# Toward Model-Based Verification of Adaptive Allocation Managers

William Leal, Frank Drews, Chang Liu, Lonnie Welch
Ohio University
{ leal@cs.ohiou.edu, drews@ohiou.edu, changliu@cs.ohiou.edu, welch@ohio.edu }

## 1 Introduction

Embedded computer systems have been used for many years to control satellites, automobiles, robots, unmanned aerial vehicles, medical instruments, radars, and other kinds of devices. In most cases, the tasks that these devices perform are pre-set and have only limited variation. However, as the use of embedded control has grown, so has the need for flexibility. Satellites that previously had performed a fixed function, such as taking photographs of the earth, now must be prepared to respond flexibly to changes in the environment. If, for example, a volcanic activity is detected, then the satellite's resources must be focused on high-resolution photographs of that event; or if there is cloud cover, then low resolution photographs suffice.

The problem with adding flexibility to a computerized control system is that it might no longer get its job done in a timely fashion. If the controller starts running slowly, an airplane could crash, a robot could walk off a cliff, and so forth. Hence with the flexibility comes the need to make certain guarantees about the timeliness of the system.

Mike Masters, Chief Scientist with Naval Surface Warfare Center, Dahlgren has developed a challenge problem statement [10] to guide researchers in addressing the problem of adaptive resource management. His requirements for certification of systems that employ adaptive resource management are summarized as follows:

1. Traditional functional and performance testing of system functional components.
2. Functional and performance testing of the adaptive resource allocation capability.
3. Verification of feasibility (that is, schedulability) of adaptive resource allocations. This means that allocations produced should be capable of being scheduled with respect to the available resources.
4. Verification of stability of adaptive resource allocations. This means that small changes in the environment should not cause a reallocation.

The focus of this paper is the third requirement: verification of feasibility. We begin by making the requirement precise, and then propose several ways to satisfy it. More information about adaptive resource management is in [9, 6].

## 2 The Verification Problem

Given a set of resources and a set of tasks that require those resources, an allocation manager must produce feasible allocations. There are other requirements as well: the stability requirement mentioned above, along with the requirement that the allocation manager perform its calculations within some specified deadline. These requirements are beyond the scope of this paper, which focuses on the problem of ensuring that a candidate allocation manager in fact produces feasible allocations.

We begin by defining those characteristics of a system that are relevant to our purpose. A system $S$ has the following components.

- A collection $\mathcal{R}$ of *resource classes*. Resource classes include CPU's, network connections, memory, disk storage, and so forth. In each class there can be several resources; for example, there could be several different CPU's of the same type but perhaps of different speeds. Each resource class has an associated measurement (MFLOPS, megabytes, and so forth) and each resource has an associated capacity in that measurement; hence resources in a class can be compared.

- A set $T$ of *tasks* with hard schedule requirements. Periodic tasks are run at some specified frequency; aperiodic tasks are run when required. In either case, the schedule is hard, which means that they must complete within their specified deadline.

  Each task also has an associated resource requirement, indicating for each class the amount of resource that is required for each execution of the task.

- A first-order predicate logic formula $q$ that specifies the *required task sets*. Let $\mathcal{Q}$ be the collection of required task sets: $\mathcal{Q} = \{Q : Q \vDash q\} \subseteq 2^T$, where $2^T$ is the powerset of $T$. Since the available resources may not be sufficient to run all tasks simultaneously, $q$ specifies the subsets that must be run together.

Given a required task set $Q$, a *feasible allocation* of $Q$ to $\mathcal{R}$ is an assignment of the tasks in $Q$ to resources in $\mathcal{R}$ such that each task has sufficient resources allocated, and no resource is over-allocated. An *allocation manager* for a system is one that, upon being given $Q$, will produce a feasible allocation of $Q$ to $\mathcal{R}$: that is, each task can run in such a way that it will meet its schedule. Such an allocation manager is *adaptive*, since it produces different allocations according to its input, and is designed to be run on-line.

We propose three approaches to verification. In the first approach, we calculate each $Q$ in $\mathcal{Q}$ and run $M$ against it. However, the size of $\mathcal{Q}$ may make this impractical. $M$ here is treated as a black box, but an alternative approach is to model-check the program code of $M$; since model checkers have been able to handle systems with up to $10^{120}$ states, this may be able to handle cases that direct verification cannot. If neither of these is suitable, the third alternative is use symbolic verification via predicate calculus.

## 3 Direct Verification

Given a system, our approach is quite simple.

1. Develop an algorithm to enumerate the elements of $\mathcal{Q}$ based on $q$.
2. Perform the following for each $Q \in \mathcal{Q}$.
   a. Run $M$ on $Q$.
   b. Check that the resulting allocation is feasible.

This approach to verification has many advantages: it is simple, it does not require human intervention beyond the first step, and it can be parallelized to run simultaneously on many different processors. However, the size of $\mathcal{Q}$ can be too large for this to work. For example, given that there are $n$ tasks, a system requirement might be that $\mathcal{Q}$ consists of all sets containing $n/2$ or fewer tasks. In this case, $\mathcal{Q}$ will grow exponentially in the number of tasks. Hence direct verification would be practical for only a limited number of tasks.

A way to reduce the complexity of direct verification is to require that systems have an *incremental* property. Intuitively, the property means that if a set $Q$ of tasks is feasible, then not only is every subset feasible, but so is every other set derived from $Q$ by replacing one task in $Q$ with another that requires the same resources or less. This does not seem to be very restrictive. However, since we have multiple resource classes, not every pair of tasks can be compared for resource usage. For example, if $t$ uses $m$ units of cpu and $n$ units of bandwidth while $t'$ uses $m'$ units of cpu and $n'$ units of bandwidth, where $m < m'$ and $n > n'$ then we cannot say which uses more resources. Instead we define a partial order among tasks s.t. $t <_r t'$ iff for all resource classes $R$, $t.R \leq t'.R$ and for some resource class, $t.R < t'.R$. Here, $t.R$ refers to the amount resources that $t$ requires from $R$. We also define $t =_r t'$ iff for all resource classes $R$, $t.R = t'.R$.

This relation, with the incremental property, can be used to reduce the number of sets that must be checked. For example, if $q$ indicates that $\{T_1, T_2\}$ and $\{T_1, T_3\}$ are required, and if $T_2 \leq_r T_3$, then only $\{T_1, T_3\}$ needs to be checked.

An alternative is to calculate the maximal feasible sets that can be calculated by $M$. By abuse of notation, we can define $Q.R$ as the total amount of resource class $R$ required by tasks in $Q$. Then we can define $Q <_r Q'$ by substituting $Q$ and $Q'$, respectively, in the definition of $t <_r t'$, and similarly for $=_r$.

A set $Q$ is *maximally feasible* if it is feasible and if $Q' >_r Q$ implies that $Q'$ is infeasible. Hence the maximal feasible sets form a boundary between the feasible and infeasible sets of tasks. This set is polynomial in the number of tasks and hence potentially less complex than $\mathcal{Q}$. Below we discuss how to calculate this collection of maximal feasible sets. Once we have it, the revised procedure is as follows.

1. Develop an algorithm to enumerate the elements of $\mathcal{Q}$ based on $q$.
2. Calculate the maximal feasible sets based on $M$, $\mathcal{F}$.
3. For each $Q \in \mathcal{Q}$, check to be sure $Q \leq_r Q'$ for some $Q'$ in $\mathcal{F}$. Indexing on $\mathcal{F}$ can be used to increase the efficiency of the check.

Thus instead of running $M$ on every element of $\mathcal{Q}$, we run it enough to calculate the maximal feasible sets, and then use a potentially faster check to compare $\mathcal{Q}$ against $\mathcal{F}$. This use of a more efficient decision mechanism means that we can handle larger sizes of $\mathcal{Q}$. In addition, this procedure, by calculating $\mathcal{F}$, gives us the maximal capabilities of $M$ with respect to the host system.

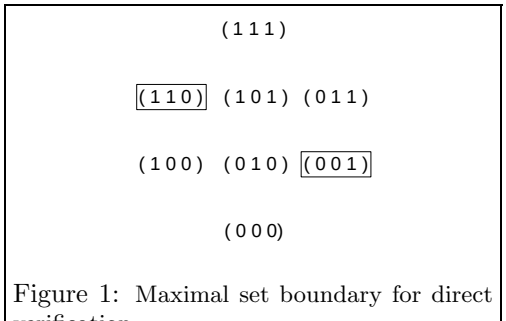

Figure 1: Maximal set boundary for direct verification.

Turning now to the calculation of this boundary, we begin by assuming a single resource class so that the $<_r$ relation becomes a total order. Then we can, without loss of generality, order the tasks into $t_1, \ldots, t_n$ s.t. $t_i \leq_r t_{i+1}$. This means that $\{t_1, t_2\} \leq_r \{t_2, t_3\}$. Now we can form a structure, illustrated in Fig. 1, in which the tasks are indexed, with the tuples representing the presence (1) or absence (0) of the corresponding task. The first line in the diagram gives the set of all tasks; the next line gives the set of tasks

taken two at a time, and so forth. The fact that $<_r$ is a total order means that we can employ binary search on each line to find a feasibility boundary. The maximal feasible sets are marked with a box in the figure.

A problem with the approach is that there can be ambiguous sets. For example, with four tasks, we don't know whether $Q_1 = (\ 1\ 0\ 0\ 1\ )$ is greater than or less than $Q_2 = (\ 0\ 1\ 1\ 0\ )$; consider the situation where $t_1, t_2, t_3, t_4$ use $1, 3, 3, 4$ units of resource, respectively. Then $Q_2 >_r Q_1$. Alternatively, suppose they use $1, 3, 3, 6$ units of resource. Then $Q_1 >_r Q_2$. From our study of ambiguous sets, we are of the opinion there will be at most polynomially many ambiguous alternatives for a given $Q$, but we do not have a proof, and it remains an open question.

# 4 Verification via Model Checking

Model checking is a verification technology that models systems under verification as Kripke structures (that is, finite state machines with labelled states). In the context of these Kripke structures, model checkers can automatically verify properties of interest that are expressed in the form of temporal logic formulas [4]. Since Clarke and Emerson pioneered the field of model checking in the early 1980s [3], there have been significant improvements on the scalability and efficiency of model checking technologies. It is now possible to model check systems with over $10^{120}$ states [2].

In the model checking approach towards the problem of adaptive allocation manager verification, we consider the allocation manager as a reactive system and attempt to leverage on the optimized exhaustive checking capability of model checkers. The key idea is to encode the allocation algorithm in a Kripke structure and express the following assertion in a temporal logic formula: "After a required task set has been provided, eventually the algorithm will produce a feasible allocation for the set." We can employ a model checker to prove this assertion in the context of the Kripke structure representing the allocation algorithm.

Specifically, this approach involve the following steps.

1. Express the feasibility function $F : \mathcal{A} \rightarrow \mathit{Boolean}$, where $\mathcal{A}$ is the set of possible allocations, in first-order predicate logic.

2. Express the adaptive allocation manager in a suitable language such as PROMELA [8] or SMV [11]. The state of interest is a required task set and a feasible allocation, though there will in general also be other state required by the program.

   Note that when a new required task set is introduced, the corresponding allocation in the state will in general be wrong. It is the goal of the program that expresses the allocation manager to eventually reach a state s.t. the allocation is feasible for the required task set.

3. Generate a Kripke structure from the encoded representation of the allocation manager and $\mathcal{Q}$. This step can be performed automatically by tools.

4. Run a model checker and show that the temporal logic formula $\Diamond\, F(A)$, where $A$ is the current allocation, holds on the Kripke structure. The formula means that from every state, $F(A)$ eventually holds; that is, the allocation will eventually become feasible.

When the model checker produces a negative result, it will give a counter example, which can help identify the source of the problem. When the result is positive, we can be confident that the adaptive allocation manager will not lead the system into infeasible allocations.

# 5 Verification via Predicate Calculus

In predicate calculus we trade the complexity of the size of $\mathcal{Q}$ for complexity of another sort: proving symbolically that a candidate $M$ is an adaptive allocation manager. Our approach here is verification by means of weakest precondition [5]. The approach may be sketched as follows.

1. The formula $q$ serves as a pre-condition to $M$.
2. Give a post-condition of feasible allocations: first-order predicate $p$ is satisfied exactly when allocation $A$ is a feasible allocation of input $Q \in \mathcal{Q}$ to resources $\mathcal{R}$.
3. Express the adaptive allocation manager in a suitable language. Dijkstra's simple imperative language is commonly used.
4. Calculate the weakest precondition for $(M, p)$.
5. Prove that $q \implies p$.

Given a Boolean function to indicate feasibility of an allocation, step 2 is easily satisfied. There are computerized tools available to help with steps 4 and 5. For example, VCGen [12] can perform the weakest precondition calculation provided loop invariants are given. Other tools can help calculate the loop invariants (see, for example, [7]). A theorem prover such as PVS [1] can help prove the final implication. In each of these cases, however, manual intervention is typically required.

# 6 Conclusions

We have proposed three approaches to verification of an adaptive allocation manager. The direct approach is simple, but may not be practical due to complexity issues. The model checking approach may handle more complex systems, but could also suffer from complexity issues. The predicate calculus approach is insensitive to the size of $\mathcal{Q}$, but requires more human involvement in the process.

Each approach has both advantages and drawbacks. We believe that all three approaches should be developed as techniques to verify adaptive allocation managers.

# References


[1] Prototype verification system (PVS). http://www.csl.sri.com/projects/pvs/.

[2] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, 1994.

[3] E. M. Clarke, Jr. and E. A. Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science (LNCS), Proceedings of the Workshop on Logics of Programs*, 131:52–71, 1981.

[4] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[5] E. W. Dijkstra. *A Discipline of Programming*. Pearson Education POD, 1997.

[6] K. Ecker, D. Juedes, L. Welch, F. Drews, Chelberg, D. Fleeman, and D. Parrot. An optimization framework for dynamic, distributed real-time system. In *In Proceedings of the 11th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2003)*, 2003.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[8] G. J. Holzmann. *The Spin Model Checker Primer and Reference Manual*. Addison-Wesley, Boston, MA, 2004.

[9] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete qos options. In *IEEE Real Time Technology and Applications Symposium*, pages 276–, 1999.

[10] M. W. Masters. Challenge problem: system certification for real-time systems that employ dynamic resource management. In *Eleventh International Workshop on Parallel and Distributed Real-Time Systems*, 2003.

[11] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[12] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, 1998.