SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **35**:899–921
Published online 14 April 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.655

SESAG: an object-oriented application framework for real-time systems

Pao-Ann Hsiung[1,*,†], Trong-Yen Lee[2], Jih-Ming Fu[3] and Win-Bin See[4]

[1]*Department of Computer Science and Information Engineering, National Chung Cheng University, 160 San-Hsing, Min-Hsiung, Chiayi, Taiwan–621, Republic of China*
[2]*National Taipei University of Technology, Taipei, Taiwan, Republic of China*
[3]*Cheng-Shiou University, Kaohsiung, Taiwan, Republic of China*
[4]*Aerospace Industrial Development Corporation, Taichung, Taiwan, Republic of China*

SUMMARY

Advancements in hardware and software technologies have made possible the design of *real-time* systems and applications where stringent timing constraints are imposed on critical tasks. The design of such systems is more complex than that of temporally unrestricted systems because *system correctness* depends on the satisfaction of functional as well as temporal requirements. To aid users in correctly and efficiently designing systems, *object-oriented frameworks* provide a useful environment for significant reuse and reduction in design effort. In contrast to other application domains, there has been relatively little work on an application framework for the design of real-time systems. Facing the growing need for real-time applications, we propose a novel application framework called SESAG, which consists of five components, namely *Specifier*, *Extractor*, *Scheduler*, *Allocator*, and *Generator*. Within SESAG, several design patterns are proposed and used for the development of real-time applications. A new evaluation metric called *relative design effort* is proposed for evaluating SESAG. Experiences in using SESAG show a significant increase in design productivity through design reuse and a significant decrease in design time and effort. Two complex application examples have been developed using SESAG and evaluated using the new evaluation metric. The examples demonstrate relative design efforts of at most 18% of the design efforts required by conventional methods. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented application framework design; real-time application development; task scheduling; resource allocation; code generation; framework evaluation; design patterns

*Correspondence to: Pao-Ann Hsiung, Department of Computer Science and Information Engineering, National Chung Cheng University, 160, San-Hsing, Min-Hsiung, Chiayi, Taiwan–621, Republic of China.
†E-mail: hpa@computer.org

Contract/grant sponsor: National Science Council, Taiwan; contract/grant numbers: NSC91-2213-E-194-008, NSC92-2213-E-194-003, NSC92-2218-E-194-009 and NSC93-2213-E-194-002

Received 16 March 2001
Revised 30 September 2004
Accepted 30 September 2004

1. INTRODUCTION

Real-time systems such as telecommunications, avionics, multimedia, robotics, and factory automation impose stringent timing constraints on tasks executions. Violation of any temporal constraint results in an incorrect system. To meet such temporal constraints, task scheduling and resource allocation have become crucial phases of all real-time systems design. It is here that object-oriented (OO) application frameworks can be taken advantage of because common design phases such as scheduling can be implemented as reusable classes and components. Thus, allowing application designers to concentrate on application tasks and not on reimplementing the wheel. Corresponding to different specification styles and scheduling policies, we need different design patterns. Combining design patterns and components, a novel OO application framework called SESAG is proposed for the design of real-time systems. SESAG helps users to efficiently design applications by minimizing design time and effort spent on routine tasks such as task scheduling, resource allocation, and code generation. Applying SESAG to two complex application examples shows that the relative design effort is only 18% of that without using SESAG.

A *real-time system* is generally specified as a collection of *tasks* which might share resources. The tasks are independent and periodic. Execution time, period, deadline, type of priority and resource requirements are specified for each task. Hard real-time systems do not allow the violation of any timing constraint, that is, no task can violate its deadline. Soft real-time systems strive to minimize deadline violations. To statically guarantee satisfaction of all timing constraints, the tasks must be scheduled using either *priority-based* scheduling algorithms such as rate-monotonic (RM) [1], earliest-deadline first (EDF) [1], mixed-priority (MP) [1], pin-wheel or using *time-based* scheduling algorithms such as cyclic scheduling or shared-based scheduling algorithms.

OO technology employs *encapsulation* and *inheritance* as basic reuse techniques. Objects are identified in a system, encapsulated, and positioned in a hierarchy of classes, by taking into account their inter-relationships. *Class libraries* are thus the basic structures for reusing objects. System designers found such libraries to be limited in their reuse capability because they strived to cater for a more general purpose reuse. Recently, several application-domain-specific techniques have been proposed, which significantly improve the degree of reuse. As introduced in the rest of this section, design patterns, software architectures, software components, and OO application frameworks are common reuse techniques, ordered ascendingly by their degrees of reuse.

Design pattern is a problem-solution pair that captures successful development strategy. Patterns aid in reusing successful design strategies by giving a more abstract view to concrete design strategies. For example, some core design patterns are Adaptor, Proxy, Facade, and Bridge [2].

In software architecture technology, there are four types of reusable entities: (1) architectural style, (2) architectural design, (3) architectural framework, and (4) architectural platform. For example, pipes and filters, functional decomposition, telecommunication infrastructure, and CORBA are examples of the four reuse entities in software architecture, respectively.

Software components are self-contained instances of abstract data types that can be integrated into complete applications. Examples include VBX Controls and CORBA Object Services. A component acts as a black-box allowing designers to reuse it through knowledge of its interface only.

An OO application framework (OOAF) is a reusable, 'semi-complete' application that can be specialized to produce custom applications [3]. OOAF can support application–domain specific reuses, for example in the domains of user interfaces or real-time avionics. Examples include MacApp,

ET++, Interviews, ACE, Microsoft's MFC and DCOM, Javasoft's RMI, and implementation of OMG's CORBA. Frameworks can be further distinguished by their scope into *system infrastructure frameworks* (used internally within a software organization), *middleware integration frameworks* (used to integrate distributed applications and components), and *enterprise application frameworks* (used in application-specific domains) [4].

Since OOAFs provide the highest degree of reuse, we propose a new OOAF, called SESAG, for developing real-time applications. Common tedious tasks encountered by a real-time application developer during the creation of applications have been incorporated into SESAG. In this way, a developer can devote more time and effort to the actual application tasks, instead of real-time system peculiarities. SESAG is modularized into five replaceable components that are used at different stages of generic application development. Currently, the most common application features have been implemented in SESAG such as task extraction, task scheduling, resource allocation, and code generation. More specific features such as network delay, network protocols, and on-line task scheduling are not currently in the scope of SESAG. SESAG ensures that the applications created by a user of SESAG satisfy all user-specified real-time constraints through the application of theoretically-proven scheduling algorithms. As far as other system performance characteristics are concerned, SESAG currently does not provide any sort of optimization, just leaving them to either the application developer or the environment designer. However, features such as context switch time and rate, external events handling, I/O timing, mode changes, transient overloading, and setup time will be incorporated into SESAG in the future.

The above short description sets SESAG in a generic scope such that most real-time applications can be developed using SESAG, but with varying efforts. Simplicity has been a major goal in SESAG's development, while at the same time providing extension flexibility so that different applications can be designed. The target systems designed by SESAG are limited to real-time systems that have statically schedulable periodic tasks. Safety analysis and verification are also not within the scope of SESAG, only scheduling feasibility is considered.

The rest of this article is organized as follows. Section 2 gives some previous and related work on applying OO technology to real-time system design. Section 3 describes the five components of SESAG using a *components-patterns view*. Section 4 describes all the classes provided in SESAG, thus giving a *class view* of SESAG. Section 5 describes the implementation of SESAG and proposes an evaluation metric for frameworks such as SESAG. Section 6 illustrates how a designer may use SESAG to develop a real-time application. Section 7 presents the experimental results of two complex examples developed using SESAG. Section 8 gives the final conclusion with some future work.

2. PREVIOUS WORK

OO technology has been used in the development of real-time systems for quite some time. Research literatures have shown how the concept of objects is useful in real-time systems. OO real-time system models such as MO2 [5], evaluation taxonomy such as in [6], OO real-time language design [7,8], concurrency exploitation in OO real-time systems using metrics-driven approach [9], checking time constraints [10], and verification of function and performance for such systems [11] are some of the recent work on applying OO technology to real-time system design.

Though OO technology has been applied to the design of real-time systems in several proposed works, as yet there has been little work on the development of an OOAF for real-time system application design. An *OO real-time system framework* (OORTSF) [12–14] is a simple framework showing the classes used in the development of real-time applications. No design patterns were proposed specific to real-time system application design. This results in a difficult comprehension of the collaboration among the classes. Exactly how a real-time application is developed using OORTSF is not very clear from the work. It is also unclear how the application code is generated from OORTSF, how the resources are allocated, and how the tasks are scheduled. Further, the flexibility of specifying real-time objects, the ease of using OORTSF, the benefits of applying OORTSF, and other issues related to OOAFs are not described in the work.

According to the knowledge of the author, besides OORTSF, there is no other work on *enterprise application frameworks* devoted to general real-time system application development. As far as *middleware integration frameworks* are concerned, the TAO real-time object request broker has been proposed by Schmidt [15].

3. SESAG COMPONENTS

SESAG derives the name from its five constituent components namely *specifier*, *extractor*, *scheduler*, *allocator*, and *generator*, ordered by the sequence in which they are used. For ease of comprehension, we present two different views of the framework: a *components-patterns view* and a *class view*. The components-patterns view of the framework allows a designer to better understand *how* exactly SESAG must be used and the class view allows him/her to grasp *what* exactly are the classes to be used. Our presentation is thus based on two complementary levels of description: an *abstract* components-patterns view (presented in this section) and a *concrete* class view (presented in Section 4).

Figure 1 illustrates the *components-patterns view* of SESAG using *Unified Modeling Language* (UML) notations [16]. An application designer uses the five components of SESAG as follows. Objects specific to a real-time application are specified by a designer using Specifier. Real-time constraints are specified in two ways: (1) embedded within user-defined application specific objects, or (2) independently using the *Timed Object Constraint Language* (OCL). In the former case, Extractor is used for extracting constraints. Extractor is also used to extract tasks from the given domain objects. Scheduler schedules the tasks using some scheduling algorithm and Allocator allocates resources among the tasks that are running concurrently. Finally, Generator is used to generate the application code based on the decisions made in the other components.

3.1. Specifier

Specifier is the main interface component between a user and SESAG. Using Specifier, a user of SESAG defines objects specific to an application under design, which are called *application domain objects* (ADOs). Three design patterns are used in this component: *objects-with-constraints*, *objects-and-constraints*, and *tasks-with-constraints*. We call them design patterns because often the real-world objects are not specified as tasks, whereas a real-time system application is generally described in terms of a set of canonical tasks. This semantic gap has become a design issue [10] and topic of research [8] for the OO model of real-time systems. The three design patterns correspond, respectively, to how

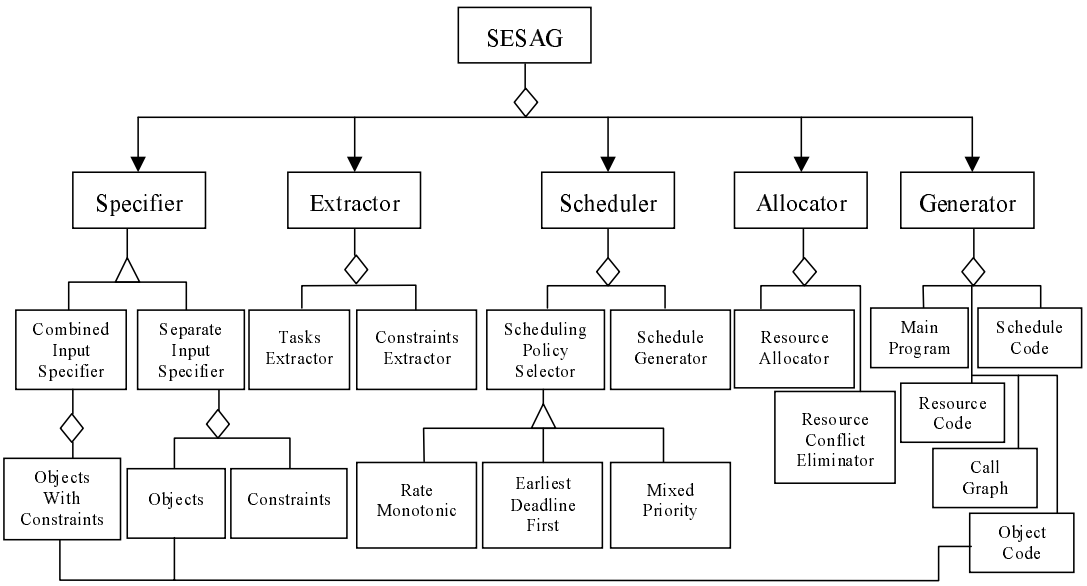

Figure 1. Components-patterns view of SESAG.

SESAG provides a designer with the flexibility of choosing: (1) to specify application domain *objects* with constraints *coupled* to the objects; (2) to specify application domain *objects* with constraints as a *separate* entity; (3) to specify real-time *tasks* with constraints. The first two design patterns do not explicitly specify what the tasks are, which are instead automatically extracted by the Extractor component in SESAG, as described Section 3.2.

For the objects-with-constraints pattern, there are two types of method in an object namely *event-triggered* and *time-triggered*. Event-triggered methods are invoked by other methods or by the main function. Once started, time-triggered methods are invoked periodically, until stopped. When a method has timing constraints associated with it, it is called a time-triggered method, otherwise it is event-triggered. Timing constraints consist of the specification of at least *period* and *deadline*. Other timing constraints such as *arrival time*, *execution time*, *stop time* may also be specified. There are two ways in which timing constraints may be specified with a time-triggered method, including (1) *annotations* in a programming language such as C++ [10], and (2) language extensions such as a real-time extension of C++ called RTC++ [7], ARTS/C and ARTS/C++ used in the ARTS real-time distributed operating system kernel [17], real-time Euclid [18], real-time Mentat [19], or FLEX [20]. SESAG supports the first method namely C++ with annotations. An annotation example is shown in Figure 2(a), where the class `xyz` has two real-time tasks represented by time-triggered methods `t1` and `t2` with periods 5 and 3, deadlines 10 and 12, respectively. It also has one event-triggered method `t3` that is invoked by both `t1` and `t2`.

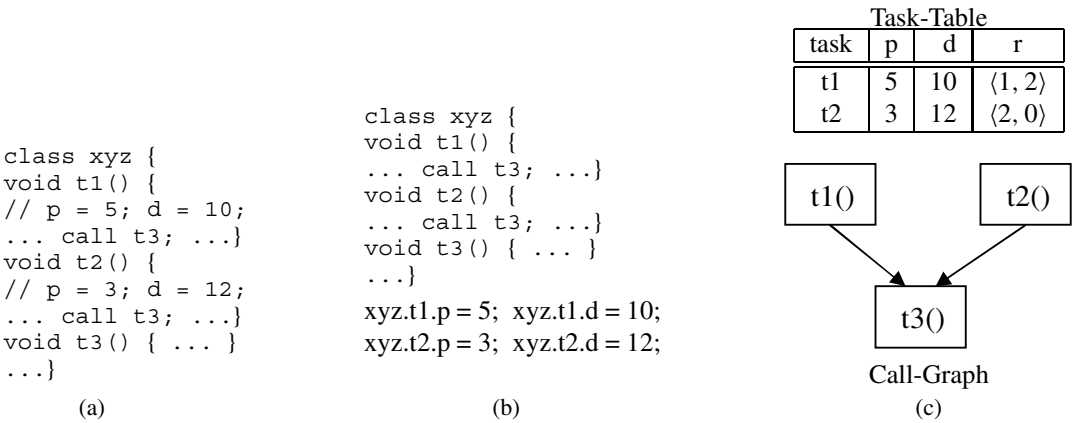

Figure 2. An illustration example using the three design patterns in specifier: (a) objects-with-constraints; (b) objects-and-constraints; and (c) tasks-with-constraints.

For the objects-and-constraints pattern, constraints are specified using the OCL that was proposed along with UML. An example is shown in Figure 2(b) that is equivalent to the one in Figure 2(a).

For the tasks-with-constraints pattern, a set of real-time tasks are specified along with timing constraints and resource requirements by instantiating two classes: *Task-Table* and `Call-Graph`, where the former records information about each task and the latter record the invoke relationships among the tasks. An example is shown in Figure 2(c) that is equivalent to the one in Figure 2(a). Here, the resource usages are explicitly specified as $r = \langle r_1, r_2 \rangle$, where $r_1, r_2 \in \mathcal{Q}_{\geq 0}$ are the amount of resources of type 1 and type 2 used by a task and $\mathcal{Q}_{\geq 0}$ is the set of non-negative rational numbers.

3.2. Extractor

The Extractor component transforms the specification provided by a designer within Specifier into a uniform intermediate format suitable for schedulability analysis. The intermediate format is necessary since Specifier allows a designer to have several choices of specifying his/her application. The main job of Extractor is to extract important information from the objects and formulate them into two parts: a `Task-Table` object and a `Call-Graph` object.

For users using the objects-with-constraints pattern for specification in an application domain object, there may be one or more time-triggered methods, each of which represents a real-time task. The time constraints associated with each time-triggered method are automatically extracted by Extractor. For users using the objects-and-constraints pattern for specification, each method that has associated OCL constraints is extracted as a real-time task with time constraints. All task related information are then stored in a `Task-Table` object for reference by other SESAG components. Each task record in a *Task-Table* object consists of the task index, the associated method name, its execution time, period, deadline, type of priority (fixed or dynamic), and its resource requirements. The resource requirement is specified as a real-numbered vector, where each element corresponds to some system resources

such as memory, processor utilization, ..., and the real-number corresponds to the amount of each resource required by the particular task. System resources are specified by a designer within Specifier by defining passive application domain objects that can only be accessed. Figure 2(c) shows an example of a `Task-Table` that can be automatically instantiated from the user requirements in Figure 2(a) or (b).

Extractor also generates a `Call-Graph` object, which is a directed graph $G = (V, E)$, nodes in V represent tasks and arcs in E represent the call relationships between two tasks. Each time-triggered method or method with associated OCL constraints is traversed to find all the other methods that are invoked by it. Arcs are then generated from the caller to the callee. For example, method `t3` is invoked by both methods `t1` and `t2` in Figure 2(c). As shown in the following sections, the `Call-Graph` is useful for schedulability test, resource allocation, and conflict resolution.

3.3. Scheduler

The `Call-Graph` and the `Task-Table` objects, either generated by Extractor or instantiated by a user in Specifier, are scheduled into a feasible application by Scheduler. There are many priority-based real-time scheduling policies such as rate-monotonic, earliest-deadline first, and mixed priority [1]. It is sometimes evident from the application as to which scheduling policy should be applied. However, in most applications, the prime concern is the satisfaction of the timing constraints, irrespective of which scheduling algorithm is applied. Scheduler includes a design pattern similar to the *strategy pattern* [2] adapted to real-time systems. In this pattern, one and only one scheduling algorithm must be chosen from the set of all scheduling algorithms for scheduling the tasks in the `Call-Graph` and the `Task-Table`.

This component mainly consists of two parts: a *policy selector* and a *schedule generator*. A designer can choose to assign a particular scheduling policy he deems fit or the designer can also choose to allow SESAG to automatically determine the right choice. The choice is made by performing schedulability tests with respect to each scheduling algorithm. One of the scheduling algorithms in the successful cases is then selected as the automatic decision result. This selection can be arbitrary or based on some criteria such as the shortest schedule length (i.e. the shortest scheduled time). Currently, SESAG leaves this option to the designer. The schedule generator generates the actual start/end timing of each task based on the schedule policy chosen and on the `Call-Graph` constraints such as precedence relationships. Resources and their allocation are not considered here as they will be dealt with in allocator.

The tasks recorded in a `Task-Table` are not independent because they have precedence relationships among themselves as given by the corresponding `Call-Graph`, thus the set of tasks that is concurrently ready to run is not static, but changes dynamically depending on which task is scheduled to run at any given time. In the scheduling terminology, this means that the *initial phase* of each task is constrained by the latest end-time of all of its predecessors in the `Call-Graph`. The tasks are all assumed to be preemptive as required by the scheduling policies. In SESAG, static schedules are generated by applying EDF or RM scheduling to each set of concurrently enabled task at any point of time. When the deadlines are equal to corresponding periods of all tasks, the schedules generated by SESAG using EDF and RM are identical. An EDF unschedulable task set is also RM unschedulable, hence EDF is more powerful in terms of scheduling a given task set and of utilizing a microprocessor processing time. However, it is easier to ensure that deadlines will be satisfied by RM than by EDF.

Due to priority-based scheduling, the well-known *priority inversion problem* exists. This problem occurs when a higher-priority task is blocked from execution due to resources being held by a lower-priority task. A middle priority which is not blocked can then run for a long time preventing both the low- and high-priority tasks from running. To solve this problem, we adopt the priority inheritance approach [21].

Consider the illustrative example given in Figure 2, suppose the execution times for tasks `t1` and `t2` are 2 and 1, respectively, and that they are concurrently enabled. Then, their processor utilization is $\frac{2}{5}$ and $\frac{1}{3}$, respectively, which gives a total utilization of $\frac{11}{15} = 0.733$. Thus, by the schedulability constraint given in [1], we can conclude because $0.733 < 0.83$, where 0.83 is the upper bound for two tasks, the system is schedulable using RM. The RM schedule for this example is thus $\langle$`t2`, `t1`$\rangle$.

3.4. Allocator

Resource allocation is handled by this component. The scheduled `Call-Graph` does not yet contain any resource information. It is in this component that resources are allocated to each task based on its resource requirements as recorded in the `Task-Table` and defined in Definition 1.

Definition 1 (Task resource requirement). Given a system with n resources, a task resource requirement for a task t is defined as $r(t) = \langle r_1, r_2, \ldots, r_n \rangle$, where $r_i \in \mathcal{Q}_{\geq 0}$ is the amount of the ith resource required by task t and will be denoted in short as $r_i(t)$.

Allocator is composed of two modules: *resource allocator* and *conflict eliminator*. The former allocates resources to each task and detects resource conflicts, which are then eliminated by the latter.

It may happen that certain tasks, scheduled to be simultaneously executing, conflict in their resource requirements. For example, if the resource requirements of task A is $\langle 0, 1, 3 \rangle$ and that of task B is $\langle 3, 2, 1 \rangle$, suppose the system has only three of each resource type, then tasks A and B cannot be simultaneously executed as they totally require four of the last type of resource while the system has only three. In general, a resource conflict is defined formally as in Definition 2.

Definition 2 (Tasks with resource conflict). Given a set of m concurrently enabled tasks $\{t_1, \ldots, t_m\}$ that use n system resources such that the amount of available ith resource is $\max_{r_i} \in \mathcal{Q}_{\geq 0}$, the tasks are said to be in conflict if there is some jth resource such that $\sum_i r_j(t_i) > \max_{r_i}$.

Several conflict resolution methods can be used. For example, rescheduling of both tasks (as in the message-passing CSMA/CD network protocol), or rescheduling of any one of the tasks. We adopt a straightforward solution to resource conflicts in the same principle as priority inheritance. Suppose a task is using some resource, then all other tasks that arrive later and need the same resource are delayed and rescheduled. Conflicts in resource requirements of scheduled tasks are thus eliminated and the resulting scheduled `Call-Graph` is a feasible one for code generation.

Take the illustration example from Figure 2 again. As described in Section 3.3, the scheduler generated a schedule $\langle$`t2`, `t1`$\rangle$ for this example with the two tasks `t2` and `t1` having execution times 1 and 2. The schedule generator from the scheduler component generates the start and end times for each task shown in Figure 3, marked by *Original schedule*. In this schedule, task `t2` preempts task `t1` at time 6 because `t2` arrives at 6 and its priority by RM scheduling policy is higher than that of `t1`. Now suppose there are two instances of the first type of resource in this system. As we see from the task resource requirements in Figure 2, the two tasks totally need 3 instances of the first resource.

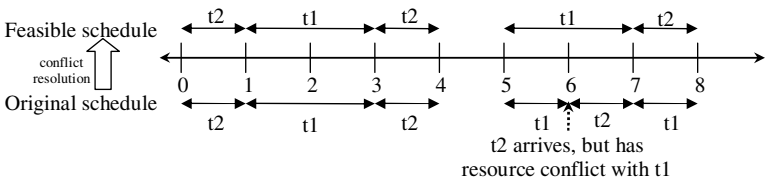

Figure 3. Example of resource conflict and resolution.

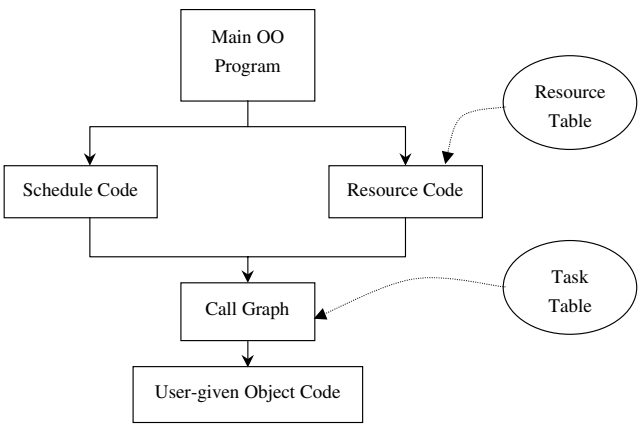

Figure 4. Code hierarchy in SESAG.

Thus, `t2` cannot run by preempting `t1` at time 6 because `t1` has one of the first resource which is needed by `t2`. Thus, there is a resource conflict here. By the conflict resolution mechanism in SESAG, `t1` is allowed to run to completion and `t2` starts at time 7. This new schedule is a feasible one and is shown as *Feasible schedule* in Figure 3.

3.5. Generator

Generator is responsible for producing OO code for a real-time application under development by a designer using SESAG. The hierarchical structure of the generated code as shown in Figure 4 consists of five parts: a main OO program, schedule code, resource allocation code, `Call-Graph` code, and user-given application domain object code. The hierarchy is a *calling* hierarchy, that is, the main OO program calls schedule code functions and the resource allocation code functions, which in turn call the `Call-Graph` code methods, and finally the user-given object code is called for execution. Two auxiliary codes used for reference are `Resource-Table` and `Task-Table`, which contain all the information for system resources and tasks, respectively.


```
(1)     INT MAIN()
        { // var declarations
(2)       foreach schedule time point { // see Figure 3
(3)           if (SETJMP(jmpbuf)==0) then SET_ALARM(next_time_point);
(4)                                  else continue;
(5)           while (REQUEST_RESOURCE(task)==NULL) ;
(6)           CALL_RESUME(task);
(7)           RELEASE_RESOURCE(task);   // interrupts disabled during release
(8)           WAIT(); } }
```

Algorithm 1. Pseudo code for the example.

The main OO program maintains a global clock which is used for recording progress in the developed system or application. It also contains exception handlers for error recovery, fault handling, and other mechanisms to handle exceptions such as constraint violations. The main program ensures that the system is always in an acceptable state. This is achieved by calling the schedule code functions and resource handling functions in the resource allocation code.

The schedule code is an implementation of the schedules of the tasks in the `Call-Graph` and after all resource conflicts are eliminated. This code depends on the scheduling policy selected either by the user or by SESAG. The schedule code consists of the actual time a task must start execution, is preempted, is resumed, and is terminated. Everything is settled statically for optimal performance and satisfaction of timing constraints in real-time systems.

Resource-related information are recorded in a `Resource-Table` object. Resource allocation code is responsible for accepting resource allocation/deallocation requests, transmitting them to the resource codes (or the actual physical resources in case of an embedded real-time system), handling exceptional situations such as dynamic resource failure and recovery, and resource conflict handling. Although resource conflicts among tasks were eliminated statically in the scheduled `Call-Graph` (refer to the Allocator component of SESAG), conflicts may still arise in exceptional situations when resources become faulty or when tasks violate timing constraints due to external environmental disruptions.

The `Call-Graph` code is an implementation of the resource-allocated, scheduled `Call-Graph` object. This code is required in spite of the calling scheme of tasks (method calls) being implicit in the user-given domain object code, because scheduling information and resource requests are, respectively, not given and not explicit in the object code. The object code is just the user-given code for all the application domain objects.

The hierarchical structure of the code generated by SESAG has many advantages including easy debugging, code modularization, and explicit interface with user-given code.

For our illustration example from Figure 2, which was scheduled and resource allocated in the previous sections, a partial pseudo code is shown in Algorithm 1, where static preemptive schedules are executed along with resource requests and releases.

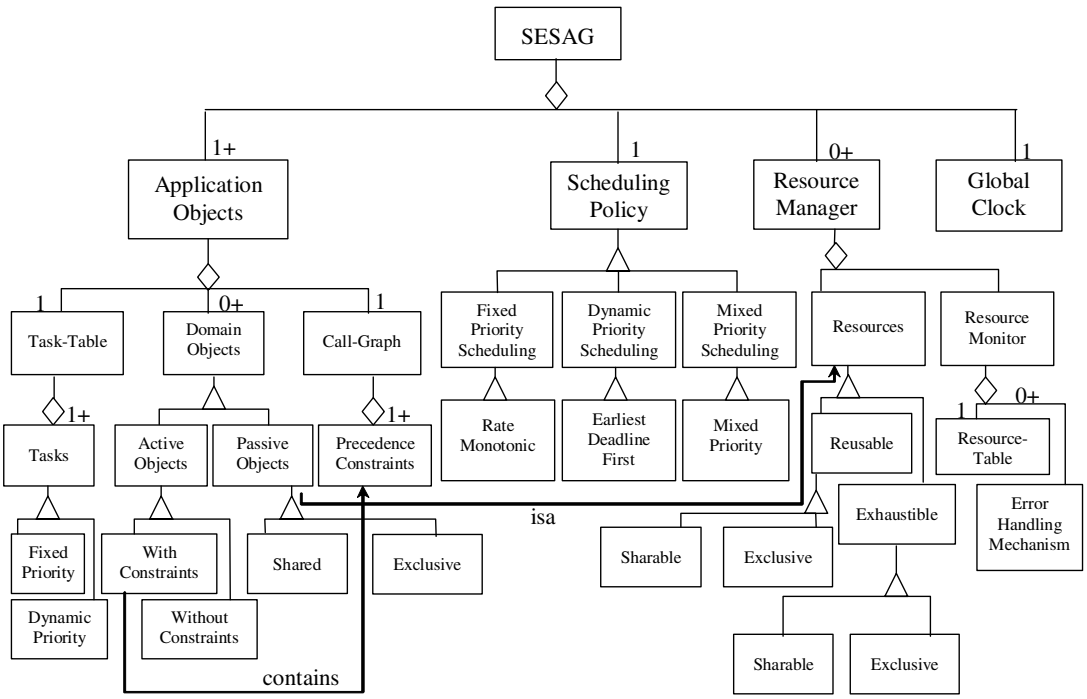

Figure 5. Class view of SESAG.

4. SESAG CLASSES

After an overview of how SESAG generates an application by using the five constituent components, we now describe the classes in SESAG that are used by a designer to specify an application. Figure 5 shows the hierarchy of classes provided by SESAG. The notation used is UML [16]. SESAG classes are classified into three categories: *application objects*, *scheduling policy*, and *resource manager*. There is also a `Global-Clock` class that is responsible for maintaining the global timing of a system under design. A designer can instantiate any class in the hierarchy by filling in the data attributes and the member function definitions or derive a new class from the given classes. When information is partially entered, requests for more data values or function definitions will be made by SESAG after an overall analysis.

4.1. Application specification

In SESAG, there are two ways to specify the objects in a real-time application. A designer can either choose to define application domain objects that represent real-world entities used in an application,

or to instantiate two given classes, `Task-Table` and `Call-Graph`, which provide a more abstract but useful view of an application. When specification is made in the form of application domain objects, the Extractor in SESAG (as introduced in Section 3.2) automatically instantiates the `Task-Table` and `Call-Graph` classes.

The `Task-Table` class is an aggregation of one or more `Task` class(es), each of which consists of data attributes such as the computation time of each instance of a task (called a *job*), the period of execution, the deadline for a job (this is usually the same as its period, but could also be a multiple), the phase (this is normally assigned values after scheduling and resource conflict elimination), and the resource requirements (this is a vector specifying how much of each resource is required). `Task` class also consists of member functions such as task-setup(), task-execution(), task-start(), task-end(), request-resource(), and deadline-violation-handler().

The `Call-Graph` class is defined as a separate class so as to segregate the task objects from their call relationships. This class mainly consists of data attributes such as precedence constraints (the predecessors and the successors of each task) and delay constraints (the time delay between two tasks). The member functions include context-switch(), context-delay(), and switch-error-handler().

Application domain objects are defined by a designer and are classified into two types: *active* and *passive*. An *active* object accesses and changes the state of other objects, whereas a *passive* object is one whose state is accessed and changed by other objects and who does not access or change the state of other objects. For example, a system controller is an active object, whereas a database or memory record is a passive one. This classification is useful for identifying system resources. Proper access of system resources is critical to system correctness. An active object can be instantiated in two ways: *with* and *without* constraints. In the case of without constraints, a designer must instantiate the `Task-Table` class at the same time to specify the task constraints. Constraints can also be specified inside the objects as method annotations in C++ [10]. A passive object can be *shared* or *exclusive* depending of the characteristic of the resource being modeled.

4.2. Scheduling policy

Application is statically scheduled in SESAG for optimal performance, hence task schedules are embedded within the generated application code. `Scheduling-Policy`, a SESAG class, is responsible for providing the schedule code for the statically scheduled tasks. Three kinds of priority-based scheduling policies are supported in SESAG, namely fixed, dynamic, and mixed. For fixed priority-based scheduling, the popular rate-monotonic scheduling algorithm is supported. For dynamic priority-based scheduling, the earliest deadline first scheduling algorithm is supported. For mixed priority, the scheduling algorithm given in [1] is supported. The data attributes of the `Scheduling-Policy` class include priority type, scheduling algorithm, and scheduling mode (automatic or user-specified). Member functions include assign-task-start-time() and assign-task-end-time().

4.3. Resource manager

Resources in a real-time system can be user-defined through the definition of new passive application domain objects by a designer, as described in Section 3. Resources can also be standard ones such as flash memory and communication interfaces which are represented as an explicit reusable

`Resource` class. All resources defined and instantiated by a designer are collected into a `Resource-Table` class that contains all resource related information. Errors related to resource problems such as resource conflicts and read/write failures are handled in SESAG by the instantiation of one or more `Error-Handler` classes. A pair of classes `Resource-Table` and `Error-Handler` is aggregated into a single `Resource Monitor` class.

Each `Resource` object has data attributes including use-type (exhaustible or reusable), share-type (sharable or exclusive use), value (it may be a simple integer or a complex record), and status (currently in use, free, or allocated). Member functions of a `Resource` object include grant-resource(), free-resource(), read-value(), change-value(), and get-status(). The `Error-Handler` class mainly handles errors due to resource conflicts such as when two or more tasks need the same resource but the amount of that resource available is not enough for their requirements. This class has a method called conflict-handler(), which resolves conflicts according to the resolution method described in Section 3.4.

4.4. Global clock

This special class maintains the system time. The temporal increment can be changed by a designer as required. The initial value of `Global-Clock` is zero and it increments at a uniform rate. The data attributes include value (the time of the system), initial value, breakpoint value, and final value. Member functions include start-timer(), stop-timer(), reset-timer(), record-breakpoint-value(), change-increment(), and read-value().

5. FRAMEWORK IMPLEMENTATION AND EVALUATION

SESAG is implemented in C, C++, and Java. It generates application code also in these three languages, however, the Java code conforms to the real-time specification for Java. The previous two sections described what each component of SESAG does and the patterns and classes that may be used by an application designer. Before describing how one actually develops an application program using SESAG, we describe in this section how SESAG is implemented by describing *auxiliary programming* and *class programming*, thus co-relating the two views of SESAG described in Sections 3 and 4.

5.1. Auxiliary programming

SESAG was implemented in five parts, each corresponding to one of the five components. Some classes are shared among the parts, while some were exclusively implemented. Besides the implementation of user-level classes as described in Section 4, special-purpose functions such as graph algorithms, decision schemes, and database managements are indispensable for the implementation of a framework. Our experiences in the implementation of SESAG show that the following auxiliary programming can be generalized.

1. *Object interface*: interactions among user-defined objects and framework objects require type-checking, parameter transformation, constraints checking, and interface generation. All of these are implemented as special-purpose functions.

2. *Information extraction*: information extraction from user-defined objects such as task extraction and constraints extraction in SESAG require scanning of the object specification and human domain-expert knowledge. All of these are also implemented as special-purpose functions.
3. *High-level manipulation*: high-level objects are often composed of a network of elementary objects which form some kind of interconnection topology, for example, SESAG uses a `Call-Graph` for a uniform intermediate format. Manipulation of such high-level objects requires graph algorithms, such as graph traversal and shortest-path.
4. *Database management*: objects that rarely do anything actively are termed as passive objects. Physically, these objects are resources or some form of databases. For example, in SESAG we have `Task-Table` and `Resource-Table`. When such resources or databases become large, we require database management procedures to ensure the most efficient handling which is critical to real-time processing.
5. *Design automation*: design automation besides design reuse is another crucial issue in both hardware and software. It is our experience that automation in frameworks often helps the user in making complex decisions, thus saving considerable design time. By automation in frameworks, we mean the framework uses some decision algorithms to decide upon an optimal or near-optimal solution to some difficult problem at hand. For example, in SESAG the scheduling policy needs to be decided before the tasks can be scheduled. This decision is complex in the sense it is difficult for a human to accurately schedule several tasks using different algorithms. SESAG uses a decision procedure as described in Section 3.3 to decide upon a good scheduling policy.

5.2. Framework evaluation

It is difficult to evaluate a framework since each application developed using the framework may exhibit different characteristics pertaining to the specific domain. In our experiments using SESAG, we found that though applications differ from each other, yet the complexities of the applications are generally indicative of how much design effort can be saved using a framework instead of designing from scratch or some simple libraries. Our experimental results indicate that the more complex an application, the more design effort is saved through the use of a framework. This is intuitive because a complex application requires greater effort to design and if much of the decision and house-keeping tasks are carried out by a framework, then the design time can be reduced significantly.

The new evaluation metric we propose for OOAFs is called *relative design effort* (RDE) as defined in the following.

Definition 3 (RDE). *RDE* is the sum of products of *application complexity fraction* and *design time fraction*. The summation is over the team members involved in a project. Application complexity fraction is the fraction of application domain objects and the total number of objects in the final application program. Design time fraction is the fraction of design time required for developing the application with a framework and that without the framework.

$$\mathrm{RDE}(t) = \sum_i \left(\frac{\#\mathrm{ADO}_i}{\#\mathrm{ADO}_i + \#\mathrm{AFO}_i} \times \frac{\mathrm{FT}_i + \mathrm{IT}_i + \mathrm{LT}_i(t)}{\mathrm{FT}'_i + \mathrm{IT}'_i} \right) \tag{1}$$

where t is a measure of the time elapsed since starting use of SESAG, i represents the index of the ith team member, $\#\mathrm{ADO}_i$ is the number of application domain objects designed by the ith team member,

$\#AFO_i$ is the number of application framework objects in the ith application subprogram (without counting the application domain objects), FT_i is the design time using the framework, FT'_i is the design time without using the framework, IT_i is the integration time required by the ith team member using the framework, IT'_i is the integration time required by the ith team member without using the framework, and $LT_i(t)$ is the non-monotonic framework learning time required by the ith team member, which may depend on a lot of factors in addition to time, such as programmer experience, capability, etc., and which decreases as time increases.

From Equation (1), we observe that the value of the metric decreases when the complexity of an application complexity increases because the first term, application complexity fraction, increases to a value approximately one while the design time fraction approaches zero. This shows that the more complex an application is, the smaller is the relative design effort. This is by no means an indication of the *absolute* design effort since the actual design effort must increase with application complexity. The relative design effort is merely an indication of how well a framework helps an application designer to save design efforts (in terms of coding objects and coding time). We will use this metric to evaluate SESAG in later sections.

For example, if there is only one team member, 25 application domain objects, 10 application framework objects, and the design time with framework is a half of that without framework, then the RDE for this application is $\frac{5}{14} = 35.7\%$, whereas the absolute design effort is $25 \times t$ when framework is used and $35 \times 2 \times t = 70 \times t$ when the framework is not used, where t is the average design, integration, and learning time using the framework.

6. APPLICATION DEVELOPMENT

After an overview of the SESAG framework and its implementation. This section will describe how one designs a real-time application using SESAG. Figure 6 illustrates the development strategy (central column) in context with both the components-patterns view (left column) and the class view (right column) of SESAG. Rectangular boxes represent processes to be accomplished either by the user or by SESAG as indicated on the left corner of the boxes. Diamonds represent a branching choice. Ovals represent class instantiations. There are three phases in the application development of SESAG, namely, *specification*, *integration*, and *generation*.

From our experiences, irrespective of the type of OOAF, we recommend the 'SIG' strategy for application development (SIG stands for *specification*, *integration*, and *generation*). Such a modularized development will be helpful in program debugging, code generation, and future maintenance. In the following we describe each part of the SIG strategy for SESAG. Two example applications on avionics and vehicle control developed using SESAG are illustrated in Section 7.

- *Specification*. A user of SESAG (also called an application designer) begins with specifying his/her system. The user may either define application domain objects (such as in the autonomous intelligent cruise controller (AICC) example presented in Section 7) or directly input tasks by instantiating the `Task-Table` class and the `Call-Graph` class (as in the avionics example of Section 7). When application domain objects are specified, `Task-Table` and `Call-Graph` are instantiated by SESAG through the Extractor component. An example of real-time applications has been developed using SESAG as described in Section 3.

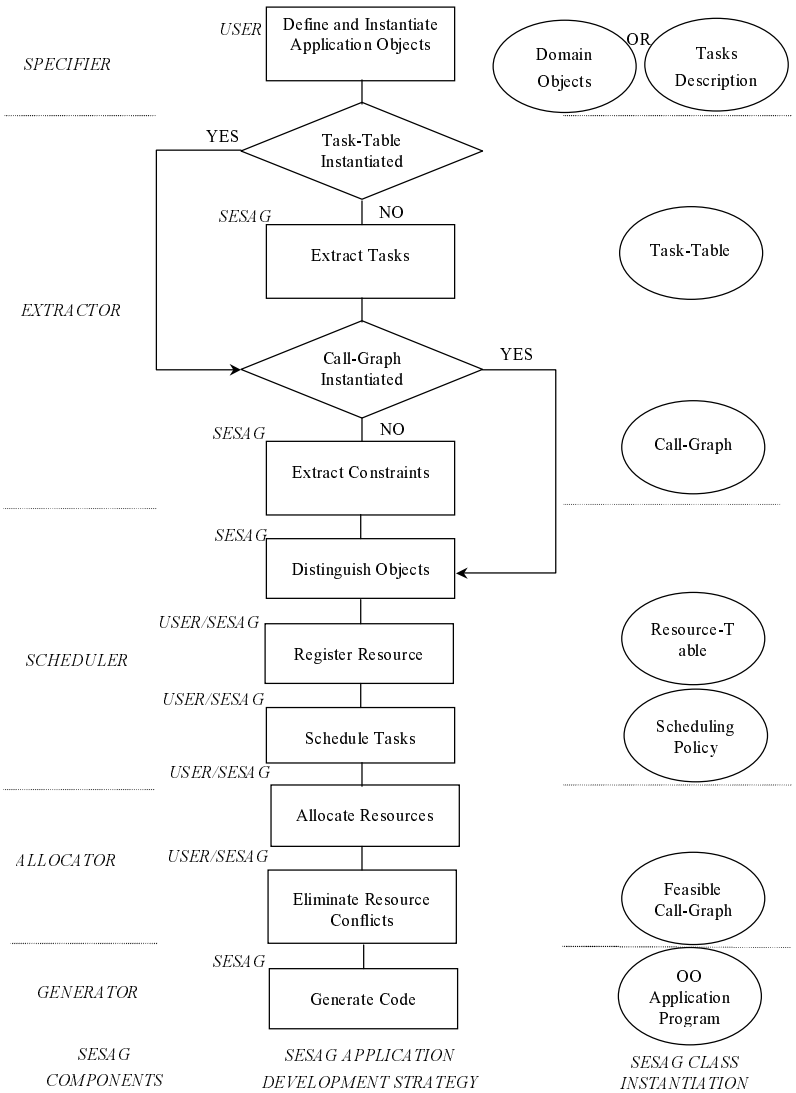

Figure 6. Application development using SESAG.

- *Integration*. At this stage, we have the `Task-Table` and `Call-Graph` instantiated either by the user or by Extractor in SESAG. SESAG then distinguishes passive objects from active ones. This distinction is required so that implicit resources may be classified as resources and registered in the `Resource-Table` class. Any object that does not actively call other object methods is called a passive object, otherwise it is an active one. Tasks from the `Task-Table` and their interdependencies from the `Call-Graph` are scheduled using a user-specified scheduling policy or through automatic decision by SESAG (as described in Sections 3.3 and 5). Resources are then allocated to the scheduled tasks. Resource conflicts are eliminated as described earlier in Section 3.4.
- *Generation*. After integration, we already have the skeleton for a feasible application where tasks have been explicitly distinguished, registered, scheduled, resources allocated without conflicts, and timing constraints satisfied through scheduling. Code generation is carried out by SESAG as described in Section 3.5.

The above description gives an idea of which parts a user must define and code, what SESAG does at each stage, and how the application is completely crafted into a working program. In contrast to commercial code generators based on the RT-UML design language [22], SESAG is not only a code generator, but it also helps schedule concurrent tasks such that real-time and resource constraints are satisfied. Thus, code generated by SESAG will meet user-given constraints, whereas code generated by RT-UML-based tools may not meet user-given constraints.

7. APPLICATIONS AND EXPERIMENTAL RESULTS

Two application examples developed using SESAG are presented in this section. An avionics example consisting of 24 tasks used to control an aircraft and a cruiser example consisting of 12 tasks used to control the vehicle speed under different circumstances. Both examples are industrial ones whose specifications [23,24] were used for development in SESAG. The benefits of using SESAG in developing the two examples have been evaluated and the results show a marked improvement in design productivity and efficiency. The relative design efforts (see Definition 3) for both the examples were quite small, 9 and 18%, respectively.

7.1. Avionics example

An illustrative example is an avionics system application: digital flight control [23]. The 24 tasks in this example are specified as shown in Table I [23,25,26]. The hardware resource for executing these tasks is the Software-Implemented Fault-Tolerance (SIFT) computer [25] as shown in Figure 7, where P is a processor, M is memory, and S is a switch. We consider eight processors, with each processor having an instruction execution rate of 0.5 MIPS and an address space of 64 Kbytes. Since the total utilization [23] of the tasks on a single processor does not exceed the rate-monotonic scheduling upper bound of 0.693 [1], rate-monotonic scheduling is used. The `Call-Graph` as produced using SESAG is shown in Figure 8. There were three resources identified by SESAG, namely, `Graphics-Display`, `Text-Display`, and `Memory`. Each of them required a separate `Error-Handler` class to resolve conflicts. This application when developed with SESAG took only one week for

Table I. Avionics example: digital flight control tasks [23].

Index	Task description	Execution time (ms)	Period (ms)	Utilization	Memory
1	Attitude control	2.456	50.00	0.049 12	2075
2	Flutter control	0.276	4.00	0.069 00	92
3	Gust control	0.116	4.17	0.027 84	60
4	Autoland	0.684	6.25	0.109 44	1025
5	Autopilot	0.400	200.00	0.002 00	250
6	Attitude director	5.120	33.33	0.153 60	1310
7	Inertial navigation	2.700	40.00	0.067 50	2250
8	VOR/DME	1.540	200.00	0.007 00	300
9	Omega	1.600	200.00	0.008 00	505
10	Air data	0.400	200.00	0.002 00	135
11	Signal processing	3.500	5000.00	0.000 70	315
12	Flight data	11.040	200.00	0.055 20	550
13	Airspeed	1.098	62.50	0.017 57	430
14	Graphics display	7.950	125.00	0.063 60	6250
15	Text display	3.800	100.00	0.038 00	9340
16	Collision avoidance	0.064	1.49	0.042 88	1150
17	Onboard communication	0.056	4.00	0.014 00	705
18	Offboard communication	0.310	250.00	0.001 24	687
19	Data integration	0.720	250.00	0.002 88	1300
20	Instrumentation	5.584	200.00	0.027 92	1900
21	System management	4.640	2000.00	0.002 32	950
22	Life support	4.640	2000.00	0.002 32	950
23	Engine control	7.194	30.30	0.237 40	1500
24	Executive	0.400	200.00	0.002 00	1100

Recalculated from [23].

two real-time system designers. The same two designers took totally five weeks to design the same application a second time without using SESAG. For the second time, the designers were already quite familiar with the system design, so they were much faster in designing the application domain objects, but they had to spend a large amount of time designing the objects that were provided by SESAG, such as schedulers. The integration of application domain objects with self-designed SESAG objects also incurred a substantial amount of time during the second time of designing the avionics system.

SESAG evaluation

The avionics example was evaluated using the relative design effort as defined earlier in Definition 3. There were totally 10 application domain objects (five designed by each of two designers) and 35 application framework objects (20 and 15 objects, respectively, designed by the two designers). Thus, there are totally 45 objects in the final program code generated. The integration time using the

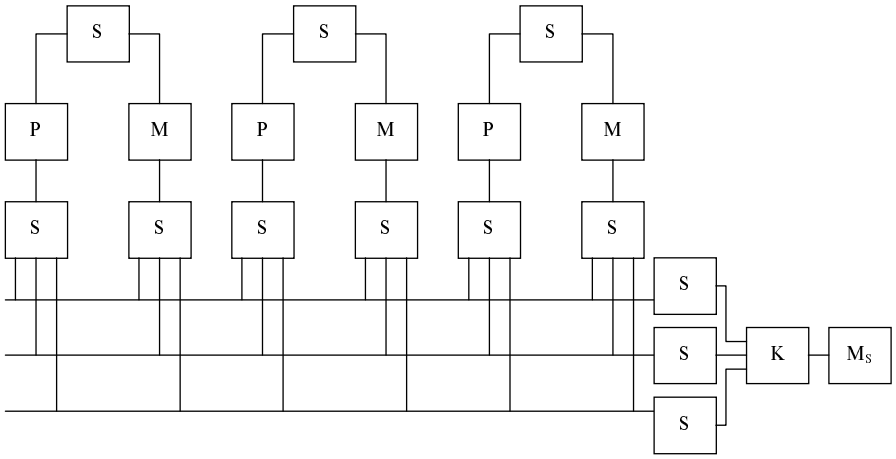

Figure 7. Avionics example: SIFT computer [23].

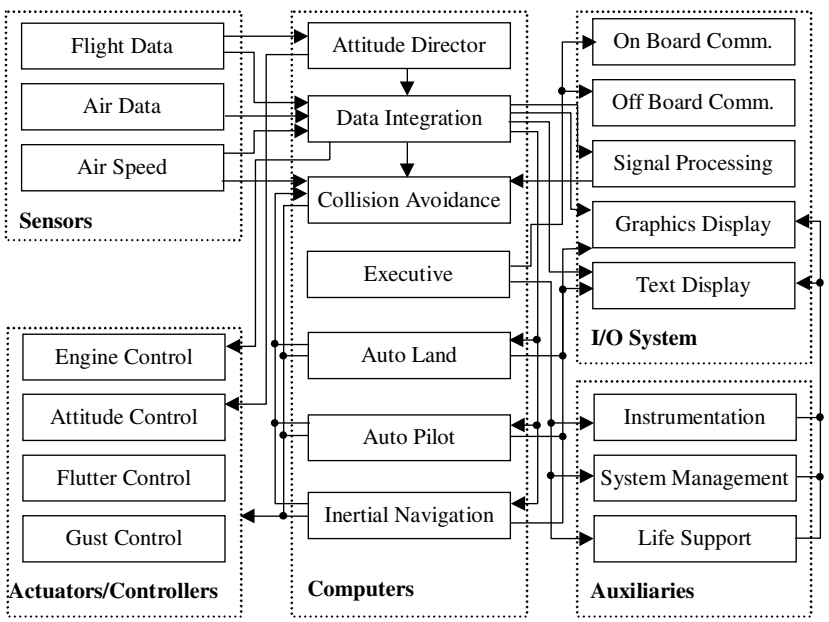

Figure 8. Avionics example: Call-Graph.

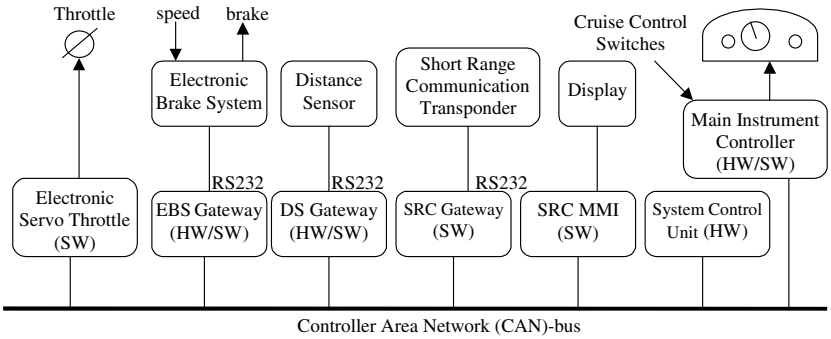

Figure 9. AICC example: System architecture.

framework was two days and that without using the framework was 10 days. The framework learning time is taken as one day for each designer amortized during the time span of the project. Using the RDE equation (1), we get the following result:

$$\text{RDE} = \frac{5}{5+20} \times \frac{7+2+1}{35+10} + \frac{5}{5+15} \times \frac{7+2+1}{35+10} = 0.09 \tag{2}$$

We observe that the relative design effort is only 9% of that required without using SESAG.

7.2. AICC cruiser example

Another illustrative example application developed using SESAG is the AICC system application [24], which had been developed and installed in a Saab automobile by Hansson *et al*. The AICC system can receive information from road signs and adapt the speed of the vehicle to automatically follow speed limits. Also, with a vehicle in front and cruising at lower speed, the AICC adapts the speed and maintains safe distance. The AICC can also receive information from the roadside (e.g. from traffic lights) to calculate a speed profile which will reduce emission by avoiding stop and go at traffic lights. The system architecture consisting of both hardware (HW) and software (SW) is as shown in Figure 9. The software development methodology used in [24] is based on sets of interconnected so-called *software circuits* (SCs). Each SC has a set of input connectors where data are received and a set of output connectors where data are produced. We model the SCs in [24] as *active application domain objects* in SESAG.

As shown in Figure 10, there are five domain objects specified by the designer of AICC for implementing a *basement* system. Basement is a vehicle's internal real-time architecture developed in the *Vehicle Internal Architecture* (VIA) project [24], within the *Swedish Road Transport Informatics Programme*. As observed in Figure 10, each object may correspond (map) to one or more tasks. `Task-Table` and `Call-Graph` instantiated by the Extractor component are as shown in Table II

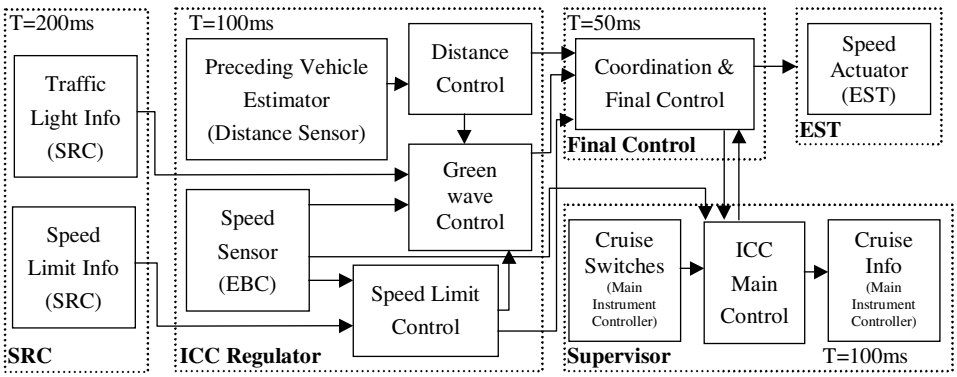

Figure 10. AICC example: Call-Graph.

Table II. AICC example: task table.

#	Task description	Object	Period (ms)	Execution time (ms)	Deadline (ms)
1	Traffic light info	SRC	200	10	400
2	Speed limit info	SRC	200	10	400
3	Proceeding vehicle estimator	ICCReg	100	8	100
4	Speed sensor	ICCReg	100	5	100
5	Distance control	ICCReg	100	15	100
6	Green wave control	ICCReg	100	15	100
7	Speed limit control	ICCReg	100	15	100
8	Coordination and final control	Final_Control	50	20	50
9	Cruise switches	Supervisor	100	15	100
10	ICC main control	Supervisor	100	20	100
11	Cruise info	Supervisor	100	20	100
12	Speed actuator	EST	50	5	50

SRC: Short Range Communication, ICCReg: ICC Regulator, EST: Electronic Servo Throttle.

and Figure 10, respectively. There are a total of 12 tasks performed in five objects. Two different resources were identified in SESAG, namely, SRC and Display. This application took five days for three real-time system designers using SESAG. The same application took the same designers 20 days to complete development a second time, without using SESAG. The large difference in design times was because SESAG automatically extracted the tasks and constraints from the object specifications.

SESAG evaluation

The AICC example was also evaluated using the RDE metric (Definition 3 and Equation (1)). There were five application domain objects specified by the designer and 21 application framework objects. The integration time using the framework was one day and that without using the framework was four days. The framework learning time is taken as one day for each designer, amortized over the time span of the project. In total, 26 objects were in the final program code generated. We have the following result:

$$\text{RDE} = \frac{2}{2+10} \times \frac{5+1+1}{20+4} + \frac{2}{2+5} \times \frac{5+1+1}{20+4} + \frac{1}{1+6} \times \frac{5+1+1}{20+4} = 0.174 \qquad (3)$$

We observe that for this application, we also obtain a relative design effort of only 17.4%.

It should be noted here, that the integration time for a designer without using SESAG in both of the above examples would have been more than that stated here if the application was developed for the first time, instead of for the second time as was the case given for the above examples. This is because the designers have already become familiar with the systems so the integration time was smaller.

From the above two examples, we observe that with the first avionics example being more complex, the design effort saved by using SESAG is greater than the second cruiser example. We see that for both the examples, using SESAG the design effort required is only approximately between 9 and 18% of that required without using SESAG. This is obtained using our newly proposed framework evaluation metric called RDE. Significant design reuse and increased design productivity justify the use of SESAG in real-time application development.

8. CONCLUSION

An OO application framework, called SESAG, was proposed for the development of real-time system applications that consists of a set of statically schedulable periodic tasks. Safety analysis was not covered as it is not within the scope of SESAG. The presentation included two different perspectives of SESAG: a components-patterns view and a class view. Several design patterns related to real-time system design were proposed and implemented in SESAG. The framework implementation and a new evaluation metric for OOAFs were also presented. Two application examples were developed using SESAG. Both the examples have shown how design time is significantly reduced due to a large extent of object and code reuse from SESAG. In addition to reuse, SESAG also automates several design phases of real-time system application development, including tasks extraction, constraints extraction, scheduling, and resource allocation. All of these design phases were very painstaking and laborious originally, when a designer had to develop either from scratch or using different specialized tools such as scheduling analysis tool, allocation tool, etc.

SESAG can be easily extended since new specification languages, scheduling algorithms, etc. can always and easily be integrated into it. Future extensions will support other scheduling algorithms. More examples will also be developed using SESAG. This basic version of SESAG will be enhanced in the future by considering more advanced features of real-time applications, such as network delay, network protocols, and on-line task scheduling. Performance related features such as context switch time and rate, external events handling, I/O timing, mode changes, transient overloading, and setup time will also be incorporated into SESAG in the future.

REFERENCES

1. Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery* 1973; **20**(1):46–61.
2. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
3. Johnson R, Foote B. Designing reusable classes. *Journal of Object-Oriented Programming* 1988; **1**(5):22–35.
4. Fayad M, Schmidt DC. Object-oriented application frameworks. *Communications of the ACM (Special Issue on Object-Oriented Application Frameworks)* 1997; **40**(10):32–38.
5. Attoui A, Schneider M. An object oriented model for parallel and reactive systems. *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society Press: Los Alamitos, CA, 1991; 84–93.
6. Hammer DK, Welch LR, van Roosmalen OS. A taxonomy for distributed object-oriented real-time systems. *ACM OOPS Messenger (Special Issue on Object-Oriented Real-Time Systems)* 1996; **7**(1):78–85.
7. Ishikawa Y, Tokuda H, Mercer CW. Object-oriented real-time language design: Constructs for timing constraints. *ECOOP/OOPSLA'90 Proceedings*, *ACM SIGPLAN Notices* 1990; **25**(10):289–298.
8. Achauer B. Objects in real-time systems: Issues for language implementors. *ACM OOPS Messenger* 1996; **7**(1):21–27.
9. Welch LR. A metrics-driven approach for utilizing concurrency in object-oriented real-time systems. *ACM OOPS Messenger* 1996; **7**(1):70–77.
10. Gergeleit M, Kaiser J, Streich H. Checking timing constraints in distributed object-oriented programs. *ACM OOPS Messenger (Special Issue on Object-Oriented Real-Time Systems)* 1996; **7**(1):51–58.
11. Browne JC. Object-oriented development of real-time systems: Verification of functionality and performance. *ACM OOPS Messenger (Special Issue on Object-Oriented Real-Time Systems)* 1996; **7**(1):59–62.
12. See WB, Chen SJ. Object-oriented real-time system framework. *Domain-Specific Application Frameworks*, Fayad ME, Johnson RE (eds.). Wiley: New York, 2000; 327–338.
13. See WB, Chen SJ. High-level reuse in the design of an object-oriented real-time system framework. *Proceedings of the International Conference on Distributed Systems, Software Engineering, and Database Systems, ICS'96*, Kaohsiung, Taiwan, ROC, 1996; 363–370.
14. Kuan TY, See WB, Chen SJ. An object-oriented real-time framework and development environment. *Proceedings of the OOPSLA'95 Workshop #18*, *ACM OOPS Messenger* 1995; **6**(4):207.
15. Schmidt DC. Applying design patterns and frameworks to develop object-oriented communication software. *Handbook of Programming Languages*, vol. I, Salus P (ed.). Macmillan Computer Publishing, 1997.
16. Rumbaugh J, Booch G, Jacobson I. *The UML Reference Guide*. Addison-Wesley: Reading, MA, 1999.
17. Tokuda H, Mercer CW. ARTS: A distributed real-time kernel. *Operating Systems Review* 1989; **23**(3):29–53.
18. Kligerman E, Stoyenko AD. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering* 1986; **SE-12**(9):941–949.
19. Grimshaw AS, Silberman A, Liu JWS. Real-time Mentat programming language and architecture. *Proceedings of the IEEE Globecom*. IEEE Computer Society Press: Los Alamitos, CA, 1989; 141–147.
20. Lin KJ, Natarajan S. Expressing and maintaining timing constraints in flex. *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society Press: Los Alamitos, CA, 1988; 96–105.
21. Sha L, Rajkumar R, Lehoczky JP. Priority inheritance protocols: an approach to real-time synchronization. *Technical Report CMU-CS-87-181*, Computer Science Department, Carnegie Mellon University, 1987.
22. Douglass BP. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley: Reading, MA, 1999.
23. Bannister JA, Trivedi KS. Task allocation in fault-tolerant distributed systems. *Acta Informatica* 1983; **20**(3):261–281.
24. Hansson HA, Lawson HW, Stromberg M, Larsson S. BASEMENT: A distributed real-time architecture for vehicle applications. *Real-Time Systems* 1996; **11**(3):223–244.
25. Ratner RS, Shapiro EB, Zeidler HM, Wahlstrom SE, Clark CB, Goldberg J. Design of a fault-tolerant airborne digital computer, vol. 2, computational requirements and technology, *SRI Final Report, NASA Contract NAS1-10920*, 1973.
26. Potkonjak M, Wolf W. A methodology and algorithms for the design of hard real-time multi-tasking ASICs. *ACM Transactions on Design Automation of Electronic Systems* 1999; **4**(4):430–459.

