# Hierarchical Approach for Design of Multi-vehicle Multi-modal Embedded Software

T. John Koo, Judy Liebman, Cedric Ma, and S. Shankar Sastry

Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley
Berkeley, CA 94704,
koo,judithl,cedricma,sastry@eecs.berkeley.edu,

**Abstract.** Embedded systems composed of hardware and software components are designed to interact with a physical environment in real-time in order to fulfill control objectives and system specifications. In this paper, we address the complex design challenges in embedded software by focusing on predictive and systematic hierarchical design methodologies which promote system verification and validation. First, we advocate a mix of top-down, hierarchical design and bottom-up, component-based design for complex control systems. Second, it is our point of view that at the level closest to the environment under control, the embedded software needs to be time-triggered for guaranteed safety; at the higher levels, we advocate an asynchronous hybrid controller design. We briefly illustrate our approach through an embedded software design for the control of a group of autonomous vehicles.

## 1 Introduction

Embedded software is designed to process information to and fro between the information and physical worlds. Advances in software, computation, communication, sensing and actuation have enabled the rapid realization of high-performance and sophisticated multi-vehicle systems. The rapidly growing demand for high-confidence embedded software that is required in order to control new generations of autonomous vehicles for collectively delivering high levels of mission reliability, is putting tremendous pressure on control and software designers in industry.

A high-confidence system should have the following characteristics: correctness by construction and fault-tolerance, and resistance to information attack. In fact, the cost of system development is mainly due to prohibitively expensive embedded software integration and testing techniques that rely almost exclusively on exhaustively testing of more or less complete versions of complex systems. Formal methods are hence introduced for the synthesis and verification of embedded software in order to guarantee that the system is correct by construction with respect to a given set of specifications. The system design should also be fault-tolerant so that it can handle all anticipated faults that might occur in the system and possibly recover from them after the faults have been detected and

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 344–360, 2001.

identified. Furthermore, the design should also ensure that the system is not vulnerable to attack from the information world by taking into account models of attack. While this last item is important we do not address it in this paper since it would require a lengthy discussion of models of attack of embedded systems.

In this paper, we address this bottleneck by focusing on systematic hierarchical design methodologies. We briefly illustrate our approach through an embedded software design for the control of a group of autonomous vehicles. Consider that the task consists of flying a group of autonomous vehicles in a prespecified formation. We assume that each vehicle is equipped with the necessary sensing, communication, and computation capabilities in order to perform a set of predetermined tasks. The control of the multi-vehicle systems can be organized as a distributed hierarchical system. The meaning of a distributed system refers to a system comprised of several subsystems which are spatially distributed. Large-scale systems ranging from automated highway systems (AHS) [1], air traffic management systems (ATMS) [2], and power distribution networks are typical examples of distributed systems. However, large-scale systems are systems of very high complexity. Complexity is typically reduced by imposing a hierarchical structure on the system architecture. In a such a structure, systems of higher functionality reside at higher levels of the hierarchy and are therefore unaware of lower-level details. A component-based design provides a clean way to integrate different models by hierarchical nesting of parallel and serial composition of heterogeneous components. This hierarchical composition also allows one to manage the complexity of a design by information hiding and by reusing components.

To cope with these complex design challenges, we advocate a mix of top-down, hierarchical design and bottom-up, component-based design for complex control systems. Hierarchical design begins with the choice and evaluation of an overall distributed, multi-layered system architecture; component-based design begins with the derivation of robust mathematical control laws and decision-making procedures.

Hybrid systems, in general, are defined as systems built from atomic discrete and continuous components by parallel and serial composition, which is arbitrarily nested. Hybrid systems refers to the distinguishing fundamental characteristics of embedded control systems, namely, the tight coupling and interaction of discrete with continuous phenomena. The coupling is inherent to embedded systems since every digital software/hardware implementation of a control design is ultimately a discrete approximation that interacts through sensors and actuators with a continuous physical environment. There has been a large and growing body of work on formal methods for hybrid systems: mathematical logics, computational models and methods, and automated reasoning tools supporting the formal specification and verification of performance requirements for hybrid systems, and the design and synthesis of control programs for hybrid systems that are provably correct with respect to formal specifications.

Depending on the level of abstraction and domain-specificity of design practice, different models of computation (MOCs) [6] which govern the interaction

among components can be chosen and mixed for composing hybrid systems. There are a rich variety of models of computation that deal with concurrency and time in different ways. As mentioned in [7], MOCs such as continuous-time (CT), synchronous data flow (SDF), finite-state machine (FSM), synchronous/ reactive (SR) [8,9,10,11], discrete-event (DE), and time triggered (TT) [13,14] are useful for the design of embedded systems.

Hybrid systems have been used in solving synthesis and verification problems of some high-profile and safety-critical applications such as conflict resolution [27] for multi-vehicle platforms, and multi-modal control [20] and envelope protection [17] for single vehicle platforms. In the above words, the hybrid system is a composition of CT and FSM. Any transition in system discrete states can occur only when the transition guard, which is a function of discrete/continuous states and inputs, becomes true. The continuous state in a discrete location evolves continuously according to differential equations, as long as the location's invariant remains true. Each transition is asynchronous since it can happen any time as long as the guard condition is satisfied.

Embedded systems composed of hardware and software components are designed to interact with a physical environment in real-time in order to fulfill control objectives and system specifications. In the real-time computing literature, the word *time* means that the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced. In the implementation of control laws using digital computers, the necessary computational activities for the implementation of control laws are decomposed and translated into periodic tasks. Hence, there is a *hard* deadline in time for the task to meet in each period of execution. Failure in meeting the deadline may cause catastrophic consequences on the environment under control. The word *real* indicates that the reaction of the systems to external events must occur during their evolution. Notice that the notion of time is not an intrinsic property of a control system but it is strictly related to the environment under control. For example, the design of a real-time embedded system for flight control has to take the timing characteristics of a vehicle into consideration.

It is our point of view that at the level closest to the physical systems under control, the embedded software needs to be *time-triggered* for guaranteed safety. At higher levels, we advocate *asynchronous* decision-making procedures. In Section 2, a hierarchical architecture which allows modular verification will be introduced for the construction of embedded systems. Next, in Section 3, we will show the state-of-the-art in verification and synthesis of control laws and decision-making algorithms based on formal methods. Motivation and implementation of the control design in embedded software using a time-triggered framework will be presented in Section 4. Finally, we conclude our work in Section 5.

## 2 Hierarchical Architecture for Multi-vehicle Multi-modal Systems

Imposing a hierarchical structure on the system architecture has been used for solving the control problem of large-scale systems. A desired hierarchical structure should not only provide manageable complexity but also promote verification. There are several approaches to understanding a hierarchy depending on the design perspective. In particular, two distinct approaches have been shown in [3] for the design and analysis of AHS [1]. One approach to the meaning of hierarchy is to adopt *one-world* semantics, and the other approach is referred to as *multi-world* semantics.

In one-world semantics for hierarchical systems, a higher-level expression is interpreted in a process called *semantic flattening*: the expression is first compiled into lower-level expression and then interpreted. In other words, an interpretation at each level is semantically complied into a single interpretation at the lowest-level in the *imperative* world. Furthermore, semantic flattening implies that checking any high-level truth-claim can be performed by an automatic procedure if there is a facility for automatically verifying the lowest-level interpretation. This approach provides a unique interpretation to system description and verification. The main drawback to one-world semantics is that higher-level syntax and truth-claims have to be reformulated if there are some changes at any of the lower levels. On the other hand, in multi-world semantics for hierarchical systems, an expression at each level is interpreted at the same level. Therefore, checking the truth-claim at that level is performed in its own *declarative* world. Moreover, this approach conforms with common system design practice. However, relating these separate worlds together is a nontrivial task.

To take advantage of the two approaches, a decision structure has been suggested in [3] for providing connections between the multiple world of declarative semantics and the single world of imperative semantics. The construction of the structure is based on the following ideas: 1. Ideal compilation - a higher-level expression is complied into an idealized lower-level expression and then interpreted; 2. Usage of Invariants - higher-level truth-claims are conditional lower-level truth-claims; 3. Modal decomposition - Splitting multi-world semantics into multiple-frameworks, each dealing with identified faults, should be done if one-world interpretation leads to falsification of higher-level claims that are true in multi-world semantics.

Ideal compilation and usage of invariants are enablers for performing formal verification of the truth-claim at each level of hierarchy. Modal decomposition suggests that the system should be modeled as a multi-modal system which has multiple modes of operation. *Abstraction* is a key concept for providing ways to connect disjointed worlds together. In [4,5], the notions of abstraction, or aggregation, refer to grouping the system states into equivalence classes. Depending on the cardinality of the resulting quotient space, there may be discrete or continuous abstractions. Truth-claims made at each world are then based on a consistent notion of abstraction in order to have a well-defined definition. Furthermore, at different levels of abstraction, the definitions of environment could

also be different. In general, the environment is defined as the entities which could not be designed.

Here, we describe the construction of a hierarchical architecture which keeps the advantages of the multiple world of declarative semantics and the single world of imperative semantics. Consider the control laws and decision-making procedures as the basic components for the construction of the hierarchical system for controlling the environment. Any changes in the scenario are triggered by a fault in the lower level. Each fault, defined in each world, triggers a change in scenario. The construction is formalized by using component-based design approach.

A component-based design provides a clean way to integrate different models by hierarchical nesting of parallel and serial composition of heterogeneous components. This hierarchical composition also allows one to manage the complexity of a design by information hiding and reusing components. In addition to the syntactic issues mentioned in above, for interpreting the syntax, semantic issues have to be addressed in the design. A semantics gives meaning to components and their interconnections. A collection of semantics models which are useful for embedded software design have been codified in [6] as *models of computation* (MOCs). Thus, in the design, the selection of a MOC governing interactions between components depends on whether its properties match the application domain.

CT models represented by differential equations are excellent for modeling physical systems. Execution in FSM, which is a strictly ordered sequence of state transitions that are not concurrent, and FSM models, are amenable to in-depth formal analysis. In DE models of computation, an event consists of a value and time stamp. There is no global clock tick in DE, but there is a globally consistent notion of time. This model has been realized in a large number of simulation environments. In TT, systems with timed events are driven by clocks, so that signals with events are repeated indefinitely with a fixed period. Time-triggered architecture (TTA) [13] is a hardware architecture supporting this highly regular style of computation. The Giotto programming language [14] provides a time-triggered software abstraction which, unlike the TTA, is hardware independent. It is intended for embedded software systems where periodic events dominate.

As defined in [7], a model is a hierarchical aggregation of components, which are called *actors*. Actors encapsulate an atomic execution and provide communication interfaces to other actors. An actor can be *atomic* or *composite*. A composite actor can not only contain other actors but can also be contained by another composite actor. Hence, hierarchies can be arbitrarily nested. A MOC associated with a composite actor is implemented as a *domain*. A domain defines the communication semantics and the execution order among actors. Consider the multi-modal system [12] for single vehicle control depicted in Figure 1. It is constructed by hierarchically nesting parallel and serial compositions of heterogeneous components. There are two levels of hierarchy introduced. At the first level, in the CT domain, the multi-modal controller is modeled as a composite component and the vehicle dynamics is modeled as ordinary differential

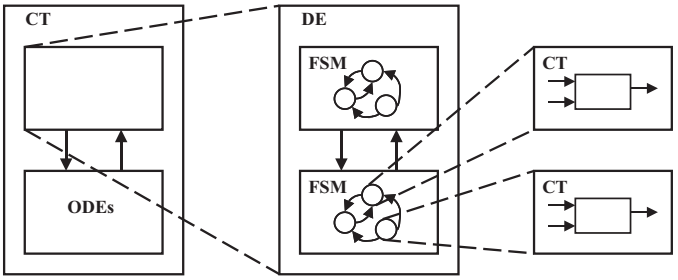


**Fig. 1.** A multi-modal control system

equations (ODEs). The multi-modal controller is hierarchically refined by two components in the DE domain due to the asynchronous interaction between them. At the second level, the component on the top, models the mode switching logic and the other component, on the bottom, models the control switches by FSM. In each discrete state of the FSM, the component is further refined to model the continuous-time control laws. Notice that domain DE is chosen due to the asynchronous behaviors exhibited by the models. The model assumes a fault-free scenario.

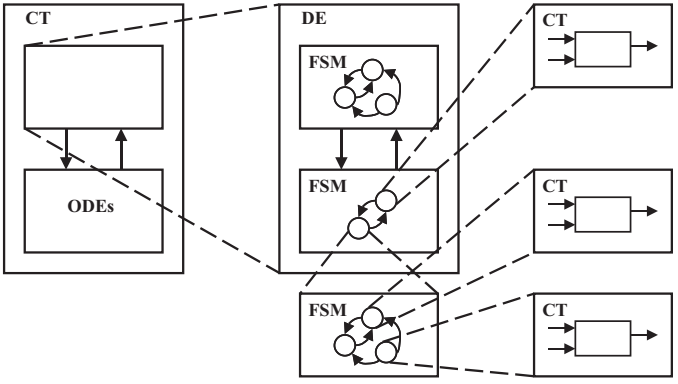


**Fig. 2.** A multi-modal control system with fault handling

In order to handle anticipated faults such as failures in sensors and actuators or even changes in dynamics, the single world has to be split into two modes of operation in order to handle different scenarios: a non-faulty mode and a faulty mode. The switching between framework is caused by two events generated after fault detection and fault recovery. Therefore, FSM domain is introduced for handling the execution. As shown in Figure 2, a FSM is introduced and there

are two discrete states representing the two modes. In the non-faulty mode, the component is refined by a FSM for modeling control switches, and then further refined by continuous-time control laws. While in the fault mode, the component is refined by a continuous-time controller for handling the anticipated fault.

In order to to reconcile different world views, we propose a hierarchical architecture for the control of multi-vehicle multi-modal systems. It is shown in 3. On the lowest level, the dynamics of the vehicles are combined and modeled by ODEs and the composite components represent the controllers for the vehicles. Consider the hierarchy inside the controller for vehicle 1. At each level of hierarchy, there is a non-fault mode and fault modes. The operation of the non-fault mode assumes that no fault happened in the lower level. The transitions among modes are governed by FSM. Then, the component of each discrete state of FSM can further be refined by the control laws. DE is chosen for the domain to govern the asynchronous communications between levels. In DE domain, although the execution is asynchronous, there is still a globally consistent notion of clock. Similarly, one can construct more levels by further defining and refining the composite components.

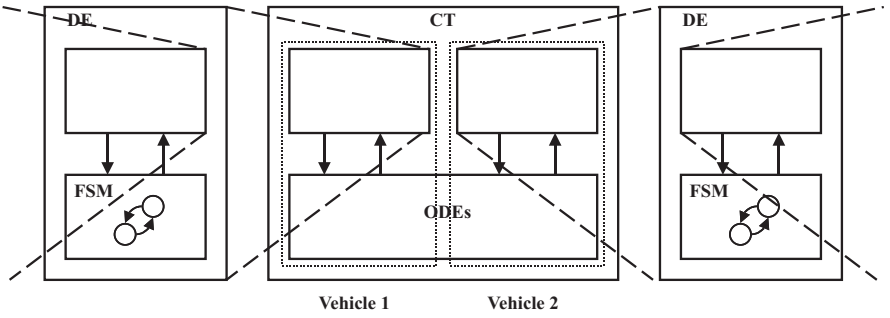


**Fig. 3.** Proposed hierarchical architecture for multi-vehicle multi-modal systems

## 3 Design of Multi-vehicle Multi-modal System Components

A hierarchical architecture for multi-vehicle multi-modal systems has been proposed in the previous section. In this section, the components with which the system is composed are presented. The task assigned to the system is decomposed and then executed by the components, according to the system architecture. Each component has to be designed so that it provides a guarantee on its performance under certain assumptions. Consequently the truth-claims of the whole system can be inferred. Formal methods for the synthesis and verification of the components are presented.

Reconsider that a task for a group of autonomous vehicles is to keep flying in a prespecified formation. In the group, one vehicle is assigned to be the leader of the group. The leader is responsible for making decisions on the route and formation pattern for the group. In a bigger picture, leaders from different groups can exchange information and negotiate with other group leaders via a global communication network to perform a mission collectively, for example a pursuit-evasion game [32,33]. Among group members, assume that there exits a local communication network for distributing information for performing the task.

According to the mission along with the information and assumptions made about current environment, the leader has to compute the best possible route and formation for the group with the maneuverability of the vehicles taken into consideration. The set of maneuvers by which a vehicle can perform depends on the available controllers being designed and coded in the embedded software.

As in most of the designs of large-scale systems for autonomous vehicle control, the task is decomposed into four subtasks which are responsible for the control of the vehicle, control mode switching, maneuver sequence generation and route selection. Furthermore, possible faults in each subtask due to the violation of the underlying assumptions of the design and the corresponding fault-handling procedures are presented.

### 3.1 Control Law Synthesis

Modern control design methodologies [15,16] have enable the synthesis of robust and high-performance control systems. Consider each vehicle dynamics modeled by differential equations of the form

$$\dot{x}(t) = f(x(t)) + g(x(t))u(t), \quad x(t_0) = x_0, \quad t \geq t_0 \tag{3.1}$$

where $x \in \mathbb{R}^n$, $u \in \mathbb{R}^p$, $f(x) : \mathbb{R}^n \to \mathbb{R}^n$ and $g(x) : \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^p$. The system is assumed to be as smooth as needed. The controllers are synthesized with a given set of performance specifications and design criteria with assumptions on the vehicle dynamics, sensor and actuator models which describe sensor noise, actuator dynamics and anticipated structured disturbance. Therefore, the validity of the claim about the control system, at a higher-level, depends on the validity of the assumptions made on the vehicle dynamics, sensor, and actuator models at a lower-level. However, in the presence of sensor and actuator faults or unanticipated disturbance, in order to keep the vehicle in safe operating condition, least restrictive control is introduced in [17] by keeping the vehicle states within the flight envelope so that no catastrophic consequences can happen. The controllers *guarantee* the safe operation of the vehicle under multiple anticipated fault scenarios. In each scenario, game theoretic approach is used to synthesize the so-called *least restrictive control laws*, and the *maximal safe set* for keeping the vehicle state safe is obtained by solving the corresponding Hamilton-Jacobi equation. The Hamilton-Jacobi equation is a partial differential equation of the form

$$\frac{\partial J}{\partial t}(x,t) = -H^*(x, \frac{\partial J}{\partial x}(x,t)) \tag{3.2}$$

where $H^*$ is the so-called Hamiltonian determined from the appropriate sets and the game between the controllable and uncontrollable actions. A computational tool is designed by [18] for computing the reachable set based on a level set technique developed by [19], which computes the viscosity solution to the Hamilton-Jacobi equation, ensuring that discontinuities are preserved.

### 3.2 Control Mode Switching Synthesis

Given a specific set of controllers of satisfactory performance, a variety of high-level tasks can be accomplished by appropriately switching between low-level controllers. Here, we define a control mode as the operation of the system under a controller that is *guaranteed* to track a certain class of output trajectories. Formally, we have:

*A control mode, labeled by $q_i$ where $i \in \{1, \ldots, N\}$, is the operation of the nonlinear system (3.1) under a closed-loop feedback controller of the form*

$$u(t) = k_i(x(t), r_i(t)) \tag{3.3}$$

*associated with an output $y_i(t) = h_i(x(t))$ such that $y_i(t)$ shall track $r_i(t)$ where $y_i(t), r_i(t) \in \mathbb{R}^{m_i}$, $h_i : \mathbb{R}^n \to \mathbb{R}^{m_i}$, $k_i : \mathbb{R}^n \times \mathbb{R}^{m_i} \to \mathbb{R}^p$ for each $i \in \{1, \ldots, N\}$. We assume that $r_i \in \mathcal{R}_i$, the class of output trajectories associated with the control mode $q_i$, when the initial condition of the system (3.1) starts in the set $S_i(r_i) \subseteq X_i$, output tracking is guaranteed and the state satisfies a set of state constraints $X_i \subseteq \mathbb{R}^n$.*

Consider a reachability task as reaching a desired final control mode from an initial control mode. For this reachability task, [20] has derived a formal method for the synthesis of *control mode graph*. Therefore, if there exists at least one path between an initial control mode to a desired final control mode on the control mode graph, the reachability problem is solvable with a finite number of switching of control modes and, furthermore, the switching conditions can be derived. Switching of controllers [17] with multiple objectives can also be formulated within the same framework, and the partial orders on objectives can be used to bias the search for feasible solutions. The approach consists of extracting a finite graph which refines the original collections of control modes, but is consistent with the physical system. Thus, proof of "control mode switching problem is solvable" is conducted in the semantics of directed graphs. Computational tools based on hybrid system theory have been developed for computing exactly, or approximately reachable sets [17,21,22,23,24] for solving the reachability problem.

However, the design is valid under the free-space assumption in the proximity of the vehicle. Failure in sensing the rapid unanticipated changes in the environment may lead to the vehicle colliding with the environment. Since avoiding obstacles can be formulated as a reachability problem, obstacle avoidance algorithms should be designed to reuse the existing low-level controllers.

### 3.3 Maneuver Sequence Synthesis

With the availability of a local communication network, vehicles can communicate with each other to perform tasks together. Reconsider the problem of formation. As shown in [25], with consideration on different levels of centralization for formation, one can determine differential geometric conditions that *guarantee* formation feasibility given the individual vehicle kinematics. Information requirements for keeping a formation while maintaining mesh stability, *i.e.* any transient errors dampen out as they travel away from the source of errors within the formation, has been studied in [26].

While there is a communication failure within the network, the group vehicles may not be able to maintain *minimum separation* with each other. Any conflict resolution algorithm must use available, probably local, information to generate maneuvers that resolve conflicts as they are predicted. In [27], given a set of flight modes which are defined as an abstraction of the control modes by considering the kinematic motion of the closed-loop system, the conflict resolution could synthesize the parameters of the maneuver, *i.e.* a sequence of flight modes, and the condition for switching between them. The result would be a maneuver that is proven to be safe within the limits of the model used. The algorithm is based on game-theoretic methods and optimal control. The solution is obtained by performing a hybrid game that is a multi-player structure in which the players have both discrete and continuous moves. Hybrid games have been classified with respect to the complexity of the laws that govern the evolution of the variables and with respect to the winning conditions for the players. This has been studied in the timed game [28,29] for constant differential equations of the form $\dot{x} = c$, the rectangular games [30,31] for rectangular differential inclusions of the form $c_i \leq \dot{x}_i \leq d_i$ for $i \in \{1, \ldots, n\}$, and the nonlinear hybrid games [27] for nonlinear differential equations with inputs.

Regardless of the availability of communication, the formations and the maneuvers should be generated with the consideration of anticipated, static obstacles in the environments.

### 3.4 Routing Sequence Verification

Given a discretized map which can be represented as a directed graph under the assumption made on its maneuverability, the existence of a route for going from an initial node to a desired final node can be verified by using discrete reachability computation. This is important for performing tasks such as pursuit-evasion games [32,33] over graphs. Again, the validity of the claim depends upon the compiled ideal-typical behavior at the lowest-level.

## 4 Embedded Software Design

Given a system architecture and the components for composing the system, the high-level truth-claims rely heavily on the ideal-typical behaviors of the lower

levels. For this reason the *correct* execution of the control laws at the lowest-level must be enforced to avoid any catastrophic consequences from occurring. In this section we discuss the embedded software that is close to the environment under control.

Control laws are decomposed and translated into sets of *periodic* computational tasks in order to be implemented in embedded software. Along with functionality specifications, timing specifications are also given for the execution of the tasks in order to ensure that the implementation is consistent with the design. There are precedence relations not only defined among computational tasks but also specified for the logical ordering of event executions: sensing, control, and actuation.

In such an embedded system, processors, memory, inputs/outputs (I/O), sensors, and actuators are considered hardware components. We assume that the hardware components and some software components, such as the operating system (OS) and I/O drivers are given and are not parts of the design. Hence, the embedded software that is referred to here is made up of software components which are supported by the services provided by the OS. Therefore, the embedded software layer must work within the timing and scheduling framework of the underlying OS.

Another given in the system is that the basic design for an embedded controller includes a reliance on I/O and I/O rates in order to run any computational tasks. The system as a whole is then assumed to be driven by various local clocks and data processes that may occur at differing rates. Hardware components, such as sensors and actuators, exchange data with the embedded software via the I/O handled by the OS.

Operating systems, including real-time operating systems (RTOS), are designed under the same basic assumptions made in timesharing systems, where tasks are considered as unknown asynchronous activities activated at random instants. Except for the priority, no other parameters are provided to the system. The problem of scheduling a set of independent and preemptable periodic tasks has been solved under fixed and dynamic priority assignments [34], using the Rate-Monotonic algorithm and Earliest Deadline First algorithm. Given the precedence constraints and the Worst Case Execution Time (WCET), an efficient solution is provided by [35] for classes of the static scheduling problem. However, in order to implement the embedded software correctly, domain expertise is required for solving the scheduling problems on these heterogeneous computing platforms.

In the following, we discuss an embedded software example where the above configuration is utilized, and where the expected timing and ordering issues arise. We then continue and suggest adding a middle layer to the embedded software setup that eliminates these problems with, and dependencies on, any particular operating system. Consider a typical system configuration of a flight control system as depicted in Figure 4. This is motivated by our design experience on the development of embedded software for our helicopter-based unmanned aerial vehicles developed at the University of California, Berkeley. In the sys-

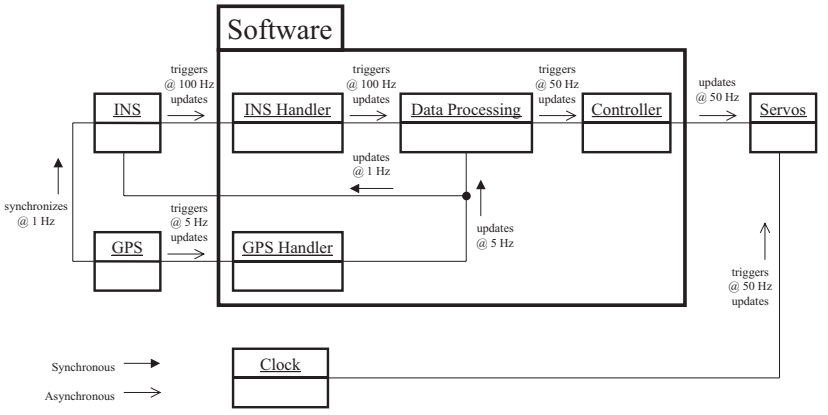


**Fig. 4.** Current configuration of system components is illustrated in Component Diagram. Communication between components via asynchronous and synchronous methods are shown. Notice that the software components are highlighted by grouping them together.

tem, there are hardware and software components running concurrently. Notice that the sensors, namely the Inertial Navigation System (INS) and the Global Positioning System (GPS), push the data asynchronously to separate software components. Because the INS and GPS each have their own internal clock, it is difficult to predict the times at which each of these two software processes will execute. Not only do these components have their own clocks, but they also must run at disparate rates as dictated by the design: the GPS at 5Hz, the INS at 100Hz, and the Servos at 50Hz. The rate of the Servos dictates that the control laws have to be applied at the same rate. The combination of the differing rates and the inherent variations of the clock edges creates random lags in the overall system. Furthermore, the execution of most of the software components relies on the timing of other software components. For example, the Data Processing task waits until it is triggered with data sent from the INS Handler, and the Controller waits until it is triggered by the Data Processing task. Due to the distinct clocks and rates of different software components, the interdependency of components allows for drastic uncertainty of the deadlines for each task. Consider that the Rate-Monotonic scheduling scheme is used for assigning priorities among tasks. Since the scheduling scheme does not take into account these dependencies between tasks, no guarantee can be made on the execution of the software components. Consequently, jitter may occur and it affects the regularity in time for computing the control results.

In order to mitigate these undesirable conditions with which the embedded software must work, we suggest reorganizing the system to include a middle layer of software, middleware, that establishes a time-triggered architecture. The middleware should provide a suitable level of abstraction for specifying functionality

and timing requirements of periodic tasks and ensure the correct execution of the tasks in the lower level provided that the scheduling problem is feasible to be solved. Next, in order to eradicate the problem of RTOS scheduling,

A middleware, named Giotto, is being developed by [14] in order to provide a time-triggered software abstraction for the execution of the embedded control software to meet hard real-time deadlines. Giotto, like other middlewares, is designed to provide the isolation between the embedded control software and the underlying computing platforms. In the example, the embedded control software refers to the software components, the Data Processing and the Controller. Hence, the scheduling of processing recourses and I/O handling is handled by the middleware. This modeling language is entirely time-triggered, and does not allow for any event triggered occurrences.

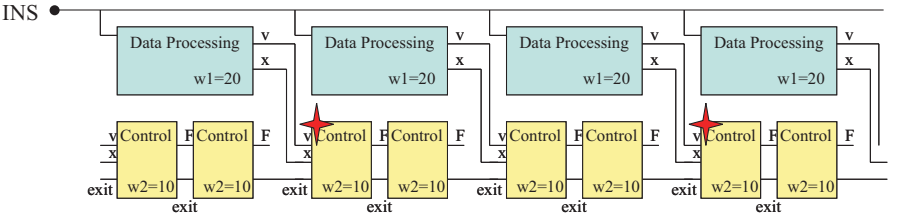


**Fig. 5.** The time line for the invocation of tasks in Giotto.

However, several changes are necessary to implement on the embedded system in order to allow for the usage of the time-triggered middleware. The most drastic change is that the entire system must now be comprised of synchronous elements in both the hardware and the software. For this effect, we propose two main changes in the hardware configuration. First, several local clocks in different components have to be collapsed into one global clock to avoid any phase differences. Second, due to the fact that the sensors are the basis of the current event driven paradigm, the sensors have to be reconfigured into a pull configuration so that the sensors would be synchronously queried for data by the software components. These two changes would translate the hardware system from an event-driven domain into a time-triggered one.

The software components must be slightly modified to fit into the Giotto language semantics. Since the embedded software is originally made up of periodic tasks that run concurrently and have desired end times associated with them, fitting the system into the framework of Giotto is relatively intuitive. The only complication with this change in the software specifications is the introduction of an increased, though bounded, delay. This delay is a function of the time-triggered paradigm. Instead of acting on data immediately, as in the event-driven scenario, a time-triggered architecture acts on data at a set time and, in this way, introduces a stale data phenomenon. In order to curtail the added

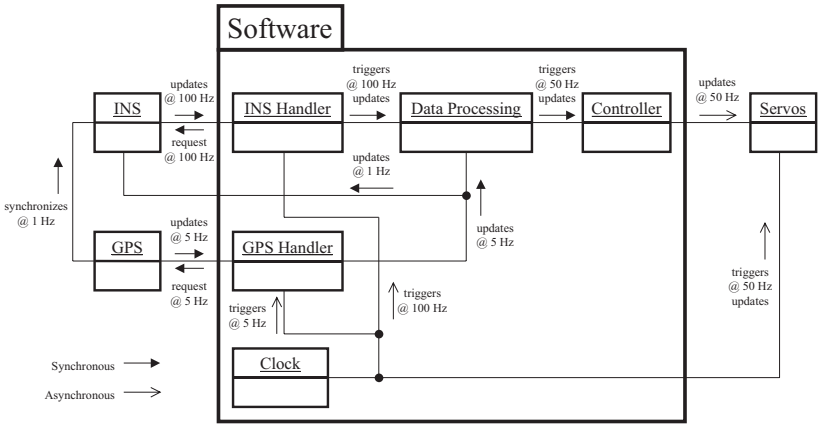


**Fig. 6.** Proposed configuration of system components is illustrated in Component Diagram. Synchronous communication methods are used between components, except the clocks, for communication.

delay in our modified example, the deadline times for the control computation task are shortened. Since shortening the execution time allowed for a block necessarily bounds the delay to the length of the block, the designer can regulate the added delay in this manner. Figure 5 displays the embedded software design under Giotto of the INS handler and the controller blocks. In order to bound the delay in data usage without computing the control more than is necessary, the control block deadline time is shortened but the control is only computed at the rate necessary (on the blocks marked with a star). In this manner, the software can be restructured to make use of the benefits of a time triggered middle layer with a minimum of added bounded delay.

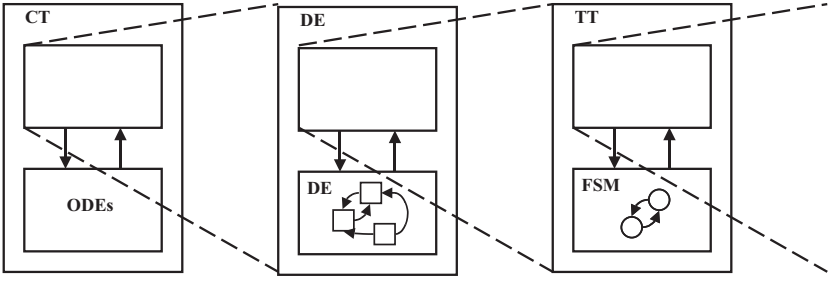


**Fig. 7.** Proposed embedded system hierarchy for a single vehicle

The resulting *synchronous* hybrid system as shown in Figure 6does have a predictable behavior and therefore is able to guarantee safety critical control performance. The drawback to this approach is that there is an increase in delay over the original unpredictable system. However, as long as the timing deadlines set are within the tolerances of the control design, the performance of the overall system can be guaranteed. Since Giotto also supports multi-modal behavior, the hierarchy of the system for a single vehicle can be further constructed and the idea of the design is depicted in Figure 7. The hardware and software components below the level of Giotto, labeled as TT domain, are represented by a single component in DE domain to model the asynchronous behaviors of the component.

## 5 Conclusion

In this paper, we have considered the design problem of embedded software for multi-vehicle multi-modal systems. A hierarchical architecture which promotes verification is presented for the construction of embedded systems. Hybrid control design techniques are an important design tool for rapid prototyping of system components for real-time embedded systems. Motivated by our design experience on the development of embedded software for our helicopter-based unmanned aerial vehicles which are composed of heterogeneous components, we believe that at the level closest to the environment under control, the embedded software needs to be time-triggered for guaranteed safety; at the higher levels, we advocate a asynchronous hybrid controller design. Current work focuses on the realization of the system for hardware-in-the-loop (HIL) simulation. This is especially challenging since it will demand the development of formal methodologies for the integration of multiple MOCs and for the analysis of the resultant hybrid system.

**Acknowledgments.** The authors would like to thank Benjamin Horowitz, Jie Liu, Xiaojun Liu, Hyunchul Shim, and Ron Tal for stimulating discussions and valuable comments. This work is supported by the DARPA SEC grant, F33615-98-C-3614.

## References

1. P. Varaiya. Smart Cars on Smart Roads: Problems of Control, *IEEE Transactions on Automatic Control*, 38(2):195-207, February 1993.
2. C. Tomlin, G. Pappas, J. Lygeros, D. Godbole, and S. Sastry. Hybrid Control Models of Next Generation Air Traffic Management, *Hybrid Systems IV*, volume 1273 of Lecture Notes in Computer Science, pages 378-404, Springer Verlag, Berlin, Germany, 1997.
3. P. Varaiya. A Question About Hierarchical Systems, *System Theory: Modeling, Analysis and Control*, T. Djaferis and I. Schick (eds), Kluwer, 2000.
4. P. Caines, and Y. J. Wei, Hierarchical Hybrid Control Systems : A lattice theoretic formulation, *IEEE Transactions on Automatic Control*, 43(4), 1998.

5. G. J. Pappas, G. Lafferriere, and S. Sastry. Hierarchically Consistent Control Systems, *IEEE Transactions on Automatic Control*, 45(6):1144-1160, June 2000.
6. E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217-1229, December 1998.
7. E. A. Lee. Overview of the Ptolemy Project, *Technical Memorandum UCB/ERL*, M01/11, University of California, Berkeley, March 2001.
8. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation, *Science of Computer Programming*, 19(2):87-152, 1992.
9. A. Benveniste and P. Le Guernic. Hybrid Dynamical Systems Theory and the SIGNAL Language, *IEEE Transactions on Automatic Control* 35(5):525-546, May 1990.
10. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Programming Synchronous Systems, *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January 1987.
11. F. Maraninchi. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems, in *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.
12. J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. S. Sastry, and E. A. Lee. Hierarchical Hybrid System Simulation. In *Proceedings of the 38th Conference on Decision and Control*, Phoenix, Arizona. December 1999.
13. H. Kopetz. The Time-Triggered Architecture, in *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing*, Kyoto, Japan, April 1998.
14. T.A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded Control Systems Development with Giotto, in *Proceedings of LCTES 2001*, Snowbird, Utach, June 2001.
15. A. Isidori. Nonlinear Control Systems, Springer-Verlag, New York, 1995.
16. S. S. Sastry. Nonlinear Systems: Analysis, Stability, and Control, Springer-Verlag, New York, 1999.
17. J. Lygeros, C. Tomlin, and S. Sastry. Controllers for Reachability Specifications for Hybrid Systems, Automatica, Volume 35, Number 3, March 1999.
18. I. Mitchell, and C. Tomlin. Level Set Methods for Computation in Hybrid Systems, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, Springer Verlag, 2000.
19. S. Osher, and J. A. Sethian. Fronts Propagating with Curvature-dependent Speed: Algorithms based on Hamilton-Jacobi Formulations, *J. Computat. Phys.*, vol. 79, pages 12-49, 1988.
20. T. J. Koo, G. Pappas, and S.Sastry. Mode Switching Synthesis for Reachability Specifications, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, Springer Verlag, 2001.
21. G. Lafferriere, G.J. Pappas, S. Yovine. Reachability Computation for Linear Hybrid Systems, In Proceedings of the 14th IFAC World Congress, volume E, pages 7-12, Beijing, 1999.
22. E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli, Effective Synthesis of Switching Controllers for Linear Systems, Proceedings of the IEEE, 88(2):1011-1025.

23. A.B. Kurzhanski, P.Varaiya, Ellipsoidal Techniques for Reachability Analysis, Hybrid Systems : Computation and Control, Lecture Notes in Computer Science, 2000.
24. A. Chutinan, B.H. Krogh, Verification of polyhedral-invariant hybrid systems using polygonal flow pipe approximations, Hybrid Systems : Computation and Control, Lecture Notes in Computer Science, 1999.
25. P. Tabuada, G. Pappas, and P. Lima. Feasible Formations of Multi-Agent Systems, in *Proceedings of American Control Conference*, pages 56-61, Arlington, Virginia, June, 2001.
26. A. Pant, P. Seiler, T. J. Koo, and J. K. Hedrick. Mesh Stability of Unmanned Aerial Vehicle Clusters, in *Proceedings of American Control Conference*, pages 62-68, Arlington, Virginia, June, 2001.
27. C. Tomlin, J. Lygeros, and S. Sastry. A Game Theoretic Approach to Controller Design for Hybrid Systems, in Proceedings of the IEEE, pages 949-970, Volume 88, Number 7, July 2000.
28. O. Maler, A. Puneli, and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems, in *STAC 95: Theoretical Aspects of Computer Science*, E.W. Mayr and C. Puech (eds). Munich, Germany: Springer-Verlag, 1995, vol. 900, Lectures Notes in Computer Science, pages 229-242.
29. E. Asarin, O. Maler, and A. Puneli. Symbolic Controller Synthesis for Discrete and Timed Systems, in *Proceedings of Hybrid Systems II*, P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (eds). Berlin, Germany: Springer-Verlag, 1995, vol. 999, Lectures Notes in Computer Science.
30. M. Heymann, F. Lin, and G. Meyer. Control Synthesis for a Class of Hybrid Systems subject to Configuration-Based Safety Constraints, in *Hybrid and Real Time Systems*, O. Maler (ed). Berlin, Germany: Springer-Verlag, 1997, vol. 1201, Lectures Notes in Computer Science, pages 376-391.
31. H. Wong-Toi. The Synthesis of Controllers for Linear Hybrid Automata, in *Proceedings of IEEE Conference on Control and Decision*, San Diego, CA, 1997.
32. T. D. Parsons. Pursuit-Evasion in a Graph, *Theory and Application of Graphs*, pages 426-441, Y. Alani and D. R. Lick (eds), Springer-Verlag, 1976.
33. J. P. Hespanha, H. J. Kim, and S. Sastry. Multiple-Agent Probabilistic Pursuit-Evasion Games, in *Proceedings of IEEE Conference on Decision and Control*, pages 2432-2437, Phoenix, Arizona, December 1999.
34. G. C. Buttazzo. *Hrad Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer, 1997.
35. F. Balarin, L. Lavagno, P. Murthy, A. Sangiovanni-Vincentelli. Scheduling for Embedded Real-Time Systems, *IEEE Design and Test of Computers*, pages 71-82, January 1998.