // Get the tss_log_msg from thread-specific storage.
307 if (ACE_Thread::getspecific (log_msg_tss_key_,
308 ACE_reinterpret_cast (void **,
309 &tss_log_msg)) == -1)
310 return 0; // This should not happen!
311
312 // Check to see if this is the first time in for this thread.
```

```
313 if (tss_log_msg == 0)
314 {
...
321 {
322 ACE_NO_HEAP_CHECK;
323
324 ACE_NEW_RETURN (tss_log_msg,
325 ACE_Log_Msg,
326 0);
...
331 if (ACE_Thread::setspecific (log_msg_tss_key_,
332 ACE_reinterpret_cast (void *,
333 tss_log_msg)) != 0)
334 return 0; // Major problems, this should *never* happen!
335 }
336 }
337
338 return tss_log_msg;
```

DEFECT CLASS: Memory Leak Defect Number: 3

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\TP_Reactor.cpp : 275

DESCRIPTION:

Local variable aeh, declared on line 258, is assigned a pointer to a block of memory allocated by new[] on line 261(176(OS_Memory.h)). No other pointer refers to this memory block, so it is inaccessible (still allocated, but unreachable) once aeh goes out of scope after line 275.

PRECONDITIONS:

The conditional expression (POINTER == 0) on line 261(177(OS_Memory.h)) evaluates to false AND

The conditional expression (!guard.is_owner ()) on line 274 evaluates to true.

IMPACT:

Depending on how long the application runs, how frequently the leak occurs, and the amount of available (virtual) memory, memory leaks will sooner or later cause performance degradation of the application, and potentially of the entire system. Eventually, the performance degradation may lead to a fatal out-of-memory condition. This condition may be encountered by an application unrelated to the one that caused the memory leak.

CODE FRAGMENT:

```
253 int
254 ACE_TP_Reactor::remove_handler (const ACE_Handle_Set &handles,
255 ACE_Reactor_Mask m)
256 {
257 // Array of <Event_Handlers> corresponding to <handles>
258 ACE_Event_Handler **aeh = 0;
259
260 // Allocate memory for the size of the handle set
261 ACE_NEW_RETURN (aeh,
262 ACE_Event_Handler *[handles.num_set ()],
263 -1);
264
265 size_t index = 0;
266
267 // Artificial scoping for grabbing and releasing the token
268 {
269 ACE_TP_Token_Guard guard (this->token_);
```

```
270
271 // Acquire the token
272 int result = guard.acquire_token ();
273
274 if (!guard.is_owner ())
275 return result;
276
277 ACE_HANDLE h;
278
279 ACE_Handle_Set_Iterator handle_iter (handles);
280
281 while ((h = handle_iter ()) != ACE_INVALID_HANDLE)
282 {
283 size_t slot = 0;
284 ACE_Event_Handler *eh =
285 this->handler_rep_.find (h, &slot);
```

DEFECT CLASS: Null Pointer Dereference Defect Number: 4

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\MEM_Acceptor.cpp : 104

DESCRIPTION:

The local pointer variable len_ptr, declared on line 82, and assigned on line 82, may be NULL where it is dereferenced on line 104.

PRECONDITIONS:

The conditional expression (this->shared_accept_start (timeout, restart, in_blocking_mode) == -1) on line 86 evaluates to false AND

The conditional expression (remote_sap != 0) on line 101 evaluates to true.

IMPACT:

A NULL pointer dereference usually causes a program exception. Notable exceptions to this rule are some embedded environments, in which a NULL pointer dereference does not cause program termination.

CODE FRAGMENT:

```
73 int
74 ACE_MEM_Acceptor::accept (ACE_MEM_Stream &new_stream,
75 ACE_MEM_Addr *remote_sap,
76 ACE_Time_Value *timeout,
77 int restart,
78 int reset_new_handle)
79 {
80 ACE_TRACE ("ACE_MEM_Acceptor::accept");
81
82 int *len_ptr = 0;
83 sockaddr *addr = 0;
84
85 int in_blocking_mode = 1;
86 if (this->shared_accept_start (timeout,
87 restart,
88 in_blocking_mode) == -1)
89 return -1;
90 else
91 {
92 do
93 new_stream.set_handle (ACE_OS::accept (this->get_handle (),
94 addr,
95 len_ptr));
```

```
96 while (new_stream.get_handle () == ACE_INVALID_HANDLE
97 && restart != 0
98 && errno == EINTR
99 && timeout == 0);
100
101 if (remote_sap != 0)
102 {
103 ACE_INET_Addr temp (ACE_reinterpret_cast (sockaddr_in *, addr),
104 *len_ptr);
105 remote_sap->set_port_number(temp.get_port_number ());
106 }
107 }
108
109 if (this->shared_accept_finish (new_stream,
110 in_blocking_mode,
111 reset_new_handle) == -1)
112 return -1;
113
114 // Allocate 2 * MAXPATHLEN so we can accomodate the unique
```

DEFECT CLASS: Null Pointer Dereference Defect Number: 5

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\SString.cpp : 394

DESCRIPTION:

The local pointer variable ACE_SString::rep_, declared on line 231(SString.h) , and assigned on line 393, may be NULL where it is dereferenced on line 394. This NULL pointer dereference only happens in an Out Of Memory context. Similar errors can be found on lines 492 and 522.

PRECONDITIONS:

The function malloc, called on line 393, returns NULL.

IMPACT:

A NULL pointer dereference usually causes a program exception. Notable exceptions to this rule are some embedded environments, in which a NULL pointer dereference does not cause program termination.

CODE FRAGMENT:

```
381 ACE_SString::ACE_SString (ACE_Allocator *alloc)
382 : allocator_ (alloc),
383 len_ (0),
384 rep_ (0)
385
386 {
387 ACE_TRACE ("ACE_SString::ACE_SString");
388
389 if (this->allocator_ == 0)
390 this->allocator_ = ACE_Allocator::instance ();
391
392 this->len_ = 0;
393 this->rep_ = (char *) this->allocator_->malloc (this->len_ + 1);
394 this->rep_[this->len_] = '\0';
395 }
```

DEFECT CLASS: Null Pointer Dereference Defect Number: 6

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\Svc_Conf_y.cpp : 1535

DESCRIPTION:

The local pointer variable mt, declared on line 1521, and assigned on line 1521, may be NULL where it is dereferenced on line 1535.

PRECONDITIONS:

The conditional expression (sr == 0 || st == 0 || mt == 0) on line 1524 evaluates to true.

IMPACT:

A NULL pointer dereference usually causes a program exception. Notable exceptions to this rule are some embedded environments, in which a NULL pointer dereference does not cause program termination.

CODE FRAGMENT:

```
1511 static ACE_Module_Type *
1512 ace_get_module (ACE_Static_Node *str_rec,
1513 ACE_Static_Node *svc_type,
1514 int & ace_yyerrno)
1515 {
1516 const ACE_Service_Type *sr = str_rec->record ();
1517 const ACE_Service_Type_Impl *type = sr->type ();
1518 ACE_Stream_Type *st = sr == 0 ? 0 : (ACE_Stream_Type *) type;
1519 const ACE_Service_Type *sv = svc_type->record ();
1520 type = sv->type ();
1521 ACE_Module_Type *mt = (ACE_Module_Type *) type;
1522 const ACE_TCHAR *module_type_name = svc_type->name ();
1523
1524 if (sr == 0 || st == 0 || mt == 0)
1525 {
1526 ACE_ERROR ((LM_ERROR,
1527 ACE_LIB_TEXT ("cannot locate Module_Type %s or STREAM_Type %s\n"),
1528 module_type_name,
1529 str_rec->name ()));
1530 ace_yyerrno++;
1531 }
1532
1533 // Make sure that the Module has the same name as the
1534 // Module_Type object from the svc.conf file.
1535 ACE_Module<ACE_SYNCH> *mp = (ACE_Module<ACE_SYNCH> *) mt->object ();
1536
1537 if (ACE_OS::strcmp (mp->name (), module_type_name) != 0)
1538 {
1539 ACE_DEBUG ((LM_DEBUG,
```

```
1540 ACE_LIB_TEXT ("warning: assigning Module_Type name %s to Module %s since
       names differ\n"),
1541 module_type_name,
1542 mp->name ()));
1543 mp->name (module_type_name);
1544 }
```

DEFECT CLASS: Uninitialized Variable Defect Number: 7

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\Configuration.cpp : 362

DESCRIPTION:

The local variable rhsInt, declared on line 347, is used on line 362, before rhsInt has been initialized.

PRECONDITIONS:

The conditional expression (nonconst_rhs.find_value (rhsSection, valueName.c_str (), rhsType) != 0) on line 310 evaluates to false AND

The conditional expression (valueType != rhsType) on line 318 evaluates to false AND

The conditional expression (valueType == STRING) on line 326 evaluates to false AND

The conditional expression (valueType == INTEGER) on line 345 evaluates to true AND

The conditional expression (nonconst_this->get_integer_value (thisSection, valueName.c_str (), thisInt) != 0) on line 348 evaluates to true.

IMPACT:

Usage of uninitialized variables can cause unpredictable results in the program (because the value of the variable is essentially random), and in the worst case, a program exception.

CODE FRAGMENT

```
260 int ACE_Configuration::operator== (const ACE_Configuration& rhs) const
261 {
...
309 // look for the same value in the rhs section
310 if (nonconst_rhs.find_value (rhsSection,
311 valueName.c_str (),
312 rhsType) != 0)
313 {
314 // We're not equal if the same value cannot
315 // be found in the rhs object.
316 rc = 0;
317 }
318 else if (valueType != rhsType)
319 {
320 // we're not equal if the types do not match.
321 rc = 0;
322 }
```

```
323 else
324 {
325 // finally compare values.
326 if (valueType == STRING)
327 {
328 ACE_TString thisString, rhsString;
329 if (nonconst_this->get_string_value (thisSection,
330 valueName.c_str (),
331 thisString) != 0)
332 {
333 // we're not equal if we cannot get this string
334 rc = 0;
335 }
336 else if (nonconst_rhs.get_string_value (rhsSection,
337 valueName.c_str (),
338 rhsString) != 0)
339 {
340 // we're not equal if we cannot get rhs string
341 rc = 0;
342 }
343 rc = thisString == rhsString;
344 }
345 else if (valueType == INTEGER)
346 {
347 u_int thisInt, rhsInt;
348 if (nonconst_this->get_integer_value (thisSection,
349 valueName.c_str (),
350 thisInt) != 0)
351 {
352 // we're not equal if we cannot get this int
353 rc = 0;
354 }
355 else if (nonconst_rhs.get_integer_value (rhsSection,
356 valueName.c_str (),
357 rhsInt) != 0)
358 {
359 // we're not equal if we cannot get rhs int
360 rc = 0;
361 }
362 rc = thisInt == rhsInt;
363 }
364 else if (valueType == BINARY)
365 {
366 void* thisData = 0;
367 void* rhsData = 0;
368 size_t thisLength, rhsLength;
369 if (nonconst_this->get_binary_value (thisSection,
370 valueName.c_str (),
371 thisData,
372 thisLength) != 0)
```

DEFECT CLASS: Uninitialized Variable Defect Number: 8

LOCATION:

PCES\ACE_only\ACE_wrappers\ace\Configuration.cpp : 386

DESCRIPTION:

The local variable rhsLength, declared on line 368, is used on line 386, before rhsLength has been initialized.

PRECONDITIONS:

The conditional expression (valueType != rhsType) on line 318 evaluates to false AND

The conditional expression (valueType == STRING) on line 326 evaluates to false AND

The conditional expression (valueType == INTEGER) on line 345 evaluates to false AND

The conditional expression (valueType == BINARY) on line 364 evaluates to true AND

The conditional expression (nonconst_this->get_binary_value (thisSection, valueName.c_str (), thisData, thisLength) != 0) on line 369 evaluates to true.

IMPACT:

Usage of uninitialized variables can cause unpredictable results in the program (because the value of the variable is essentially random), and in the worst case, a program exception.

CODE FRAGMENT:

```
260 int ACE_Configuration::operator== (const ACE_Configuration& rhs) const
261 {
...
318 else if (valueType != rhsType)
319 {
320 // we're not equal if the types do not match.
321 rc = 0;
322 }
323 else
324 {
325 // finally compare values.
326 if (valueType == STRING)
327 {
328 ACE_TString thisString, rhsString;
329 if (nonconst_this->get_string_value (thisSection,
330 valueName.c_str (),
331 thisString) != 0)
332 {
333 // we're not equal if we cannot get this string
```

```
334 rc = 0;
335 }
336 else if (nonconst_rhs.get_string_value (rhsSection,
337 valueName.c_str (),
338 rhsString) != 0)
339 {
340 // we're not equal if we cannot get rhs string
341 rc = 0;
342 }
343 rc = thisString == rhsString;
344 }
345 else if (valueType == INTEGER)
346 {
347 u_int thisInt, rhsInt;
348 if (nonconst_this->get_integer_value (thisSection,
349 valueName.c_str (),
350 thisInt) != 0)
351 {
352 // we're not equal if we cannot get this int
353 rc = 0;
354}
355 else if (nonconst_rhs.get_integer_value (rhsSection,
356 valueName.c_str (),
357 rhsInt) != 0)
358 {
359 // we're not equal if we cannot get rhs int
360 rc = 0;
361 }
362 rc = thisInt == rhsInt;
363 }
364 else if (valueType == BINARY)
365 {
366 void* thisData = 0;
367 void* rhsData = 0;
368 size_t thisLength, rhsLength;
369 if (nonconst_this->get_binary_value (thisSection,
370 valueName.c_str (),
371 thisData,
372 thisLength) != 0)
373 {
374 // we're not equal if we cannot get this data
375 rc = 0;
376 }
377 else if (nonconst_rhs.get_binary_value (rhsSection,
378 valueName.c_str (),
379 rhsData,
380 rhsLength) != 0)
381 {
382 // we're not equal if we cannot get this data
383 rc = 0;
384 }
385
386 rc = thisLength == rhsLength;
```

```
387 // are the length's the same?
388
389 if (rc)
390 {
391 unsigned char* thisCharData = (unsigned char*)thisData;
392 unsigned char* rhsCharData = (unsigned char*)rhsData;
393 // yes, then check each element
394 for (size_t count = 0;
395 (rc) && (count < thisLength);
396 count++)
```

B.4. UNDERSTANDING DEFECT CLASS DESCRIPTIONS

This Appendix provides a detailed explanation of each of the Illuma defect classes for C/C++. These examples assist the client in evaluating the impact of each of the defects listed in the Detailed Defect Report.

For each defect class, this document provides:

1. A short description of the class;

2. The likely impact of the defect;

3. Advice on how the defect can be repaired;

4. An example code fragment that explains the defect in detail. Note that the example code fragments are not from the inspected application.

In general, data corruption is considered the worst impact. Data corruption can go unnoticed for weeks while damage continues to accumulate.

When a program exception occurs, there is better evidence that something went wrong. Even in this case, the failure sometimes goes unnoticed, for example if the program is a cgi-bin program that runs frequently for a short period of time. However, in most other situations, a program exception is a fatal error that leads to immediate program termination. In an embedded system or "daemon" process, such a failure could be catastrophic for the overall system.

When unpredictable results happen, they may eventually result in data corruption and/or program exceptions, but they may also go unnoticed. If they finally cause an observable error to occur, a large effort is usually required to track down exactly where the error occurs, because the reduction process to track the error down often makes the error fail to recur. Certain defects in one application may cause other programs on the same platform, or even the operating system, to fail, often after some time has passed. These defects are particularly difficult to track down.

Finally, over 70% of the effort spent on most C/C++ applications is spent in maintenance. Even when not directly preventing failures, the removal of defects may greatly reduce the cost of:

- Extending the system with new functionality, such as web-enabling it;

- Finding the root cause of a failure;

- Training new staff and consultants; and

- Rewriting or replacing the system.

In the descriptions of the defect classes, as much context as reasonably possible is provided. It is beyond the scope of this document, however, to explain everything that is needed to understand fully how to program in C/C++. The following publications may help to gain a deeper understanding:

- Kernighan & Ritchie, The C Programming Language, 2nd Edition, Bell Telephone Laboratories, Inc., 1988.

- Bjarne Stroustrup, The C++ Programming Language, 3rd Edition, AT&T, 1997.

- Steve McConnell, Code Complete, Microsoft Press, 1993.

- Andrew Koenig, C Traps and Pitfalls, AT&T Bell Telephone Labs, 1989.

- P.J. Plauger, The Standard C library, Prentice Hall, 1992.

- Scott Meyers, Effective C++, 2nd Edition, Addison Wesley, 1998.

B.4.1. Memory Leak

DESCRIPTION

Memory leak refers to the loss of available memory space that occurs when dynamic data (memory allocated on the heap by calling any of the standard C library routines malloc(), calloc(), realloc(), strdup() or the C++ operator new) is no longer used but never deallocated (by calling free(), realloc() or the C++ operator delete).

IMPACT

Each time the leak occurs, the application drains the available memory pool. On some systems, memory may be allocated from a global pool, in which case the loss of available memory may affect the entire system, not just the application that caused it. Even on virtual-memory systems, where each process has its own protected address space, the gradual increase in application size can result in performance degradation that affects the entire system.

Depending on how long the application runs, how frequently the leak occurs, and the amount of available (including virtual) memory, memory leaks will sooner or later cause performance degradation of the application, and potentially of the entire system.

Eventually, the performance degradation may lead to a fatal out-of-memory condition. This condition may be encountered by an application unrelated to the one that caused the memory leak.

Understanding the application is important to rating this defect. For example, if this defect occurs in a daemon, the impact is almost always high, but if it occurs in a CGI script, the impact is usually negligible.

REPAIR

A deallocation corresponding to each allocation is not always necessary, but it is a safe programming practice. For example, it is not necessary to deallocate dynamic data that is allocated in main(), since all dynamic data is freed after main() finishes, but a conservative programmer might include the deallocation anyway.

EXAMPLE

```
int check_msg() {
msg_t *msg;
int status = RET_FAIL;
...
```

```
msg = (msg_t *)calloc(1, sizeof(*msg));
if (!msg) {
errno = OUTOFMEMORY;
return status;
}
err = Readmsg(msg);
if (err == -1)
return status;
...
}
```

In this example, memory space is allocated for a new msg. One possible execution path, however, is where the Readmsg() call fails; this leads to a return without the memory allocated for msg being freed. If this function is called frequently, and Readmsg() frequently fails, this could result in a large number of memory leaks. Eventually, this can lead to a fatal out-of-memory program exception.

B.4.2. NULL Pointer Dereference

DESCRIPTION

A NULL pointer is a pointer that refers to a specific, invalid memory address. A dereference means following a pointer to the memory location it refers to and accessing the data at that location. Thus, a NULL pointer dereference refers to an attempt to access data at this invalid address.

This defect class also reports situations where a NULL pointer is used in an assignment. Even though the assignment is not a defect by itself, a subsequent dereference will cause a program exception. The defect is reported at the assignment, because this is the earliest point in the code where the problem could be identified.

IMPACT

On most general-purpose computing platforms (such as Windows or UNIX), a NULL pointer dereference usually causes a program exception. On systems without memory management hardware, such references may not be detected at all.

REPAIR

To repair these defects, an if-statement checking for NULL values should be placed around the statements that dereference the pointers. Appropriate error-recovery should also be provided for the situations where the pointers are NULL.

EXAMPLE

```
char* ptr = strrchr(tmp_eq, '-');
int chan_num = atoi(ptr+1);
```

The string function strrchr() returns a pointer to the last occurrence of '-' in the string tmp_eq, or NULL if the character is not present. The next line of code will cause a program exception if ptr is NULL, because atoi() does not accept a NULL pointer. A check should be performed before the call to atoi() to ensure that ptr is not NULL.

B.4.3. Bad Deallocation

DESCRIPTION

Bad deallocation refers to the use of an inappropriate memory release operation (standard C library routines free() or realloc(), or C++ operators delete or delete[]) for deallocating memory, or to the deallocation of memory that was never explicitly allocated.

IMPACT

Depending on the compiler and the specific operation, the impact of this defect may range from no effect at all, to unexpected results, a memory leak, memory corruption, or a program exception.

Specifically:

- The use of delete on memory allocated with new[] may be a memory leak, because only the first element of the array is released, not the entire array;
- The use of delete on memory allocated with malloc(), calloc() or realloc(), or strdup() may cause memory corruption, or a program exception;
- The use of free() or realloc()) on memory allocated with new may cause memory corruption, or a program exception;
- The use of free() or realloc() on memory that is not heap-allocated may cause memory corruption, or a program exception.

REPAIR

Memory allocated with malloc() , calloc(), realloc(), or strdup(), should be deallocated only with free() or realloc(). Similarly, memory allocated with new should be deallocated with delete, and memory allocated with new [] should be deallocated with delete []. Furthermore, free() and delete should be applied to the exact same pointer value that was returned by the corresponding allocating expression.

EXAMPLE 1

```
DS3_NOC noct3msg;
memset((void *)&noct3msg, 0, sizeof(DS3_NOC));
...
delete (&noct3msg);
```

In this example, a local variable is initialized to all 0's using the memset() function. One of the rules in C++ is that the delete operator can only be applied to a pointer value that was previously obtained from the new operator. Use of the delete operator in the Example 1 will result in a

program exception.

EXAMPLE 2

```
char* buffer = new char[100];
...
buffer++;
...
delete [] buffer;
```

In this example, a new character array called buffer is allocated, and later the base pointer to that array is incremented. When the memory is deallocated, buffer no longer points to the beginning of the allocated block. This will often lead to a program exception, or corruption of the heap.

EXAMPLE 3

```
char* p = new char[100];
...
delete p;
```

The problem in this example is the missing [] on the delete operator. Depending on the compiler, this can result in a program exception, or corruption of the heap.

EXAMPLE 4

```
void parse_message(char *msg) {
char formatstring[100];
...
free(formatstring);
return;
}
```

The character array formatstring in this example is stack-allocated. It is inappropriate to call free() on stack-allocated memory. Depending on the memory manager, this can result in a program exception, or corruption of the heap.

B.4.4. Out of Bounds Array Access

DESCRIPTION

An out-of-bounds array access refers to a defect where an array index expression is not within the upper and lower bounds of the array.

IMPACT

An out-of-bounds array access defect can cause data corruption or lead to a program exception.

REPAIR

Array indexing needs to be guarded against out-of-bounds defects.

For situations where one array is copied into another, this type of defect can be repaired by adjusting array sizes.

For situations where index expressions are used to access arrays, an if-statement check on the index expression may suffice. In case the index expression is controlled by a loop construct, the terminating value/condition should be changed to be within the size of the array, and the index variable should be checked for manipulations within the loop that can cause an out-of-bounds access.

In case the index variable controlled by a loop is used outside the loop (which is in itself questionable programming practice), one should be aware that the value of the index variable may be outside the range specified by the loop construct.

The two most common programming mistakes are (1) using the wrong inequality test on the loop conditional (generally, <= when it should have been <), and (2) using the final value of the loop index variable after the loop to index the array, when often the loop is written such that the final value is beyond the end of the array. The following two examples demonstrate each of these problems.

EXAMPLE 1

```
#define TABLE_SIZE 20
int PowerOf2[TABLE_SIZE];
void initTable() {
int j;
PowerOf2[0] = 1;
for (j = 1; j <= TABLE_SIZE; j++)
PowerOf2[j] = PowerOf2[j-1] * 2;
}
```

In this example, the problem is that the conditional expression in the for-loop terminates when j > TABLE_SIZE. During the last iteration of the loop, j == TABLE_SIZE, which means that the assignment indexes one beyond the end of the array.

EXAMPLE 2

```
#define MAX_PATH
void MungeFilePath(char dos_path[])
{
int i;
char pathName[MAX_PATH + 1]; /* room for trailing '\0' */
/* Convert MS-DOS path characters to UNIX form, add a
* trailing '/' if necessary to prepare for later
* concatenation with the file name.
*/
for (i = 0; i < MAX_PATH && dos_path[i] != '\0'; i++) {
if (dos_path[i] == '\\')
pathName[i] = '/';
else
pathName[i] = dos_path[i];
}
if (pathName[i - 1] != '/')
pathName[i++] = '/';
pathName[i] = '\0';
...
}
```

This code fragment actually has two types of array bounds violations, both after the loop. The first defect occurs when the original path is MAX_PATH characters long and the last character stored in pathName is not a '/'. In this case, the logic is to append a '/' character, followed by a NUL character. Even though the programmer made the array one larger to hold the trailing NUL character, that is insufficient when both a '/' and a NUL character must be appended.

The second bug occurs when the initial string is zero length (i.e., the first character in dos_path is a NUL character). In this case, the first loop does not execute at all and the if-statement tests pathName[i – 1]. However, in this case i is zero, and this expression accesses a memory location before the beginning of the array!

EXAMPLE 3

```
int color[50];
...
for (i=0; i<50; i+=3) {
color[i]= colorSet(i, ...);
color[i+1]= colorSet(i, ...);
```

```
color[i+2]= colorSet(i, ...);
}
```

An array declared as a[n] has n elements, indexed from 0 to n-1. In this example, i is incremented by 3 each time through the for-loop. During the last iteration through the loop, its value is 48. When i=48, color[i+2] accesses the out-of-bounds element color[50].

B.4.5. Uninitialized Variable

DESCRIPTION

Uninitialized variable refers to a defect where local and dynamic variables are not explicitly initialized prior to use. Note that the ANSI standard requires that global and static variables are initialized (ints to 0, floats to 0.0 and pointers to NULL); therefore this defect is not reported for variables with either of these storage classes.

IMPACT

Usage of uninitialized variables can cause unpredictable results in the program (because the value of the variable is essentially random), and in the worst case, a program exception.

REPAIR

To repair this type of defect, the variable must be initialized to an appropriate value.

EXAMPLE

```
int fun() {
int len;
...
if ((to - from) > 86400) {
now = localtime(&from);
len = strftime(&OdateToStr, 64, "%m/%d/%Y - ", now);
now = localtime(&to);
len = strftime(&OdateToStr[len], 64, "%m/%d/%Y", now);
} else {
now = localtime(&from);
len = strftime(&OdateToStr[len], 64, "%m/%d/%Y", now);
}
```

In the example above, the stack-allocated variable len is uninitialized when used in the else clause of the conditional. The then clause of the conditional is fine, but if the else path is taken, then the len used in the array access is an uninitialized variable. This may result in a program exception if the random value in len causes an out-of-bounds array read.

C. ACE/TAO Defect Metrics

Illuma℠ Defect Metrics for

ACE/TAO

from Kestrel Institute

12/13/2002

Illuma Services Provided By

REASONING

2440 West El Camino Real
Mountain View, CA 94040
1-650-316-4400 · www.reasoning.com

C.1. INTRODUCTION

Illuma[SM]

Illuma is an automated software inspection service developed by Reasoning that rapidly detects critical structural defects in software. It is an important complement to functional testing, because it detects defects before testing and provides metrics that assist in riskassessment.

Deliverables

Results from an Illuma service inspection are provided in two reports. The Illuma Defect Metrics report provides high-level metrics results, and the Illuma Defect Data report includes detail for each individual defect. This document is the Illuma Defect Metrics report.

Application Overview

Reasoning inspected 736 user files in the ACE/TAO code, including all the source files.

Note that for billing and defect density computation purposes, the total number of lines in source files is used; that is, include files are not counted.

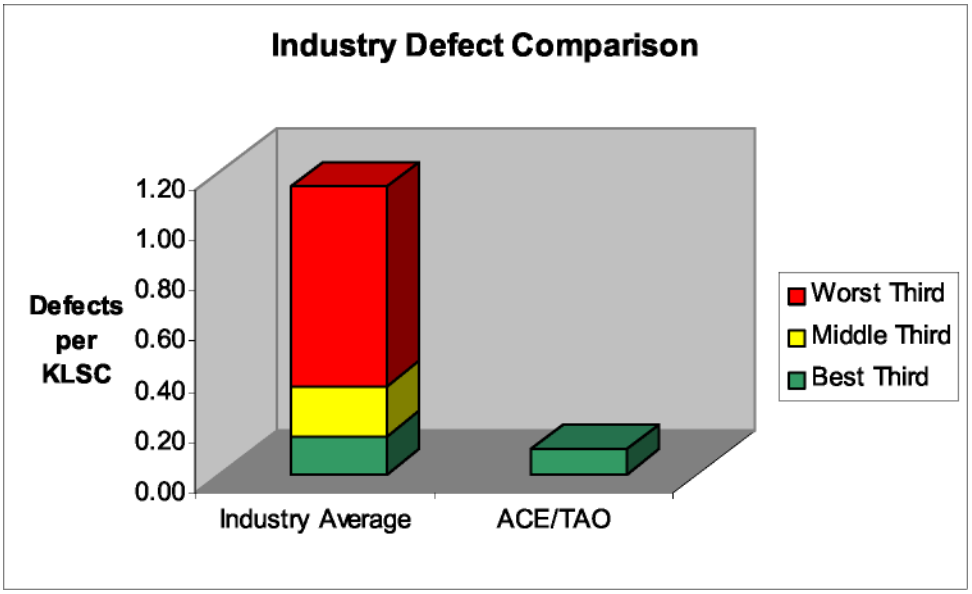
Application overview

Total Number of Source Files:	268
Number of User Include Files:	468
Total Number of User Files Processed:	736
Total Lines of Source Code in Source Files (LSC):	79,876
Number of Lines in User Include Files:	60,816
Total Lines of Code in Project:	140,692

C.2. Illuma METRICS

C.2.1 Industry Comparison

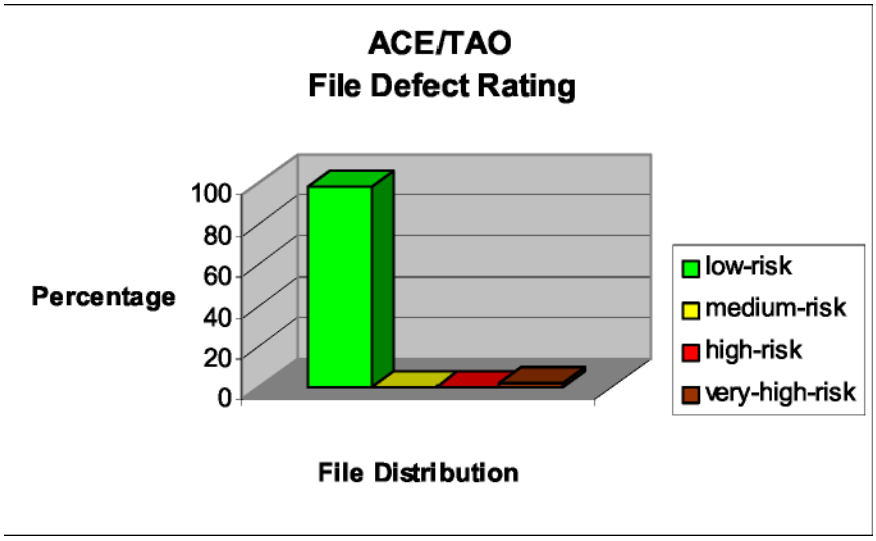
Reasoning found 8 defects in 79,876 source lines, a defect density of 0.10 Defects/KSLC. In a sampling of 160 projects totaling 22 million lines of code, 33% had a defect density below 0.15 Defects/KSLC (green), 33% had a defect density between 0.15 and 0.35 Defects/KSLC (yellow), and the remaining 33% had a defect density above 0.35 Defects/KSLC (red).



C.2.2. File Defect Rating

The following graph shows the percentages of files with low, medium, high and very high defect densities, defined with respect to the average defect density of this application:

- *Low* is less than 0.5 times the average defect density;
- *Medium* is between 0.5 and 1.0 times the average defect density;
- *High* is between 1.0 and 2.0 times the average defect density;
- *Very high* is higher than 2.0 times the average defect density.



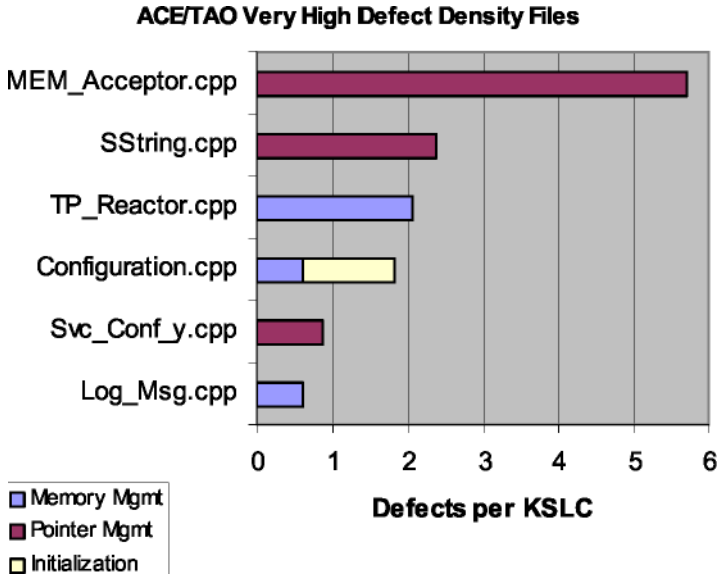
This graph shows how defects are distributed across the files. Some files may exhibit unusually high defect densities, rendering them more likely to fail in the field. Such files may require more testing or rewriting.

C.2.3. Very High Defect Density Files

In the ACE/TAO code, 6 files had a defect density in the very high range. The graph below shows the defect density in descending order for these files. It also shows how those defects are distributed over defect classes.

The defects contained in each defect class are listed below:

Defect Class	Defect
Initialization	Uninitialized Variable
Pointer management	NULL Pointer Dereference
Out of Bounds Array Access	
Memory management	Memory Leak
Bad Deallocation	



C.3. Defect Summary

The column *Defect Instances* in the table below details, per defect class, how many defects there are in the application.

The column *Files Affected* details, per defect class, the number of files in the application that have one or more defects. This is an indication of how much work is needed to fix the defects; many defects in one file are easier to fix and retest than the same number of defects scattered over many files.

Defect Summary

Inspection Class	Defect Instances	Files Affected
Memory Leak	3	3
NULL Pointer Dereference	3	3
Bad Deallocation	0	0
Out of Bounds Array Access	0	0
Uninitialized Variable	2	1
Total Defect Instances	**8**	**-**