Software Engineering Institute
Carnegie Mellon University

CYBER-PHYSICAL SYSTEMS

Bjorn Andersson, Sagar Chaki, Dionisio de Niz, Jeffery Hansen, Scott Hissam, John Hudak, Mark Klein, David Kyle, Gabriel Moreno

December 2016

Overview

Cyber-physical systems (CPS) are "engineered systems that are built from, and depend upon, the seamless integration of computational algorithms and physical components" (NSF). Our objective is to enable efficient development of high-confidence distributed CPSs whose nodes operate in a provably correct manner in terms of functionality and timing (synchronicity between physical and software components), leading to predictable and reliable behavior of the entire system. To this end, we develop scalable algorithms for functional analysis of real-time software, techniques for controlling and analyzing the effects of multicore memory access on CPS real-time behavior, and techniques for assuring coordination strategies. We also target both deterministic and stochastic CPSs. In addition, we develop, analyze, and validate portable architecture and middleware to support user-directed groups of autonomous sensors and systems. Accordingly, our research includes a number of mutually reinforcing threads.

1. **Timing Verification** to guarantee that tasks in real-time systems complete within their deadlines. For instance, the airbag of a car must completely inflate within 20 minutes; otherwise, the driver can hit the steering wheel with fatal consequences. We are developing schedulability techniques for multicore platforms, where new challenges of shared resources, such as memory, invalidate previous assumptions in single core. We are also developing scheduling techniques for mixed-criticality systems. In addition, we want to incorporate elements of the physical processes to improve qualities of the system such as resilience and performance.

2. **Functional Verification** to ensure that software behaves as required. We are developing new model checking algorithms for periodic real-time software to ensure logical correctness properties, such as absence of race conditions, deadlocks, and other concurrency errors that may lead to unsafe or undesired behavior. We also use model checking and code generation to produce high-assurance distributed software.

3. **Probabilistic Verification** to maximize the likelihood that a CPS will meet its desired goals. We are exploring probabilistic analysis techniques to estimate, with high accuracy, the chances of a desired goal being achieved (or an undesired outcome being avoided). In particular, we are investigating two analysis techniques: (a) numerical approaches based on probabilistic model checking of Markov chains and Markov decision processes, and (b) techniques based on Monte Carlo simulation such as statistical model checking and importance sampling.

4. **Collaborative Autonomy** to optimize scalability, performance, and extensibility for autonomous systems by creating a portable, open-sourced, decentralized operating environment. We integrate this environment into unmanned autonomous systems (UAS) platforms, smartphones,

tablets, and other devices and design algorithms and tools to perform mission-oriented tasks. We also design user interfaces to help single human operators control and understand a swarm of UAS, devices, and sensors.

5. **Self-Adaptation** to address the challenge of having cyber-physical systems that can quickly adapt to a variety of situations including environment changes, and malfunctions. A self-adaptive system is a system capable of changing its behavior and structure to adapt to changes in itself and its operating environment without human intervention. We are developing architecture-based self-adaptation approaches that take into account the latency of the available adaptation strategies when deciding how to adapt. Furthermore, our techniques leverage short term predictions of the environment evolution to enable proactive adaptation.

How We Can Help

The SEI helps organizations to

- apply formal verification techniques and tools to assure critical system properties
- apply real-time analysis techniques to determine if critical system timing properties will be satisfied
- provide design and implementation guidance for real-time, cyber-physical systems
- support user-directed groups of autonomous sensors and systems

Research

The goal of **High-Confidence Cyber-Physical Systems** work is to enable efficient development of autonomous CPSs. To ensure that the collective behavior of distributed elements is predictable and reliable, we must demonstrate

- scalable algorithms for functional analysis of real-time software
- techniques for controlling effects of multicore memory access on CPS real-time behavior
- techniques for assuring distributed autonomous coordination
- techniques for developing architecture and middleware to support user-directed groups of autonomous sensors and systems

Accordingly, our current research includes a number of mutually reinforcing threads.

Our research in high-confidence cyber-physical systems involves developing

- new real-time scheduling theories to ensure predictable timing behavior

- new representations of the concurrency aspects of systems that account for the intimate relationship between physical and computational realms
- new techniques for ensuring predictable collaboration among autonomous agents
- new static-analysis techniques for efficiently ensuring safety assertions of concurrent system

Functional Verification

The goal of functional verification is to ensure that the behavior of CPS software respects its specification when executing in a given environment. We are targeting both deterministic and stochastic systems. The specification expresses a safety condition, (e.g., no deadlocks, no violations of user-specified assertions) that must be satisfied in all possible executions of the software (for deterministic systems) or with some required minimum likelihood (for stochastic systems). The environment captures details of software (e.g., number of threads and their priorities), the operating system (e.g., scheduling policy), and the nature of communication (e.g., shared memory, message passing). We have two main research directions in functional verification:

1. Model Checking Periodic Real-Time Software
2. Model Checking Distributed Systems

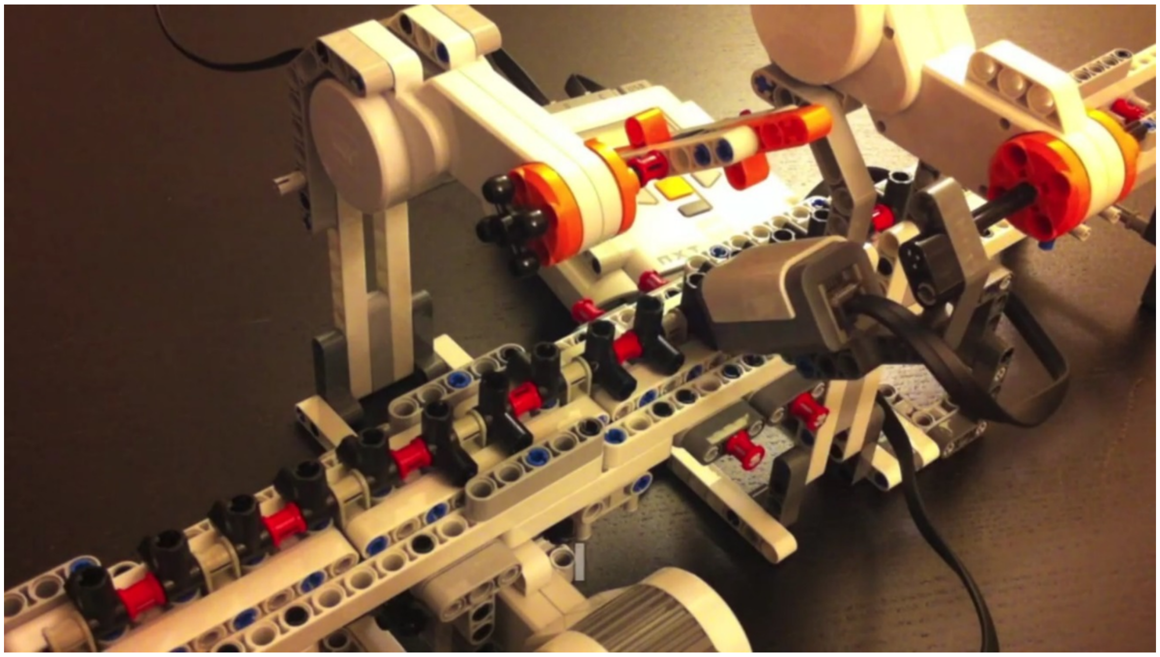
Our solutions are based on automated and exhaustive techniques (such as model checking and probabilistic model checking). We publish our results in peer-reviewed venues, and implement our algorithms in prototype tools, which are validated on examples guided by real-life systems and scenarios.

Real-Time Software

The goal of the Static Analysis of Real-Time Systems (START) project is to verify functional correctness of real-time embedded software (RTES). The current focus is on model checking periodic RTES with Rate-Monotonic scheduling, a.k.a, periodic programs. Such programs are widely used in the automotive and avionics domains. Specifically, we verify whether an assertion *Spec* can fail when the RTES is executed from an initial state *Init*. We assume that the RTES is written in C and consists of a set of periodic tasks with rate monotonic scheduling. We also assume that the RTES is schedulable and that the user knows the priority and worst-case execution time of each task. The user specifies the assertion *Spec*, and initial state *Init*. In past work, we also required a user-supplied time bound *Bound* that limits the execution of the software. In ongoing work, we are eliminating this requirement (i.e., we will verify correctness of a periodic program even if it executes for unbounded time).

Case Study: The LEGO Turing Machine

A Turing machine is the simplest form of computer. It is composed of a ribbon of paper called a tape, which stores data on a head that can read symbols on the tape, write a new symbol, and move left or right a list of transitions that tells the machine what to do next. Soonho Kong, a PhD student at Carnegie Mellon University, interned at the SEI in the summer of 2012 and built a Turing machine from LEGO components to demonstrate the START verification tool. Kong began by implementing a single task, and then added more features incrementally to create a multitasking machine. First, Kong programmed the Turing machine to perform Read, Write, and Move operations one at a time. You can see it in action in the following video: https://youtu.be/teDyd0d5M4o [Kong 2012].



The LEGO Turing Machine Performs Single-Task, Unary Addition

This sequential-operation performance wastes time because the machine could perform some operations simultaneously. To implement simultaneous movements, Kong programmed multiple tasks along with their constraints. For example, the head can move toward the tape while the tape advances to the next position, but then the tape must hold still so the color sensor can read it. The writer lever and the head can move simultaneously, but they must avoid physical collision. This multitasking with constraints provided an application for the START program to test the verification tool to ensure that an implementation has all the desired concurrency properties.

Distributed Software

Distributed systems are becoming an important part of safety-critical and mission-critical systems. They are also being endowed with more autonomy and coordination capabilities to increase effectiveness and reduce operator overload. They often operate in uncertain environments and must meet both guaranteed (e.g., collision avoidance) and best effort (e.g., area coverage within deadline) requirements. We are pursuing two research directions in functional verification of such systems:

1. **Probabilistic Verification of Distributed Coordinated Multi-Agent Systems:** This project explores analytic methods for predicting the quality of coordination mechanisms for multi-

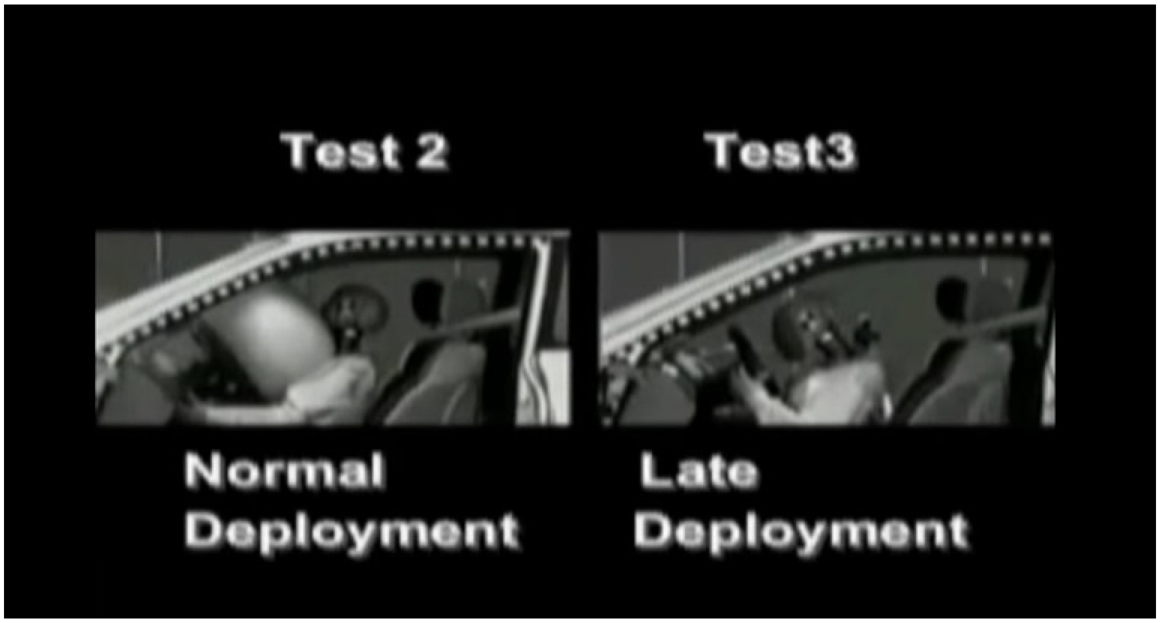
agent systems in uncertain environments (e.g., robots in a minefield). In previous work, we have developed, implemented, and validated highly scalable, compositional, probabilistic model checking algorithms for such systems with limited coordination. Our current focus is on probabilistic model checking to make predictions about teams of agents with more complex coordination. Our approach is to first build Discrete Time Markov Chain (DTMC) models for each agent, based on running and observing them individually. Next these models are composed and verified, using the probabilistic model checker PRISM to obtain the prediction. We have proved the correctness of this approach formally and implemented it. We have also developed a way to quantify the error in our predictions. Such errors are unavoidable, due to the fact that our models are constructed from a finite number of observations.

2. **Model Checking Distributed Applications:** This project is exploring the use of domain-specific languages, software model checking, and code generation to produce verified distributed applications.

Timing Verification

Real-Time Scheduling for Timing Verification

In Cyber-Physical Systems, the software must execute in sync with the physical processes it interacts with. For instance, in a car airbag system, the software must finish the evaluation of a crash and trigger the inflation of the airbag in order to complete the process in less than 20 milliseconds; otherwise the driver can be seriously injured. This completion time limit for the software is known as a *real-time deadline*. Guaranteeing these deadlines—even in the worst-case scenario (as when all tasks arrive simultaneously) —has been the study of real-time scheduling theory.



Airbag deployment is an example of a real-time deadline that is critical for passenger safety. To view a video that shows the consequences of late airbag deployment, visit https://youtu.be/YAwrq9-1oQQ [Todd Tracey Law Firm 2012]

In recent years, two trends have imposed new challenges to the timing verification for real-time systems: (i) the functional consolidation of tasks of different criticalities in shared processors--these are called *mixed-criticality real-time systems*, and (ii) the advent of multicore processors. The SEI has been researching these two areas.

Zero-Slack Scheduling

In complex military systems such as airplanes and UAVs, there is an increasing need to consolidate more and more functions into a single processor. This consolidation imposes a challenge when these functions have different criticalities. For instance, one function may be controlling the stability of the airplane while another may be processing video for a surveillance mission. While the function that controls the stability of the flight is safety-critical (i.e., a failure in this function can crash the airplane) the video processing function is not. The difference in criticality poses a challenge for the verification, validation, and certification of whole systems because functions of different criticality are held to different standards of certification. In particular, while a complex and expensive verification process may be used for safety-critical functions, a simplified and cheaper process must be applied to non-safety-critical functions in order to keep the cost under control. Unfortunately, if functions from different criticalities share a processor, failures in a low-critical function may propagate to a higher critical one--for example, because the faulty function may hold execute longer than expected, delaying the execution of the critical task beyond its tolerance (unable to compensate for cross winds). This means that if we want to preserve the quality of the verification of the safety-critical functions, we would need to apply the same complex verification process to the lower criticality functions, given the failure propagation possibility. Fortunately, if we can ensure that high-criticality tasks are protected against failures in lower criticality tasks, then we can still apply different verification processes to functions of different criticalities.

The Air Force Research Laboratory (AFRL) has recognized these challenges and created an initiative called the "Mixed-Criticality Architecture Requirements" (MCAR) to investigate the technology required to implement these protection mechanisms.

The Zero-Slack Mixed Criticality Scheduling is a scheduler that implements temporal protection (i.e., ensures that tasks are not delayed) of high-criticality tasks against lower criticality ones. In particular, during an overload, we ensure that higher criticality tasks are able to finish on time (meet their deadline) even at the expense of lower criticality ones.

Zero-Slack QRAM

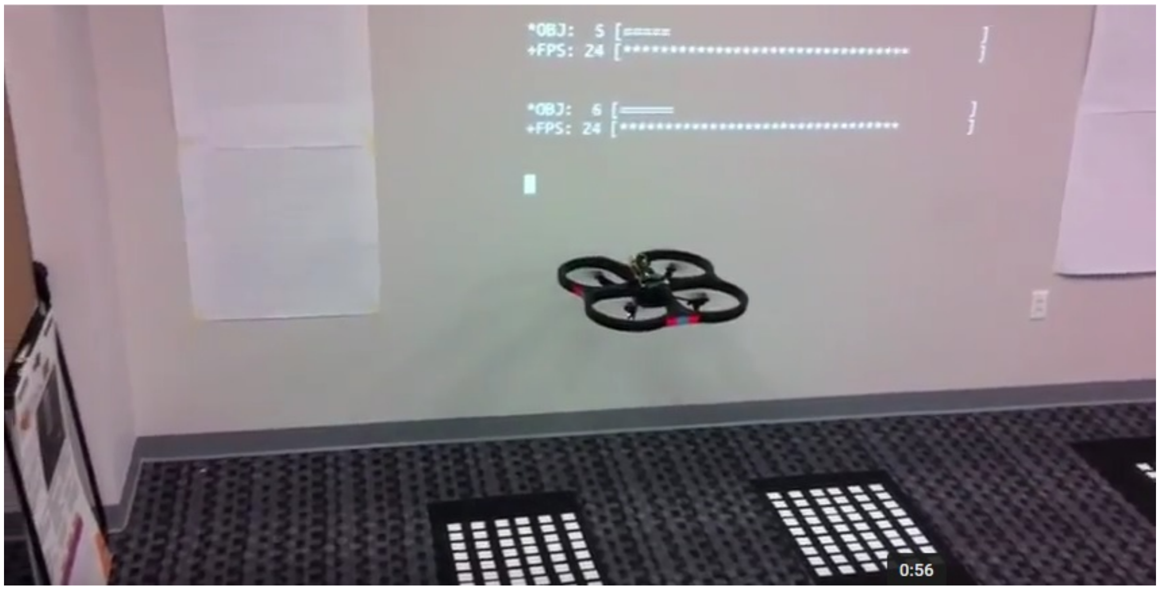
Zero-slack scheduling is a scheduling framework for real-time systems of mixed criticality. Specifically, it targets systems where the utilization-based scheduling priorities are not aligned with the criticality of the tasks. With this framework, we implemented a family of schedulers, resource-allocation protocols, and synchronization protocols to support the scheduling of mixed-criticality systems.

The zero-slack QoS resource-allocation model (Q-RAM) combines zero-slack rate monotonic scheduling and Q-RAM to enable overbooking, in which the same CPU cycles are allocated to more than one task. Zero-slack Q-RAM allows overbooking not only between tasks of different criticality but also among tasks with different utility to the mission of the system. In a given cycle, if a more critical task must execute, that task uses the cycle; otherwise, a task of lower criticality will execute.

We developed several experiments to determine the effects of this scheduler in a drone mission. First, we demonstrated how the wrong scheduling can actually crash a drone. The following video shows how the increasing demands of lower-criticality tasks decrease flight safety: https://youtu.be/7OuF9foutlQ [DroneRKwiki 2011a].

As background task utilization rises, the drone becomes more difficult to control.

Second, we show that in a full mission, zero-slack Q-RAM not only preserves the safety of the flight but also maximizes the utility of the mission. The demo in the following video shows a surveillance mission in which a video-streaming task and an object-recognition task are dynamically adjusted according to their utility to the mission. To view the video, visit https://youtu.be/hznBzKKf3lo [DroneRKwiki 2011b].



```
*OBJ:  5 [=====                                          ]
+FPS: 24 [**********************************             ]

*OBJ:  6 [======                                         ]
+FPS: 24 [*********************************              ]
```

Zero-slack Q-RAM makes the drone more controllable.

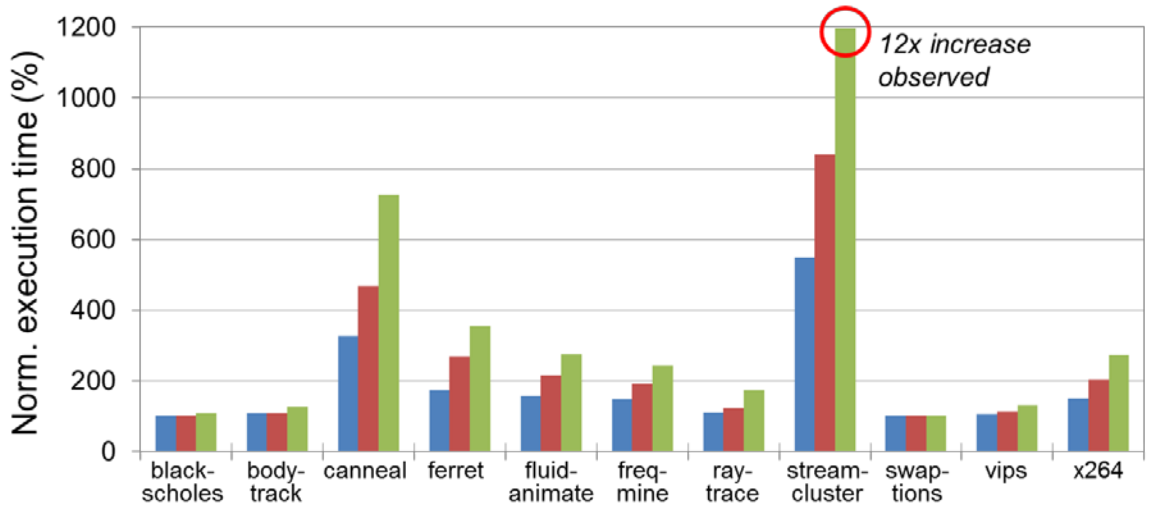
Multicore Scheduling

While scheduling in multiprocessor real-time systems is an old problem, multicore processors have brought a renewed interest, along with new dimensions, to the challenge. For instance, there is a need to tradeoff different levels of migration cost, different degrees of inter-core hardware sharing (e.g. memory bandwidth), and so on. Our research is aimed at providing new knobs to perform these tradeoffs with additional application information.

Beyond real-time systems, general-purpose systems are now faced with the fact that they need to parallelize their work in order to get the expected performance increment from additional cores in the new processors. However, partitioning the work into parallel pieces is a necessary but not sufficient condition. Equally important is the allocation of CPU cycles to these parallel pieces (tasks). In the extreme, if we run all the tasks of the parallel pieces in the same core, such parallelism is completely wiped out. Hence, the task-to-core allocation and the scheduling of hardware resources between core (e.g., cache, memory bandwidth) can change completely the performance of these systems. We are working on new ways to take advantage of application knowledge to use them as parameter in the scheduling algorithms at all levels of the computer system.

Inter-core Memory Interference in Multicore Processors

Multicore processors are quite different from multi-processors. This is due to the fact that cores within a processor share resources. One of the most critical shared resources is the memory system. This includes both shared cache and shared RAM memory. The effect of the memory interference that one task running on one core has on another running on a different core can be highly significant. We have seen extreme cases of 12X increases in the execution time due to memory interference (as the figure below illustrates), and some practitioners have observed 3X increases. This 3X basically means that in a dual core processor I am better off shutting down a core to avoid a decrease in the execution speed.



Impact of memory interference.

Once we solve the interference problem we have another challenge. Due to shared resources, we must support new tasks with parallelized jobs that require more than one core to complete before the deadline. New scheduling algorithms are necessary to schedule these tasks and must be combined with memory partitions to maximize their utilization and guarantee their time predictability.

At the SEI, we have been working with the CMU Real-Time and Multimedia Systems Laboratory to address the shared memory challenge by creating partitioning mechanisms to eliminate or reduce interference, along with analysis algorithms to take into account residual effects.

Partitions with Page Coloring

A key mechanism we use is page coloring. Page coloring takes advantage of the virtual memory system that translates virtual memory addresses to physical addresses. This mechanism assigns physical addresses that do not interfere with each other to different tasks running on different cores. Page coloring works in combination with characteristics of the memory hardware that divides the memory into areas that do not interfere with each other. Different mechanisms exist for cache and main memory.

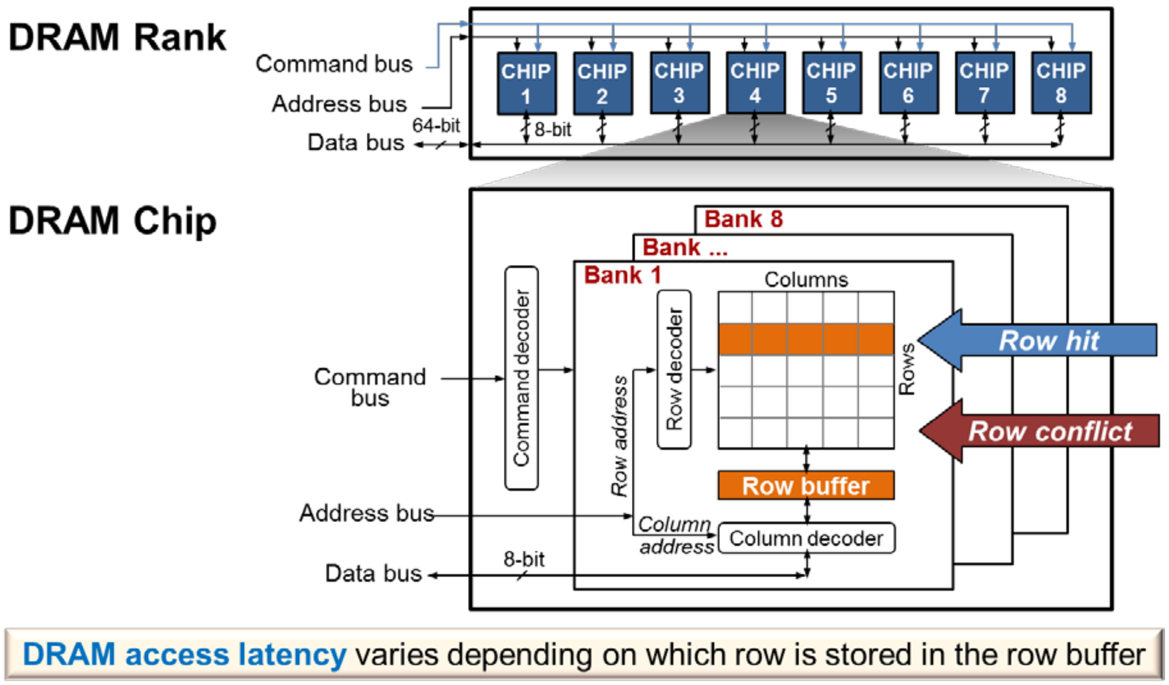
Cache Partitions (Cache Coloring)

Cache is fast memory that is used in the memory system to speed up the access to frequently used data (e.g. variables). Specifically, when a variable is first accessed, it is loaded into cache so that subsequent accesses to the same variable are performed from the cache instead of from main memory much faster. However, caches are much smaller than main memory, and as the program executes it will stop using some variables. The cache system knows how frequently each variable is accessed, so when the cache must add a newly accessed variable but the cache is full, the system clears the least frequently accessed cache block to make room. Accessing that cleared variable will take additional time because it must be loaded back from the slower main memory, as it was the first time it was loaded.

Most cache hardware divides the cache into sets of cache blocks in what is known as set-associativity. Each set is restricted to be used for certain area of the physical memory and cannot be used by any other. We take advantage of this restriction and ensure that the physical memory used by one task in one core belongs to one of these regions, while the memory of another task running on a different core belongs to a different one. This is known as cache coloring, which effectively creates cache partitions.

Memory Bank Partitions (Bank Coloring)

While cache coloring provides a great benefit to reduce the interference across cores, it's not sufficient to solve the problem. Main memory is another source of significant interference. In fact, the experimental results shown in the figure above are due to memory interference, not cache interference. Main memory is divided into regions called banks, and these banks are organized into rows and columns. When a task running in a core tries to access a memory address in main memory, this address first is analyzed to extract three pieces of information (from specific bits in the memory address): (i) the bank number, (ii) the row number, (iii) the column number. The bank memory is used to select the bank where the memory block is located. Then the memory controller loads the row from that bank into a row buffer within the bank for faster access. Finally the memory block is accessed in the column indicated by the column number from the row buffer. This is illustrated in the figure below.



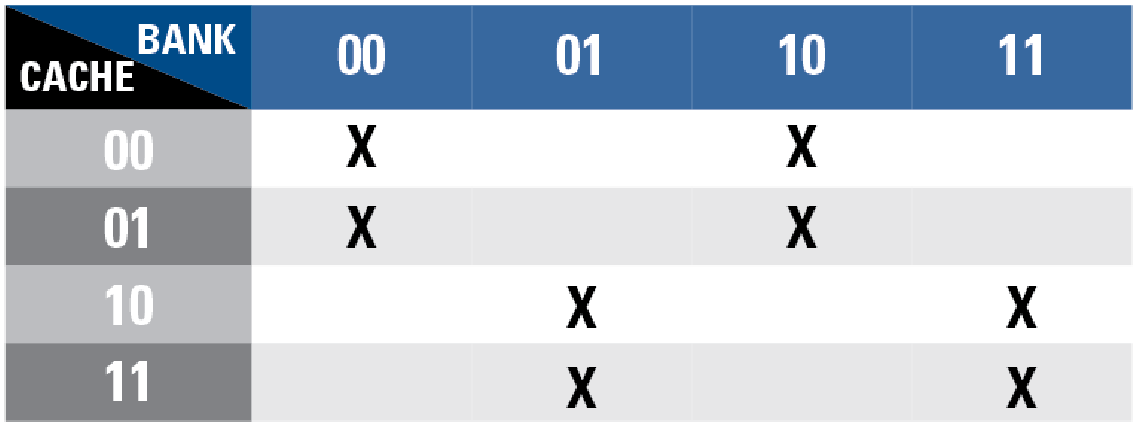
Memory bank partitioning

Because the memory controller is optimized to improve the number of memory accesses per second, it takes advantage of the row buffer and favors the memory accesses that go to the same row. Unfortunately, this means that when a task 1 in one core is accessing a row (already loaded in the row buffer) while a task 2 running in another core is trying to access another row in the same bank, the access from task 2 can be moved back in the memory access queue by another, more recent access from task 1 to the already-loaded row multiple times, creating an important delay for task 1.

Memory bank partitions are created by mapping the memory of the different tasks to different memory banks. In this way each task can have its own bank and row buffer, and no other task will modify that buffer or the queue of memory accesses to this bank.

Combined Cache and Bank Partitions

Because caches and memory banking technologies were not developed together, their partitions often intersect each other. In other words, it is not possible to select a bank color independently from a cache color because the selection of a cache color may limit the number of bank colors available. This occurs because, in some processor architectures, the address bits used to select a bank and the bits used to select a cache set share some elements. To illustrate this idea, consider a memory system with four banks and four cache sets. In this case, we need two address bits to select a bank and two bits to select a cache set. If they were independent, we could select four cache colors for each selectable bank color for a total of 16 combinations. We can visualize this as a matrix (a color matrix) where rows are cache colors and columns are bank colors. However, if cache and bank colors share one bit, then in reality we only have 2^3=8 colors. This means that in the color matrix some of the cells will not be real. The figure below illustrates this concept.



CACHE \ BANK	00	01	10	11
00	X		X	
01	X		X	
10		X		X
11		X		X

Combined cache and bank positions

Coordinated Cache, Bank, and Processor Allocation

We developed a coordinated approach to allocate cache and bank colors, along with processor, to tasks in order to avoid the cache/bank color conflicts. Our approach also maximizes the effectiveness of the memory partitions, by taking into account the difference between inter- and intra-core interference.

We developed memory reservations with cache and memory partitions in the Linux/RK OS.

Limited Number of Partitions

Unfortunately, the number of partitions obtainable through page coloring is limited. For instance, for an Intel i7 2600 processor, it is possible to obtain 32 cache colors and 16 bank colors. Given that, in practice, we may have a larger number of tasks (say 100), this number of partitions may prove insufficient for a real system. As a result, it is important to also enable the sharing of partitions whenever the memory bandwidth requirements of tasks allow it. However, this sharing must be done in a predictable way to ensure we can guarantee meeting task deadlines. At the same time, it is important to avoid pessimistic over-approximations, in order not to waste the processor cycles that we are trying to save in the first place. For this case, we developed an analysis algorithm that allows us to verify the timing interference of private and shared memory partitions.

Parallelized Tasks Scheduling

Beyond solving the resource-sharing problem, we also need to enable the execution of parallelized tasks. For this we have developed a global EDF scheduling algorithm for parallelized tasks with staged execution. These tasks generate jobs composed of a set of sequential stages. These stages are further composed of a set of parallel segments. These segments are allowed to run in parallel to each other, provided that all the segments from the previous stage have completed, or the task has arrived for the first segments of the first stage. Our algorithm allows us to verify the schedulability of these tasks with a global EDF scheduler. Beyond EDF, it is possible to use this algorithm with a global fixed priority scheduler with synchronous start, harmonic periods, and implicit deadlines, a common configuration used by practitioners.

Probabilistic Verification

Many CPSs operate in uncertain environments. From the software perspective, this means that some of its inputs are random variables. Given this randomness, the natural way to frame the verification problem is to compute the likelihood that the software satisfies a safety specification (e.g., the likelihood that a periodic real-time task never misses its deadline five times in a row, or the likelihood that the destination is reached with a preset deadline). We are exploring two main techniques for verification of stochastic CPSs:

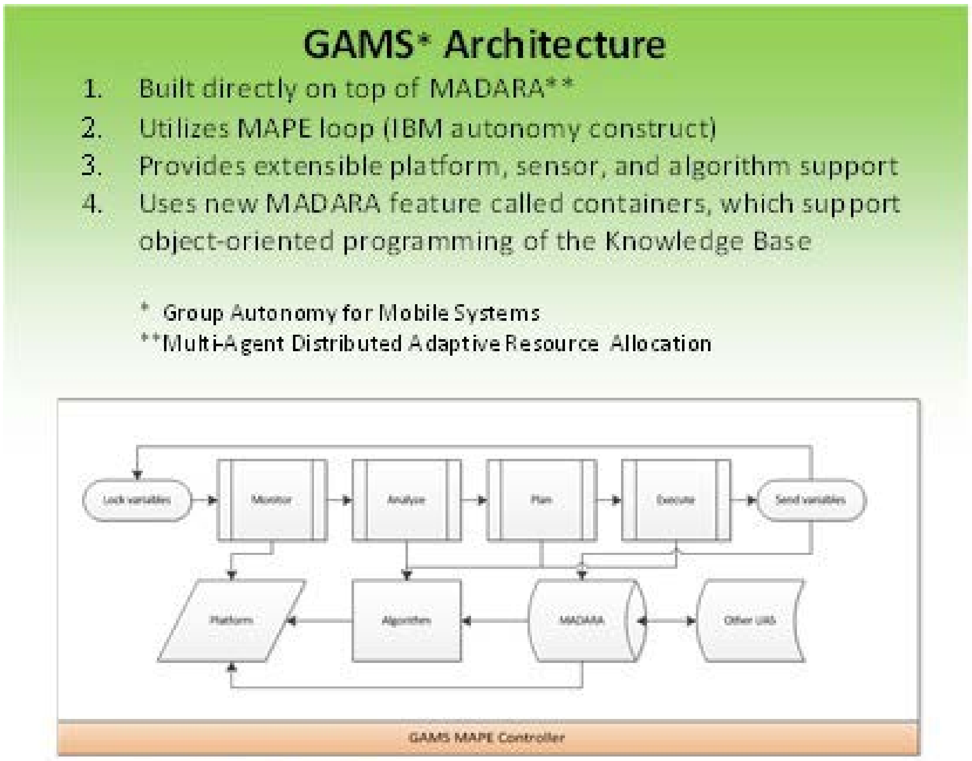
1. **Probabilistic Model Checking:** In this approach, the system is modeled as a Markov Chain and the likelihood of a property is computed by constructing a set of equations and solving them numerically. We are exploring the use of probabilistic model checking to evaluate the quality of coordination schemes in distributed multi-agent systems. For more information, see the section on **Functional Verification**.

2. **Statistical Model Checking:** In this approach, each execution of the system is treated as a Bernoulli trial and the likelihood of a property is computed via Monte Carlo simulations. A key challenge is to get a high-precision result with a small number of simulations, which is non-trivial for properties whose violations are rare events. We are exploring novel *importance sampling* techniques to achieve this goal for stochastic CPS software.

Collaborative Autonomy

Collaborative Autonomy for Mobile Systems architects, designs, analyzes, and validates portable architecture and middleware to support user-directed groups of autonomous sensors and systems. The current focus is on

- middleware that creates a decentralized, distributed operating environment for swarms of sensors and robots, guided by a human user
- area coverage techniques that specialize in prioritized zones and mission objectives
- algorithms that prioritize information flows and route mobile sensors/drones/robots into locations that best serve mission utility



GAMS* Architecture

1. Built directly on top of MADARA**
2. Utilizes MAPE loop (IBM autonomy construct)
3. Provides extensible platform, sensor, and algorithm support
4. Uses new MADARA feature called containers, which support object-oriented programming of the Knowledge Base

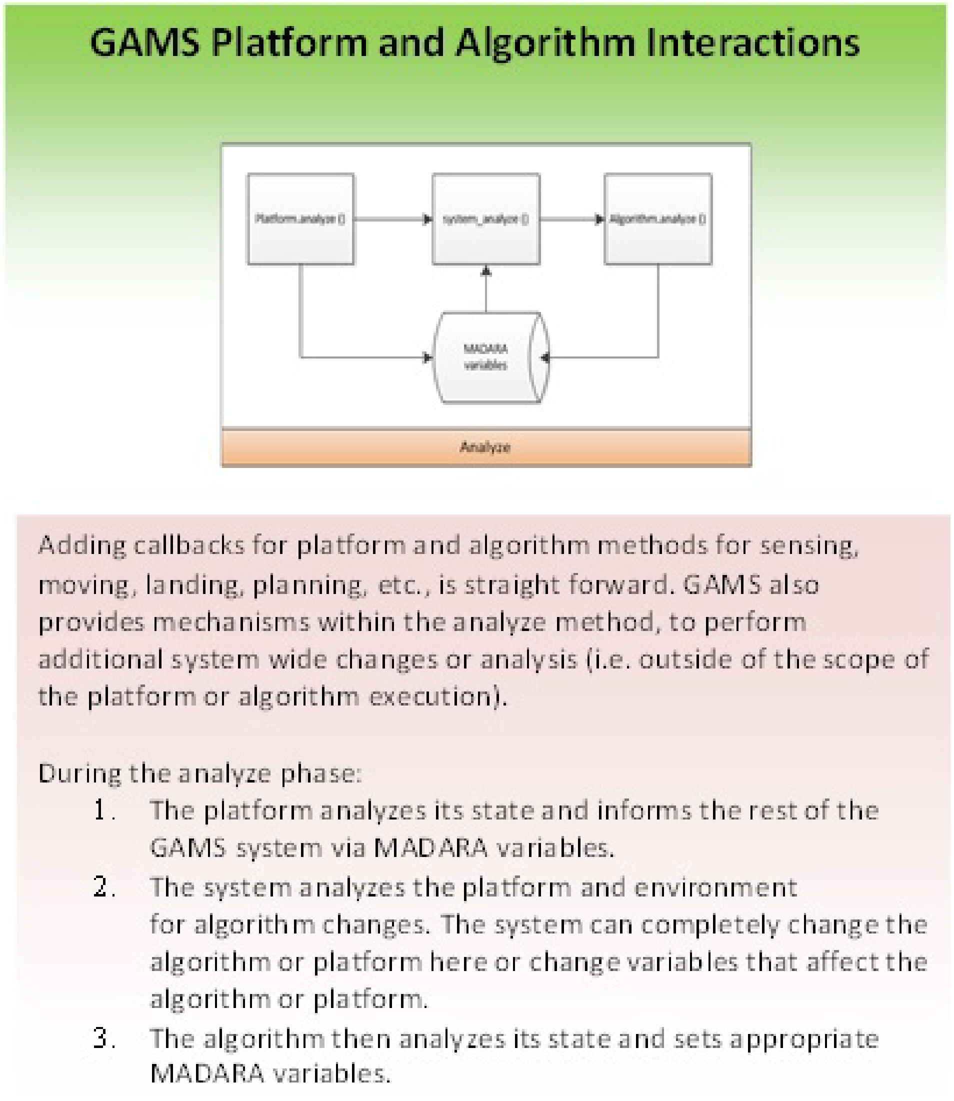
* Group Autonomy for Mobile Systems
**Multi-Agent Distributed Adaptive Resource Allocation

GAMS MAPE Loop

Collaborative Autonomy Challenges

- Autonomy focus is on single unit control.
- Focus is on centralized controllers (prone to failure/attack).
- Autonomy frameworks tend to be targeted at homogeneous platforms and algorithms.

- Blocking communications are prone to faults/attacks/ outages/loss-of-control. GPS is highly inaccurate for precise maneuvers.
- There is a lack of standardization for autonomous collaboration.



GAMS Platform and Algorithm Interactions

Adding callbacks for platform and algorithm methods for sensing, moving, landing, planning, etc., is straight forward. GAMS also provides mechanisms within the analyze method, to perform additional system wide changes or analysis (i.e. outside of the scope of the platform or algorithm execution).

During the analyze phase:

1. The platform analyzes its state and informs the rest of the GAMS system via MADARA variables.
2. The system analyzes the platform and environment for algorithm changes. The system can completely change the algorithm or platform here or change variables that affect the algorithm or platform.
3. The algorithm then analyzes its state and sets appropriate MADARA variables.

Interactions between GAMS platform and algorithm

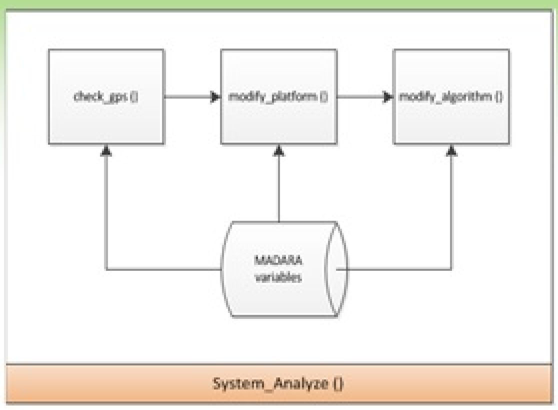
Our Approach to Collaborative Autonomy

1. Create a portable, open-sourced, decentralized operating environment for autonomous control and feedback. Focus on scalability, performance, and extensibility.
2. Integrate the operating environment into unmanned autonomous systems (UAS), platforms, smartphones, tablets, and other devices. Focus on portability.

3. Design algorithms and tools to perform mission-oriented tasks such as area coverage and network bridging between squads.

4. Design user interfaces to help single human operators control and understand a swarm of UAS, devices, and sensors (human-in-the-loop autonomy).

GAMS Platform and Algorithm Interactions



As mentioned earlier, the system_analyze () method can perform many tasks that may be useful to developers of real-world distributed autonomy. The default behavior of system_analyze, which can be overridden by extending the Base_Controller class, is the following:

1. The platform can inform the control loop of gps-spoofing, if it has capabilities, through common MADARA variables that are analyzed during check_gps()
2. Check_gps () is also intended to implement gps-spoof checking in software by checking velocities and previous positions for anomalies.
3. Environmental or platform characteristics can result in changes to the platform (e.g., an arm is damaged) or algorithm (e.g., the UAS should return home instead of completing a search algorithm because of danger or low battery).

Further interactions between GAMS platform and algorithm

Self-Adaptation

One of the challenges for CPSs is that they must be able to adapt to anomalies in themselves and in their environments, such as a degraded or failed sensor, and problems with systems to which they connect, including the infrastructure they use. Because CPSs will be ubiquitous, it will not be possible to have human operators continuously monitoring and managing them. Consequently, CPSs will be required to monitor themselves and take corrective actions as needed, either to fix problems or to improve their behavior.

Although CPSs of today have some ability to deal with changes in the environment, the approaches used to do so have several drawbacks. As an example, the adaptation code is typically entangled with the application code in the form of exception handling or conditionals. This low-level handling of the adaptation may result in taking inadequate actions that are based only on information local to that particular component. For example, one component may decide to retry sending a request to another component, assuming that the previous request was lost, when the appropriate action would be to stop sending requests until the target component, which had failed in this case, is restarted. In order to be able to make the appropriate decision in situations like this, it is necessary to have a more comprehensive and higher level view of the system to reason about the problem and the required adaptation to deal with it. The system's architecture provides the high-level perspective required to reason about the system itself and the adaptations.

In architecture-based self-adaptation, a model of the architecture of the running system is maintained at runtime and used to reason about the changes that should be made to the system to achieve the desired quality attributes. Several existing techniques for analyzing software architectures can be used to reason about the current system configuration and the possible alternatives to which it could adapt. For example, the recent self-diagnosis approaches can identify the architectural element most likely to have caused a failure, and the performance of different alternative architectures can be analyzed by transforming their architecture models into performance models that can be evaluated.

We are developing approaches to improve architecture-based self-adaptation so that adaptations can be done proactively rather than as reaction to changes. Achieving this requires explicitly considering the time it takes for adaptation strategies to be executed. Furthermore, we are using probabilistic model checking to quantitatively verify properties of the self-adaptive system.

Bibliography

[de Niz & Moreno 2012]

de Niz, Dionisio & Moreno, Gabriel. *An Optimal Real-Time Voltage and Frequency Scaling for Uniform Multiprocessors.* Software Engineering Institute, Carnegie Mellon University. August, 2012. http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=27055

[DroneRKwiki 2011a]

DroneRKwiki, Department of Electrical and Computer Engineering and Software Engineering Institute, Carnegie Mellon University. Impact of background task on AR Drone control loop without real-time kernel [video]. *YouTube.* March 31, 2011. https://www.youtube.com/watch?v=7OuF9foutlQ

[DroneRKwiki 2011b]

DroneRKwiki, Department of Electrical and Computer Engineering and Software Engineering Institute, Carnegie Mellon University.Zero Slack Rate Monotonic Demo [video]. *YouTube.* December 12, 2011. https://www.youtube.com/watch?v=hznBzKKf3lo

[Edmondson 2013a]

Edmondson, James. Building Next-generation Autonomous Systems [blog post]. *SEI Insights.* January 24, 2013. https://insights.sei.cmu.edu/sei_blog/2013/01/building-next-generation-autonomous-systems.html

[Edmondson 2014]

Edmondson, James. *Collaborative Autonomy with Group Autonomy for Mobile Systems (GAMS).* Software Engineering Institute, Carnegie Mellon University. August, 2014. http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=301720

[Edmondson 2013b]

Edmondson, James. *Group Autonomy for Mobile Systems*. Software Engineering Institute, Carnegie Mellon University. June, 2013. http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=301710

[Edmondson & Garcia-Miller 2013]

Edmondson, James & Garcia-Miller, Suzanne (interviewer). Human-in-the-Loop Autonomy. *SEI Podcast Series.* September 12,2013. http://www.sei.cmu.edu/podcasts/podcast_episode.cfm?episodeid=60990

[Edmondson 2013c]

Edmondson, James. Multi-UAV Search and Rescue (SEI-Official) [video]. *YouTube.* November 6, 2013. https://www.youtube.com/watch?v=iLnNHwp-H8E&feature=youtu.be

[Kong 2012]

Kong, Soonho. LEGO Turing Machine – Single task unary addition [video]. *YouTube*. July 23, 2012. https://www.youtube.com/watch?v=teDyd0d5M4o

[Rajkumar et al 2017]

Rajkumar, Ragunathan (Raj); de Niz, Dionisio; & Klein, Mark. *Cyber-Physical Systems*. Addison-Wesley Professional. ISBN: 9780133416152. January, 2017.

[Todd Tracey Law Firm 2012]

Todd Tracey Law Firm. AIRBAG defect - Late and early Deployment [video]. *YouTube*. March 30, 2012. https://www.youtube.com/watch?v=YAwrq9-1oQQ

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu | www.cert.org
Email: info@sei.cmu.edu