Software Engineering Institute
Carnegie Mellon University

DMPL: Programming and Verifying Distributed Mixed-Synchrony and Mixed-Critical Software

Sagar Chaki
David Kyle

June 2016

TECHNICAL REPORT
CMU/SEI-2016-TR-005

CSC Directorate, Software Solutions Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the
SEI Administrative Agent AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0003343

Table of Contents

Acknowledgments **v**

Abstract **vii**

1 Introduction **1**

2 Related Work **4**

3 An Example DRTS **5**

4 The DMPL Language **7**
4.1 Abstract Syntax 7
4.1.1 Types and Operations 7
4.1.2 Scoping and Assumptions 7
4.2 Program Instance 8
4.2.1 Global Variable Expansion and Ownership 8
4.3 Semantics 8
4.3.1 Variable Valuation 8
4.3.2 Thread Jobs 10
4.3.3 Thread Scheduling 10
4.3.4 External Function Semantics 10
4.3.5 Statement Semantics 10
4.3.6 Thread Semantics 11
4.3.7 Role Semantics 12
4.3.8 Program Instance Semantics 13
4.4 Property Specification 14
4.5 Concrete Syntax 14

5 Code Generation **20**

6 Verification via Sequentialization **23**
6.1 Implementing Sequentialization 23
6.2 Bug Finding and Full Verification 23

7 Evaluation **24**
7.1 Reconnaissance example 24
7.2 Other examples 24

8 Future Work and Conclusion **25**

References/Bibliography **26**

List of Figures

Figure 3.1 Example DRTS with a five-node reconnaissance mission 6

Figure 4.1 Grammar of statements and expressions 8
Figure 4.2 Variables appearing syntactically in expressions 9
Figure 4.3 Evaluation of expressions for node N_i 9
Figure 4.4 Execution semantics of statements and functions for node N_i in program instance P_n 17
Figure 4.5 Two executions of a program instance P_3. Time flows from top to bottom; $r(\widetilde{v}[i], d)$ = value d was read from $\widetilde{v}[i]$; $w(\widetilde{v}[i], d)$ = value d was written to $\widetilde{v}[i]$. The execution on the left is legal even though N_2 reads a stale value d_1, since it observed no previous writes to v. The execution on the right is illegal for two reasons: (i) once N_1 reads value d_2, it can no longer read the older value d_1; and (ii) N_0 must always read the most recent write to v. 18
Figure 4.6 DMPL program for 5-robot reconnaissance example 19

Figure 5.1 Generated C++ code for example DMPL program. In practice, local variables (lines 23–26) are duplicated for each thread. 21

Acknowledgments

We are thankful to Dionisio de Niz, James Edmondson, and Gabriel Moreno for help with ZSRM, MADARA, and self-adaptation, and the rest of the DART team members for many helpful comments and discussions.

Abstract

A language called DMPL for programming distributed real-time, mixed-criticality software is presented. DMPL supports a distributed system in which each node executes a set of periodic real-time threads that are scheduled on the basis of their priority and criticality. Both synchronous and asynchronous threads are allowed. The syntax and semantics of DMPL are formally described. A compiler that generates C++ code from a DMPL program is presented. Two methods of verification of properties of synchronous threads via sequentialization are proposed: fully-automated bounded model checking, and deductive verification with manually-supplied invariants. DMPL programming and verification are validated on several examples of collision avoidance in multi-robot systems.

1 Introduction

Development of verified distributed real-time software (DRTS) is becoming increasingly important. A major driver is the movement of safety-critical real-time embedded domains—e.g., automotive, avionics, medical devices, and robotics—toward more distributed deployments where multiple physically separated nodes communicate and coordinate to provide increased capability, resilience to cyber-attacks, fault-tolerance etc. In other words, the traditional "air gap" between real-time embedded devices is disappearing. For example, there is a push toward autonomous vehicles, and intersection protocols [1] to reduce accidents. Robots are being deployed in teams to achieve more complex tasks with increased fault-tolerance [20]. Even medical devices are being connected in a plug-and-play manner for critical procedures [33].

DRTS brings together two software domains—distributed and real time—each with a rich history for development, validation, and certification. However, they have been studied and developed by different communities of researchers and engineers. Consequently, programming languages have specialized for each domain. For example, languages for distributed systems focus on parallelization and orchestration of tasks and their results, and average case performance. Distributed software is developed using a wide variety of languages, and thread scheduling to meet hard real-time deadlines is not a first class concern. In contrast, real-time software is written largely in C/C++ and executed on real-time operating systems, with explicit control on thread scheduling and worst-case response times to meet hard deadlines in a predictable way. The semantics of communication between nodes is poorly specified and dependent on middleware. Neither approach is sufficient for producing verified DRTS.

Consequently, DRTS is developed today using languages and tools that neither have the right concepts and abstractions nor support rigorous verification. In this technical report, we address this challenge via a new domain-specific programming language, DMPL, for DRTS. DMPL combines concepts from both disciplines—real-time via thread-scheduling and distributed via inter-thread and inter-node communication semantics—in a precise way and supports specification and verification of safety properties.

Roles. A DMPL program specifies a set of roles, each consisting of a set of real-time threads that communicate via shared variables. At runtime, each node of the DRTS executes a specific role. Each thread executes an infinite sequence of jobs (semantically, each job is a procedure) starting at regular time intervals (its period). Each node has its own thread scheduler. DMPL supports mixed-criticality scheduling [3], a form of real-time scheduling that allows threads which perform tasks with different levels of importance (e.g., engine control vs. multimedia player) to share a single CPU under normal and overload operating conditions.

Thread Scheduling. In particular, DMPL uses Zero-Slack Rate Monotonic (ZSRM) scheduling. Specifically, each thread is statically assigned a priority (using Rate-Monotonic assignment, i.e., threads with shorter period have higher priority) and a criticality (based on the importance of the task performed by the thread). We assume that all threads are ZSRM schedulable, i.e.,: (i) under normal conditions, all jobs complete before their deadline (i.e., arrival time of the next job); (ii) under overload conditions, low-criticality jobs are suspended if necessary to enable high-criticality jobs to finish before their deadlines. In practice, this asymmetric temporal protection is implemented by a combination of a static schedulability analysis and a runtime scheduler. The details of ZSRM are available elsewhere [9] and are beyond the scope of this paper.

Communication. DMPL threads (within and across nodes) communicate via shared variables. The state space explosion resulting from variable-based interaction between threads is a major source of complexity for both programming and verification. DMPL uses two complexity-reduction mechanisms to ameliorate this problem. First, each job has a transac-

tional semantics—on arrival, the job reads all shared variables atomically into local copies, then operates on these local copies to compute new values, and finally writes the new values back atomically before terminating. This "read-execute-write" behavior reduces inter-thread interaction and enables us to abstract away details of thread scheduling and network delays to better understand the behavior of the program. It is implemented via locks supported by the ZSRM scheduler.

Synchronization. Second, DMPL supports two kinds of threads—*synchronous* and *asynchronous*—each executing under the corresponding model of computation. Values written to a share variable by jobs of an asynchronous thread are observed by other threads only under the restriction that older writes cannot be observed after newer ones—no other memory consistency is provided. In contrast, a synchronous thread executes in rounds. During each round, one job of the thread executes on each node. Writes to shared variables in round i are guaranteed to be observed during (and only during) round $i + 1$. This restricted communication further reduces the system state space and has been known to reduce verification complexity [22]. Shared variables and synchronous jobs are implemented via the MADARA [12] middleware.

Semantics. When defining the semantics of DMPL, we have to incorporate the effects of both ZSRM scheduling and the synchronous model of computation. Interestingly, both can be modeled by allowing jobs to abort their transactions non-deterministically (i.e., to behave like No Operation (NOP) statements). For ZSRM, a low-priority job that is suspended under overload conditions is semantically a NOP. In the case of the synchronous threads, a job that cannot proceed because other nodes are yet to complete the previous round is semantically a NOP. Note that the transactive nature of jobs provides a natural way to convert them to NOPs. We present this formally later.

Specification and Verification. DMPL also supports specification of safety (i.e., reachability) properties of synchronous threads. Since jobs form the natural unit of computation in DMPL programs, such properties are expressed as *job assertions*: conditions that must hold at the beginning of each job of a synchronous thread. Since DMPL programs are distributed, assertions in DMPL enable us to express conditions over multiple (e.g., pairs of) nodes. This is important to capture requirements that involve more than one node (e.g., collision avoidance). Verification is done through sequentialization. In essence, for the target synchronous thread, we construct a semantically equivalent sequential program which is then verified via existing software verification techniques. Both bounded verification (for bug finding) and deductive verification (for proving correctness) are supported.

Legacy Components. We envision that DMPL will be used to implement the overall concurrency (threading) and communication (shared variables) structure, as well as key protocols and algorithms that must be verified formally. The rest of the system will be implemented by using existing components and libraries (e.g., image processing, planning, sensing, and actuation) To support this mode of development, DMPL threads can invoke external functions that are linked to C++ libraries during compilation. DMPL also supports a limited set of C++ types to ease programming and interaction with such external functions. Note, however, that such interaction is restricted. In particular, the external functions can only return a value that could be stored in a DMPL variable. No indirect modifications of DMPL variables (e.g., through pointers) is allowed. In our experience, this provides a good tradeoff between ease of development, cleanliness of semantics, and soundness of verification.

Code Generation and Validation. We show how to generate C++ code from DMPL programs. We implemented this code generation (and sequentialization for verification purposes) in a compiler called DMPLC. To validate DMPLC, we ran several coordinated multi-robot mission examples on the Linux operating system, implemented a realistic physical environment with the V-REP [29] simulator, and also employed publicly-available implementations of ZSRM [35] and MADARA [21]. Our examples use synchronous threads to implement a collision avoidance protocol and asynchronous threads to perform other mission-relevant tasks (such as path planning

and self-adaptation). We show that bounded model checking is able to detect bugs in incorrect versions of the protocol; deductive verification with manually supplied invariants proves the correct version to be safe.

Non-Goals. Our contribution comes from the perspective of programming language design and implementation (PLDI), following the principle that a well-defined language should drive development and verification. We rely on results from several complementary disciplines. For example, we use ZSRM for schedulability, MADARA for communication and synchronization, sequentialization and software model checking for verification, Linux and V-REP for the platform and environment, and C++ as the target language. Also, we assume that security, networking, performance, timing verification (i.e., schedulability), and similar issues are handled separately. These choices were guided by their natural support for implementing DMPL semantics, public availability, and technical familiarity. We do not claim that they are optimal or that they were made by discarding others. However, they do lead to an effective engineering process for verified DRTS and identify key design decisions that must be made for verified DRTS development.

The rest of this report is organized as follows. In Section 2, we survey related work. In Section 3, we present a DRTS as a motivating example. In Section 4 we present the syntax and semantics of DMPL. In Section 5 we describe how C++ code can be generated from a DMPL program. In Section 6 we present an approach for verifying safety properties over synchronous threads via sequentialization. Finally, in Section 7 we evaluate our approach and in Section 8, we discuss areas for future work and conclude.

2 Related Work

Real-time systems comprise a very large area of research. Broadly, ZSRM [9] falls into the class of preemptive fixed-priority scheduling. This line of work originates as early as Rate-Monotonic scheduling [19] with many subsequent refinements and variations, such as mutual exclusion mechanisms [26]. ZSRM also handles mixed-criticality [30], which is a more recent concept emerging from the need to co-locate multiple software functions with different levels of importance on a single hardware infrastructure to take advantage of features such as more powerful processors. This area has spawned significant research [2, 3].

There is a very wide literature on languages for concurrent and distributed systems, such as CSP [14], CCS [23], and the π-calculus [24]. Most of these languages are targeted towards modeling and verifying concurrent systems (such as FDR [13] for CSP), not code generation. The Occam-π [31] language blends CSP and the π-calculus, and supports code generation and verification. However, it is not targeted toward the development of real-time systems and its semantics has not been connected to periodic tasks with hard deadlines or mixed-criticality. SystemJ [27] is aimed at programming distributed GALS systems. Although threads do not have to be real-time, the system is partitioned into multiple asynchronous clock domains. This means that, unlike DMPL, threads across clock domains cannot execute synchronous protocols. Also, communication is based on message-passing, not shared variables.

Software verification is yet another wide research area [15]. Our work complements verification techniques for sequential C programs since we reduce our verification problem to a sequential one. Sequentialization has been widely used for concurrent program verification. However, most of this work is targeted toward multi-threaded software [18, 28, 7] or real-time software [5] executing on a single processor, not distributed ones. There has also been work on verifying distributed algorithms [16], while our goal is to verify software.

DMPL is inspired by the language DASL [4] and its verification, but generalizes it significantly. DASL allows only one synchronous thread per node, and has no support for mixed-criticality, or deductive verification. DMPL supports multiple synchronous and asynchronous mixed-criticality threads, transactive job semantics, and both bounded and deductive verification, which are critical for real system development. The P language [10] is also a domain specific language developed to facilitate programming and verification. However, it targets asynchronous event driven programs.

We build on the work of Kyle et al. [17] but differ from it substantially. While they presented DMPL in an ad-hoc manner and focused on the specification and verification of probabilistic properties via statistical model checking [34], we give formal syntax and semantics of DMPL, and focus on the specification and verification of non-deterministic behavior via software model checking.

3 An Example DRTS

Consider the distributed system shown in Figure 3.1, which consists of a fleet of five flying robots (or nodes) on a two-dimensional grid. One node has the sensors critical for mission success, and plays the role of the leader. The remaining four nodes provide defensive countermeasures and play the role of protector. Their defenses rely on being positioned between the leader node and any attackers. For the mission to succeed, the leader must follow a specific flight path and reach a particular location by the end of the mission time T.

Protectors best cover the sensor node by flying close to it; however, this raises the chance of collisions. Nodes execute a collision-avoidance protocol. The protocol slows down the overall speed of the fleet. The degree of slowdown depends on the level of communication and coordination needed by the protocol.

We assume that different areas present different levels of attack (i.e., hazards) to the leader. In particular, the map is a 10×10 grid. For the upper right 5×5 cells, we assume high hazards (between 0.5 and 0.9, inclusive, uniformly distributed). For the remaining cells, we assume low hazards (between 0 and 0.4, inclusive, also uniformly distributed). In Figure 3.1, high hazard cells are colored red and low hazard cells are colored gray.

The system has two formations: tight and loose. In tight formation, the protectors are closer to the leader and provide better protection. However, the tight formation moves at about half the speed of the loose formation. The leader executes a simple adaption system which selects which formation to use based on the upcoming hazards and the remaining mission time in an attempt to minimize danger, while maintaining a high probability of mission completion by time T.

The DMPL program for this DRTS consists of two roles—leader and protector. The runtime instance of the program corresponding to the system consists of one node executing the leader role, and the remaining four nodes executing the protector role. The leader node executes three threads: (i) a synchronous thread with high criticality and low-priority that implements the collision avoidance protocol; (ii) an asynchronous thread with low criticality and high priority to run a path planning algorithm and compute the next waypoint; and (iii) an asynchronous thread with low criticality and medium priority to run the adaptation algorithm and decide when a formation change is needed. The adaptation algorithm itself is implemented in a separate component, and invoked through an external function call. Each protector node runs only two threads—the synchronous collision avoidance and the asynchronous path planning. On each node, the path planning thread communicates with the collision avoidance thread by updating the next waypoint coordinate. In addition, the adaptation thread on the leader node communicates with the waypoint thread (on all nodes) by updating a shared variable indicating the formation type. Note that all features of DMPL are required to implement this system.

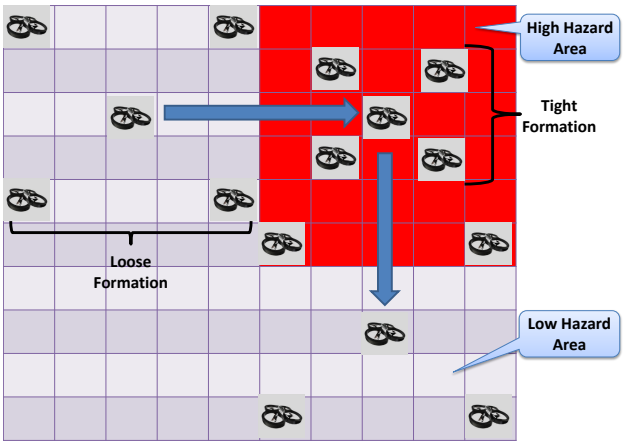

Figure 3.1: Example DRTS with a five-node reconnaissance mission

4 The DMPL Language

A DMPL program is comprised of a triple (GV, LV, R), where GV is a set of global variables, LV is a set of local variables, and R is a set of roles. A role is comprised of a pair (S, A) where S is a set of synchronous threads, and A is a set of asynchronous threads. A thread is comprised of a triple (π, κ, f) where π is its period, κ is its criticality, and f is its job-function that is executed in each period. The body of f is a set of statements that accesses local, global, temporary and id variables.

DMPL supports calls to two types of functions: (i) internal functions declared as part of the DMPL program; and (ii) external C++ functions. External functions operate on an external state and cannot modify DMPL variables. However, they can return values that can be used by their caller internal function to update DMPL variables. The job function of a thread must be internal. The set of all internal and external functions are denoted *IntFn* and *ExtFn*, respectively. We write *Func* to mean the set of all functions, i.e., $Func = IntFn \cup ExtFn$.

4.1 Abstract Syntax

Let TV be a set of temporary variables, IV be a set of id variables, and id be a distinguished variable such that GV, LV, TV, IV and $\{id\}$ are mutually disjoint. The body of a function is a statement. The abstract syntax of statements, lvalues and expressions is given by the BNF grammar in Figure 4.1. Intuitively, *skip* is a NOP, $l = e$ is an assignment, ITE is an if-then-else statement, WHILE is a while loop, $\text{ALL}(v, st)$ executes st iteratively by substituting v for the ID of each node, $\langle st_1 ; \ldots ; st_k \rangle$ executes st_1 through st_k in sequence, $\nu(v, st)$ introduces a fresh temporary variable v in scope of st, $\sim$ is an unary operator, and $\diamond$ is a binary operator. ALL enables iteration over all node IDs without knowing the exact number of such IDs a priori. In addition, the special expression $\odot$ denotes the `void` value. An internal function f with k arguments is a triple (p, b, r) where $p \in TV^k$ is the list of its arguments, $b \in stmt$ is a statement denoting its body, and $r \in exp$ is an expression denoting its return value.

4.1.1 Types and Operations

All DMPL variables are typed. These types and operations on them have the same interpretation as in C++. The following numeric types are supported: `int` and `char` (both `signed` and `unsigned` variants), `double`, `bool`. Array types are also allowed. Given a type τ and a positive integer n, the array type with n elements of type τ is denoted by $\tau[n]$. Only numeric and bitwise operations over the supported types are allowed (e.g., the address of a variable cannot be taken). The `void` type is allowed as the return type of functions. The set of all types is denoted $\mathbb{T}$. The type of variable v is denoted $\texttt{typ}(v)$.

4.1.2 Scoping and Assumptions

We assume that: (i) variables in $GV \cup LV \cup \{id\}$ are always in scope; (ii) for each statement $\text{ALL}(v, st)$ and $\nu(v, st)$, variable v is in scope of st; (iii) scoping is unambiguous, and only variables in scope are used in expressions; (iv) id and id variables do not appear on the LHS of assignments, i.e., they are read-only; (v) functions are called with arguments of proper type, and their return values (if any) are also stored into variables of proper type, where "proper" means allowed by the semantics of C++. Note that these assumptions do not limit expressivity.

$$
\begin{array}{rcl}
\textbf{(Statements)}\ stmt & := & skip \mid lval = exp \\
& \mid & \mathsf{ITE}(exp, stmt, stmt) \\
& \mid & \mathsf{WHILE}(exp, stmt) \\
& \mid & \mathsf{ALL}(IV, stmt) \\
& \mid & \langle stmt\ ;\ldots;\ stmt\rangle \\
& \mid & Func(exp, \ldots, exp) \\
& \mid & TV = Func(exp, \ldots, exp) \\
& \mid & \nu(TV, stmt) \\
\textbf{(LValues)}\ lval & := & GV \mid LV \mid TV \\
\textbf{(Expressions)}\ exp & := & \mathbb{Z} \mid lval \mid GV@IV \\
& \mid & id \mid IV \mid\sim exp \mid exp \diamond exp
\end{array}
$$

Figure 4.1: Grammar of statements and expressions

4.2 Program Instance

An instance of a DMPL program P of size n, denoted P_n, is a finite mapping from ids in the range $[0, n)$ to roles. This induces a mapping from ids to threads in a natural manner, if $P_n(i) = (S, A)$, then $S(i) = S$ and $A(i) = A$. In essence, P_n is a collection of n nodes, each with a distinct id in the range $[0, n)$ where the node with id i, denoted N_i, executes the role $P_n(i)$.

4.2.1 Global Variable Expansion and Ownership

Nodes use global variables to communicate. DMPL enforces every global variable to have a single writer node, as follows. In the program instance P_n, each global variable v of type τ is expanded to an array $\widetilde{v}$ of type $\tau[n]$. Then, node N_i only writes to the element $\widetilde{v}[i]$, but reads from all elements of $\widetilde{v}$. We write $\widetilde{GV}$ to mean the set of all expanded global variables, i.e.,

$$\widetilde{GV} = \{\widetilde{v} \mid v \in GV\}$$

As we shall see later, this restriction will be useful to define the semantics of synchronous threads.

4.3 Semantics

Semantically, the program instance P_n is a distributed system with n communicating nodes. We define this semantics formally in stages, starting with functions and threads, then proceeding to individual nodes, and finally defining the semantics of a complete program instance. The set of values over the supported types $\mathbb{T}$ is denoted D. The type of a value d is denoted $\mathsf{typ}(d)$.

4.3.1 Variable Valuation

Given a set of variables V, let $\mathcal{V}(V)$ be the set of *partial* mappings from V to values of appropriate type. In other words:

$$\forall s \in \mathcal{V}(V)\,.\,\forall v \in Dom(s)\,.\,\mathsf{typ}(v) = \mathsf{typ}(s(v))$$

$$Var(e) = \begin{cases} \{\widetilde{e}\} & \text{if } e \in GV \\ \{e\} & \text{if } e \in LV \cup TV \cup IV \cup \{id\} \\ \{\widetilde{v_1}, v_2\} & \text{if } e \doteq v_1 @ v_2 \in GV @ IV \\ Var(e') & \text{if } e \doteq\sim e' \\ Var(e_1) \cup Var(e_2) & \text{if } e \doteq e_1 \diamond e_2 \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 4.2: Variables appearing syntactically in expressions

$$e \triangleright s = \begin{cases} z & \text{if } z \in \mathbb{Z} \\ s(\widetilde{e})[i] & \text{if } e \in GV \\ s(e) & \text{if } e \in LV \cup TV \cup IV \cup \{id\} \\ s(\widetilde{v_1})[s(v_2)] & \text{if } e \doteq v_1 @ v_2 \in GV @ IV \\ \sim (e' \triangleright s) & \text{if } e \doteq\sim e' \\ (e_1 \triangleright s) \diamond (e_2 \triangleright s) & \text{if } e \doteq e_1 \diamond e_2 \end{cases}$$

Figure 4.3: Evaluation of expressions for node N_i

For any non-`void` type τ, the initial value $I(\tau)$ is 0 cast to τ as per the semantics of C++. Let $\mathcal{I}(V) \in \mathcal{V}(V)$ be the special initial mapping that maps each variable $v \in V$ of type τ to $I(\tau)$. We use the terms *state* and *valuation* interchangeably.

Let $Var(e)$ denote the set of variables appearing syntactically in expression e, as defined in Figure 4.2. Let e be an expression and s be a state such that $Var(e) \subseteq Dom(s)$. The evaluation of e under s, denoted $e \triangleright s$, is defined inductively over the structure of e in a natural manner, as shown in Figure 4.3. Note that operators are interpreted as in C++, and global variables evaluate to the i-th element of their expanded version. Also note that $e \triangleright s$ is well-defined under our assumption $Var(e) \subseteq Dom(s)$ since we always apply the mapping s to a variable in its domain. We write $s \models e$ to mean that the value $e \triangleright s$ would be converted to TRUE under the Boolean conversion rules of C++. The `void` expression $\odot$ evaluates to itself under all states, i.e., $\odot \triangleright s = \odot$. If an array is read out of bounds, the result is 0 cast to the type of the array element. If an array is written out of bounds, it is a NOP.

We define three operations on states: (i) *Update*: given a state s, variable v, and value d, the updated state $s[v \mapsto d]$ is identical to s except that it maps v to d; (ii) *Extension*: given a state s and a variable $v \notin Dom(s)$ of type τ, $s \oplus v$ is that state such that:

$$Dom(s \oplus v) = Dom(s) \cup \{v\}$$
$$\forall v' \in Dom(s) \centerdot (s \oplus v)(v') = s(v')$$
$$(s \oplus v)(v) = I(\tau)$$

and (iii) *Restriction*: given a state s and a variable $v \in Dom(s)$, $s \ominus v$ is that state such that:

$$Dom(s \ominus v) = Dom(s) \setminus \{v\}$$
$$\forall v' \in Dom(s \ominus v) \centerdot (s \ominus v)(v') = s(v')$$

Thus, $s \oplus v$ extends s with a new variable v set to its initial value, while $s \ominus v$ projects away v. We write $s \oplus (v, d)$ to mean the state $(s \oplus v)[v \mapsto d]$, i.e., the state obtained from s by adding a new variable v initialized to value d. We use state update to handle assignments, and state extension and restriction to handle temporary variables. Given a state s and a set of variables V, we obtain $s \setminus V$ by removing from s mappings for variables in V and $s \cap V$ by removing from s mappings for variables not in V. Given two states s_1 and s_2 such that $Dom(s_1) \cap Dom(s_2) = \emptyset$,

the state $s_1 \oplus s_2$ denotes their union:

$$Dom(s_1 \oplus s_2) = Dom(s_1) \cup Dom(s_2)$$
$$\forall i \in [1,2] \centerdot \forall v \in Dom(s_i) \centerdot (s_1 \oplus s_2)(v) = s_i(v)$$

4.3.2 Thread Jobs

Threads in DMPL execute periodically in a read-execute-write manner. Semantically, a thread $t = (\pi, \kappa, f)$ consists of an infinite sequence of jobs that activate at times $0, \pi, 2\pi, \ldots$ etc. Each job atomically copies local and global variables at the beginning into temporary variables, then operates over these temporary variables to update their values by executing function f, and finally atomically writes the new values back to the corresponding local and global variables. Note that due to thread scheduling, a job may not start execution immediately on activation. It may also be preempted by other jobs during execution.

4.3.3 Thread Scheduling

All DMPL threads are real-time—that is, they have worst case execution times (WCETs) and deadlines. We assume that threads which belong to a node execute on a single CPU. Threads are scheduled based on their priorities (π) and criticalities (κ) using Zero-Slack Rate Monotonic (ZSRM) scheduling [9]. At a high level, this means that threads with higher priority are preemptively allocated the CPU during normal operating conditions, while threads with higher criticality are preemptively allocated the CPU during overload operating conditions. We assume that threads are *schedulable*—that is, a job that starts at time $i \times \pi$ completes execution before time $(i+1) \times \pi$ under the worst case interference from the other threads. This is achieved by using an offline ZSRM schedulability analysis [9]. WCET estimation is a challenging problem in its own right [32], especially in the presence of caches and other components. As is common in schedulability, we assume that WCET estimation is handled through a separate procedure that covers both code generated from DMPL and external libraries.

4.3.4 External Function Semantics

For each external function f with k parameters, we assume a semantics $[\![f]\!] \subseteq D^k \times D$. Formally, $(d_1, \ldots, d_k, d) \in [\![f]\!]$ if and only if the invocation of f with arguments $d_1, \ldots, d_k$ could return with value d.

4.3.5 Statement Semantics

Let $\mathbb{V} = \widetilde{GV} \cup LV \cup TV \cup IV \cup \{id\}$ be the set of all variables, and $\mathbb{V}_s = \widetilde{GV} \cup LV$ be the set of shared (i.e., global and local) variables. The semantics of a statement st is given by the way it updates the values of variables, and the shared variables to which it writes. Formally, it is the relation $[\![st]\!] \subseteq \mathcal{V}(\mathbb{V}) \times \mathcal{V}(\mathbb{V}) \times 2^{\mathbb{V}_s}$ defined inductively over structure of st. Let us write $\{s\}[\![st]\!]\{s', w\}$ to mean $(s, s', w) \in [\![st]\!]$. Then, $[\![\cdot]\!]$ is the smallest relation satisfying the rules in Figure 4.4. A high-level description of the proof-rules is as follows:

- SKIP: The *skip* statement has no effect on the state.
- ASGN-GLOBAL: Assignment to a global variable v is semantically an assignment to the i-th element of its expanded version $\widetilde{v}$.

- ASGN-LOC and ASGN-TMP: Assignments to local and temporary variables update the lhs based on the evaluation of the rhs in the current state.
- ITE-IF and ITE-THEN: If-then-else statements execute the appropriate branch based on the evaluation of their condition in the current state.
- WHILE-EXIT and WHILE-LOOP: A while loop either terminates immediately if the loop condition fails, or executes the body once and repeats.
- ALL: The iteration statement executes the body repeatedly, assigning values $[0, n)$ to the fresh id variable.
- SEQ: A sequence of statements operates as expected, updating the state by executing its components one after the other.
- EXT-FUNC1 and EXT-FUNC2: External functions optionally return a value which is stored in a temporary variable.
- INT-FUNC1 and INT-FUNC2: Internal functions update local and global variables only. They also optionally return a value which is stored in a temporary variable.
- NEW-VAR: Fresh variables exist only within the scope of their definition.
- FUNCTION: The arguments of a function determine its semantics. From the perspective of its caller, a function updates local and global variables and returns a value.

4.3.6 Thread Semantics

Each thread $t = (\pi, \kappa, f) \in S \cup A$ executes an infinite sequence of jobs $j_0, j_1, \ldots$, with each job having an arrival time (when it becomes active) and a departure time (when it completes execution of its function or aborts). In addition, each job j_i reads all shared variables before executing f, and writes the updated values of shared variables back after executing f. Recall that $\mathbb{V}_s = \widetilde{GV} \cup LV$ is the set of shared (global and local) variables. We denote the arrival of a job with the symbol $\uparrow$, the departure of a job with the symbol $\downarrow$, and the abort of a job with the symbol $\dagger$. Thus, an execution of t is an infinite sequence:

$$\begin{array}{ll} \langle & (0, \uparrow, c_0, s_0, \emptyset), (0, e_0, c'_0, s'_0, w'_0), \\ & (1, \uparrow, c_1, s_1, \emptyset), (1, e_1, c'_1, s'_1, w'_1), \quad \ldots \rangle \end{array}$$

such that the following conditions hold:

- Events, timestamps, and states have the correct type:

$$\forall i \geq 0 \centerdot e_i \in \{\downarrow, \dagger\} \wedge c_i, c'_i \in \mathbb{R} \wedge s_i, s'_i \in \mathcal{V}(\mathbb{V}_s) \wedge w'_i \subseteq \mathbb{V}_s$$

- Jobs arrive periodically, require positive time to execute, and timestamps are non-decreasing:

$$\forall i \geq 0 \centerdot c_i \in [i \times \pi, (i+1) \times \pi) \wedge c_i < c'_i \wedge c'_i \leq (i+1) \times \pi$$

- The state is properly initialized: $s_0 = \mathcal{I}(\mathbb{V}_s)$.
- If a job aborts, it leaves the state unchanged. Otherwise, it updates the state according to the semantics of its function:

$$\begin{array}{ll} \forall i \geq 0 \centerdot & (e_i = \downarrow \implies \{s_i\}[\![f]\!]\{s'_i, w'_i\}) \wedge \\ & (e_i = \dagger \implies (s'_i = s_i \wedge w'_i = \emptyset)) \end{array}$$

Intuitively, job j_i starts executing at time c_i, reads state s_i, and either updates it to s'_i by executing function f, or aborts and leaves the state unchanged, and finally departs at time c'_i. Moreover, the first job reads the initial values of all variables. The semantics of thread t, denoted $[\![t]\!]$, is the set of all its executions.

4.3.7 Role Semantics

Next, we combine the executions of all threads into an execution of the role. In doing so, we must ensure that threads are scheduled correctly based on their priorities, and that states are observed and updated in a sequentially consistent manner, i.e., when a job arrives, its reads the most recently written values of *local* variables. Note that we do not restrict values of global variables since these are shared across nodes and hence their values are determined by the inter-node communication. Consider role $\rho = (S, A)$. An execution of ρ is an infinite sequence:

$$\eta = \langle (\iota_0, e_0, c_0, s_0, w_0, t_0), (\iota_1, e_1, c_1, s_1, w_1, t_1), \ldots \rangle$$

such that the following conditions hold:

- Each $t_i \in S \cup A$ is a thread.
- For each thread $t \in S \cup A$, the projection of η on t, i.e., the subsequence of η consisting of elements with $t_i = t$, is an execution of t.
- Timestamps are non-decreasing, i.e., $\forall i \geq 0 \centerdot c_i \leq c_{i+1}$.
- A lower priority thread t_l can only preempt a higher priority thread t_h when it is in overload and has higher criticality than t_h. In ZSRM, a thread is said to be in overload if it has an active job and the current time is no less than the job's zero-slack instant (ZSI). Informally, the ZSI of a job is the latest time at which it can be started (or resumed) so that it completes its remaining execution before its deadline. Given a thread t and a job index ι, we write $ZSI(t, \iota)$ to denote the ZSI of the ι-th job of t. We note that $ZSI(t, \iota)$ is statically computable via the ZSRM schedulability analysis. Thus, whenever η has a sub-sequence:
  $$\langle (\iota_1, e_1, c_1, s_1, t_1), (\iota_2, e_2, c_2, s_2, t_2), (\iota_3, e_3, c_3, s_3, t_3) \rangle$$
  such that:
  $$t_1 = t_3 \wedge \iota_1 = \iota_3 \wedge \pi(t_1) > \pi(t_2)$$
  then the following must be true:
  $$\kappa(t_2) > \kappa(t_1) \wedge c_2 \geq ZSI(t_1, \iota_1)$$
- Each arriving job reads the value of local variables written by the most recently successfully completed job. For each $i \geq 0$, and $v \in LV$, let $PrWr(i, v)$ be the set of indices corresponding to previous successfully completed jobs that wrote to v, i.e.,
  $$PrWr(i, v) = \{j < i \mid e_j = \downarrow \wedge v \in w_j\}$$
  Then:
  $$\forall i \geq 0 \centerdot e_i = \uparrow \implies \forall v \in LV \centerdot s_i(v) = s_m(v),$$
  $$\text{where } m = \max(\{0\} \cup PrWr(i))$$
  Note that in the absence of prior writes to v, m evaluates to 0, and the initial value of v is read.

4.3.8 Program Instance Semantics

Finally, we obtain the semantics of a program instance by combining the semantics of all the roles. In doing so, we must ensure that the values of global variables are propagated across nodes in a consistent manner. Since DMPL nodes communicate over unreliable networks with no synchronized clocks and no guaranteed order of message delivery, we do not have any total sequential consistency between nodes in terms of how global variables are modified and observed. Informally, when a job reads a global variable v, it must observe a value no earlier than the last time it read v. However, if the reader and writer nodes are the same, the most recent write is observed since no network messages are involved in this case. Figure 4.5 shows two executions to highlight this situation. Formally, consider a program instance P_n. Recall that P_n is a mapping from $[0, n)$ to roles. An execution of P_n is an infinite sequence

$$\eta = \langle(\iota_0, e_0, c_0, s_0, w_0, t_0, \rho_0), (\iota_1, e_1, c_1, s_1, w_1, t_1, \rho_1), \ldots\rangle$$

such that the following conditions hold:

- Each ρ_i is a role.
- For each role $\rho \in R$, the projection of η on ρ, i.e., the subsequence of η consisting of elements with $\rho_i = \rho$, is an execution of ρ.
- Each arriving job reads the value of a global variable that is no older than the last value that was read. Formally, this means that for each global variable $\widetilde{v} \in \widetilde{GV}$, each writer node $N_w, w \in [0, n)$, and each reader node $N_r, r \in [0, n)$, there is a *monotonic* mapping $\Phi(\widetilde{v}, w, r) : \mathbb{N} \mapsto \mathbb{N}$ from each position where a value of $\widetilde{v}[w]$ is read by N_r to the position where the corresponding value was written by N_w. Thus:

  $$\begin{aligned}&\forall i \in \mathbb{N} \centerdot \rho_i = P_n(r) \wedge e_i = \uparrow \implies \\ &\quad \rho_{\Phi(i)} = P_n(w) \wedge e_{\Phi(i)} = \downarrow \wedge s_i(\widetilde{v})[w] = s_{\Phi(i)}(\widetilde{v})[w]\end{aligned}$$

  The mapping is monotonic, thus older writes are never observed after newer ones:

  $$\forall i < i' \centerdot \rho_i = \rho_{i'} = P_n(r) \wedge e_i = e_{i'} = \uparrow \implies \Phi(i) \leq \Phi(i')$$

  Note that the mapping $\Phi(\widetilde{v}, w, r)$ varies with $\widetilde{v}$, w and r. Thus, due to network delays, writes to the same variable could be observed in different order by different readers, and writes to different variables could be observed by the same reader in different order. However, if the reader and writer nodes are the same, then there are no network messages involved, and the most recent write is observed (as for local variables). Formally, for each $i \geq 0$, global variable $\widetilde{v} \in \widetilde{GV}$, and node N_r, let $PrWr(i, \widetilde{v}, r)$ be the set of indices corresponding to previous successfully completed jobs of N_r that wrote to $\widetilde{v}[r]$, i.e.,

  $$PrWr(i, \widetilde{v}, r) = \{j < i \mid e_j = \downarrow \wedge \widetilde{v}[r] \in w_j\}$$

  Then:

  $$\begin{aligned}&\forall i \in \mathbb{N} \centerdot \rho_i = P_n(r) \wedge e_i = \uparrow \implies \\ &\quad \Phi(\widetilde{v}, r, r)(i) = \max(\{0\} \cup PrWr(i, \widetilde{v}, r))\end{aligned}$$

  Note that in the absence of prior writes to $\widetilde{v}[r]$, $\Phi(\widetilde{v}, r, r)(i)$ evaluates to 0, and the initial value of $\widetilde{v}[r]$ is observed.

- Each synchronous thread t executes in rounds. Let $S(t)$ be the ids of the nodes that execute t, i.e.,

  $$S(t) = \{j \mid P_n(j) = (S, A) \wedge t \in S\}$$

For each round $i \geq 0$, and node id $j \in S(t)$, let $Round^+(t,i,j)$ be the index in η corresponding to the start of the i-th job of thread t in node N_j that completes successfully, and $Round^-(t,i,j)$ be the index in η corresponding to the end of the i-th job of thread t in node N_j that completes successfully. Then:

$$\forall j' \in S(t) \centerdot Round^-(t,i,j') < Round^+(t,i+1,j)$$

Moreover, the value of a global variable read at the beginning of round $i+1$ equals the value written at the end of round i. Recall the definition of $\Phi(\widetilde{v},w,r)$. Thus:

$$\Phi(\widetilde{v},w,r)(Round^+(t,i+1,r)) = Round^-(t,i,w)$$

4.4 Property Specification

As mentioned, we have explored verification of properties implemented by synchronous threads. Such properties are expressed as round invariants: conditions that must hold at the beginning of each round of execution of a synchronous thread. Formally, a specification is a pair (t,f) where t is a synchronous thread, and f is an internal function that evaluates the round invariant, and returns it as a `bool`. Internal functions used in properties (a.k.a. property functions) are distinguished from other internal functions in two ways:

1. They do not modify any shared variables.
2. They can refer to local variables of arbitrary nodes. This is necessary because a property function has to evaluate relations over the states of multiple nodes. For instance, in the case of collision avoidance in our example, the property function checks that every pair of distinct nodes occupies different cells (i.e., each pair has different values of `x` or `y`).

Property functions are evaluated over program instance states. A program instance state is a pair (s_l, s_g) such that $s_l : [0,n) \mapsto \mathcal{V}(LV)$ and $s_g \in \mathcal{V}(\widetilde{GV})$. Informally, s_l maps each node id i to the value of local variables at N_i, while s_g is a mapping from global variables to their values. In general, since P_n is distributed, it may not have a well-defined program instance state at all times during an execution. In particular, the value of a global variable may be different at different nodes. However, at the beginning of each round of a synchronous thread, there is a unique program instance state since all nodes must agree on the value of each global variable. We denote by $\hat{S}(\pi,t,i)$ the program instance state at the beginning of round i of t in execution η.

Given a program instance state $\hat{s} = (s_l, s_g)$, and a property function f, we write $f(\hat{s})$ to denote the value returned by f when invoked from $\hat{s}$. This is defined similarly to a normal internal function, except that $s_l[i]$ is used to lookup the value of local variables for N_i, while s_g is used to lookup the value of global variables. Finally, we say that a program instance P_n satisfies a specification (t,f), denoted $P_n \models (t,f)$, if for every execution η of P_n, we have:

$$\forall i \geq 0 \centerdot f(\hat{S}(\eta,t,i)) = \top$$

4.5 Concrete Syntax

The concrete syntax of DMPL consists of declarations for GV and LV plus definitions of functions, threads, roles, and properties. For example, Figure 4.6 shows a fragment of the DMPL program corresponding to our example described in Chapter 3. Note the following:

- The syntax is similar to C++. This provides familiarity to practitioners and simplifies code generation and sequentialization.
- Platform-specific code is included at the top inside `%%{ ... %%}` blocks. This feature is used to include code (header files, helper functions, etc.) needed to generate compilable C++. This code is not subjected to any analysis. It is emitted verbatim during code generation. The programmer should ensure that this code does not conflict with the code generated from the rest of the DMPL program (e.g., no variable name clashes) or cause other compilation problems (e.g., syntax errors). This code is ignored during verification. In Figure 4.6, this code is used to initialize the hazard levels in each grid cell randomly, and to periodically access hazard levels and compute the total exposure faced by the leader at runtime.
- Constant definitions (lines 14–15) are allowed for readability.
- Multi-dimensional array variables are supported.
- External functions are declared using the `extern` keyword. External functions can also be labeled `pure` (e.g., line 19). This is a hint that the function should not modify the external system state. It is not checked or enforced in any way. In our case, we expect that the `GET_HAZARD()` function should be read-only. External functions are also declared or defined in C++ within the code included via the `%%{...%%}` block.
- Shared variables can be initialized to non-zero values. This is done either via direct assignment (line 26), or via constructors (line 34).
- Shared variables can also be labeled as `input` and constraints can be imposed on their initial values (lines 28–29). For example, variable `x` is constrained to be between `0` and `X-1` initially. Input variables are treated dually as follows: (i) the generated code allows their initial values to be specified from the command line, but performs a runtime check that any such value respects specified constraints; and (ii) during verification, their initial values are assumed to be non-deterministic subject to the constraints. This ensures that our verification is sound.
- Threads are defined with the `thread` keyword. The `@BarrierSync` annotation indicates a synchronous thread. Other annotations (`@Period` for period, `@Criticality` for criticality, `@WCExecTimeNominal` for WCET under normal execution, `@WCExecTimeOverload` for WCET under overload) are used to specify parameters used for thread schedulability analysis and scheduling. Time parameters are specified in microseconds.
- Iterators are available to: (i) execute a statement over all nodes (`forall_node`), all pairs of distinct nodes (`forall_distinct_node_pair`), etc.; and (ii) evaluate an expression disjunctively over nodes that have a lower id (`exists_lower`), a higher id (`exists_higher`), etc. They are all "syntactic sugar" defined formally using ALL in a natural manner.
- A role can use threads defined in the global scope. For example, the `COLLISION_AVOIDANCE` thread is used by both the `Leader` (line 71) and `Protector` (line 90) roles. A role can redefine timing parameters of imported threads. For example, the `Leader` roles redefines the WCET estimates (lines 73–74) of the `WAYPOINT` thread. A role can also define new threads. For example, the `ADAPTATION_MANAGER` thread is defined by the `LEADER` roles, since the functionality implemented by this thread is required only by the leader node.
- A role can redefine (or `override`) the initialization of shared variables. For example, the `Leader` role redefines variables `xt` and `yt` to be inputs. This is necessary since the final target cell of the leader is part of the mission parameters and is supplied as as input to the system. Note that these variables are initialized to the default value 0 (line 30) for the `Protector` role. This makes sense since the target cell for a protector changes over time

as the leader moves because each protector tries to move to a specific position relative to the leader in order to maintain the overall formation (cf. Figure 3.1).

- A role can `override` the definition of an internal function, and invoke the `base` function if needed. For example, the `Leader` role overrides the `REACHED_NEXT_XY` function (line 66) since the leader must execute additional code (compared to a protector) once it reaches the next waypoint. It invoked the base definition of `REACHED_NEXT_XY` (line 68) to implement functionality common with the protectors.
- Internal functions can be labeled `pure` (e.g., `NoCollisions` at line 98). Such functions may not modify shared variables. This is checked by the compiler, and compilation aborts if a violation is detected. Since there is no aliasing in DMPL, this check simply ensures that no shared variables are on the lhs of assignments in the body of the function, and that all internal functions invoked by this function are also labeled `pure`. Recall that external functions cannot modify shared variables indirectly.
- Properties are defined via the `require` keyword (line 107). The property function (in this case `NoCollisions`) must be `pure` to ensure that it has no side effects on the program state. Note that in our example, the property function does indeed refer to local variables `x` and `y` of multiple nodes.

Since DMPL is an experimental language, the concrete syntax presented here is not final. However, the concepts presented above should persist since they emerged from the need to program a realistic system, and we expect them to be useful for programming DRTS in general.

$$\frac{}{\{s\}[\![skip]\!]\{s,\emptyset\}}\ \text{[SKIP]} \qquad \frac{v \in GV \qquad w = \{\widetilde{v}[i]\}}{\{s\}[\![v = e]\!]\{s[\widetilde{v}[i] \mapsto e \triangleright s], w\}}\ \text{[ASGN-GLOBAL]}$$

$$\frac{v \in LV \qquad w = \{v\}}{\{s\}[\![v = e]\!]\{s[v \mapsto e \triangleright s], w\}}\ \text{[ASGN-LOC]} \qquad \frac{v \in TV}{\{s\}[\![v = e]\!]\{s[v \mapsto e \triangleright s], \emptyset\}}\ \text{[ASGN-TMP]}$$

$$\frac{s \models e \qquad \{s\}[\![st_1]\!]\{s', w\}}{\{s\}[\![\mathsf{ITE}(e, st_1, st_2)]\!]\{s', w\}}\ \text{[ITE-THEN]} \qquad \frac{\neg(s \models e) \qquad \{s\}[\![st_2]\!]\{s', w\}}{\{s\}[\![\mathsf{ITE}(e, st_1, st_2)]\!]\{s', w\}}\ \text{[ITE-ELSE]}$$

$$\frac{\neg(s \models e)}{\{s\}[\![\mathsf{WHILE}(e, st)]\!]\{s, \emptyset\}}\ \text{[WHILE-EXIT]}$$

$$\frac{s \models e \qquad \{s\}[\![st]\!]\{s', w'\} \qquad \{s'\}[\![\mathsf{WHILE}(e, st)]\!]\{s'', w''\} \qquad w = w' \cup w''}{\{s\}[\![\mathsf{WHILE}(e, st)]\!]\{s'', w\}}\ \text{[WHILE-LOOP]}$$

$$\frac{\begin{array}{c} v \notin Dom(s) \qquad s_0 = s \oplus v \\ \{s_0[v \mapsto 0]\}[\![st]\!]\{s_1, w_1\}\ \ \{s_1[v \mapsto 1]\}[\![st]\!]\{s_2, w_2\} \ldots \{s_{n-1}[v \mapsto n-1]\}[\![st]\!]\{s_n, w_n\} \\ s' = s_n \ominus v \qquad w = w_1 \cup \cdots \cup w_n \end{array}}{\{s\}[\![\mathsf{ALL}(v, st)]\!]\{s', w\}}\ \text{[ALL]}$$

$$\frac{\{s_0\}[\![st_1]\!]\{s_1, w_1\} \ldots \{s_{k-1}\}[\![st_k]\!]\{s_k, w_k\} \qquad w = w_1 \cup \cdots \cup w_k}{\{s_0\}[\![\langle st_1; \ldots; st_k \rangle]\!]\{s_k, w\}}\ \text{[SEQ]}$$

$$\frac{f \in ExtFn \qquad (e_1 \triangleright s, \ldots, e_k \triangleright s, d) \in [\![f]\!]}{\{s\}[\![f(e_1, \ldots, e_k)]\!]\{s, \emptyset\}}\ \text{[EXT-FUNC1]}$$

$$\frac{f \in ExtFn \qquad (e_1 \triangleright s, \ldots, e_k \triangleright s, d) \in [\![f]\!] \qquad s' = s[v \mapsto d]}{\{s\}[\![v = f(e_1, \ldots, e_k)]\!]\{s', \emptyset\}}\ \text{[EXT-FUNC2]}$$

$$\frac{\begin{array}{c} f \in IntFn \qquad d_1 = e_1 \triangleright s, \ldots, d_k = e_k \triangleright s \\ s_1 = s \setminus TV \qquad \{s_1\}[\![f(d_1, \ldots, d_k)]\!]\{s_2, d, w\} \qquad s' = s_2 \oplus (s \cap TV) \end{array}}{\{s\}[\![f(e_1, \ldots, e_k)]\!]\{s', w\}}\ \text{[INT-FUNC1]}$$

$$\frac{\begin{array}{c} f \in IntFn \qquad d_1 = e_1 \triangleright s, \ldots, d_k = e_k \triangleright s \\ s_1 = s \setminus TV \qquad \{s_1\}[\![f(d_1, \ldots, d_k)]\!]\{s_2, d, w\} \qquad s' = s_2 \oplus (s \cap TV)[v \mapsto d] \end{array}}{\{s\}[\![v = f(e_1, \ldots, e_k)]\!]\{s', w\}}\ \text{[INT-FUNC2]}$$

$$\frac{v \notin Dom(s) \qquad s_1 = s \oplus v \qquad \{s_1\}[\![st]\!]\{s_2, w\} \qquad s' = s_2 \ominus v}{\{s\}[\![\nu(v, st)]\!]\{s', w\}}\ \text{[NEW-VAR]}$$

$$\frac{\begin{array}{c} f = (p, b, r) \in IntFn \qquad s \in \mathcal{V}(\widetilde{GV} \cup LV) \qquad p = \langle v_1, \ldots, v_k \rangle \\ s_1 = s \oplus (v_1, d_1) \oplus \ldots \oplus (v_k, d_k) \qquad \{s_1\}[\![b]\!]\{s_2, w\} \qquad s' = (s_2 \ominus v_1 \ominus \ldots \ominus v_k) \qquad d = r \triangleright s_2 \end{array}}{\{s\}[\![f(d_1, \ldots, d_k)]\!]\{s', d, w\}}\ \text{[FUNCTION]}$$

Figure 4.4: Execution semantics of statements and functions for node N_i in program instance P_n

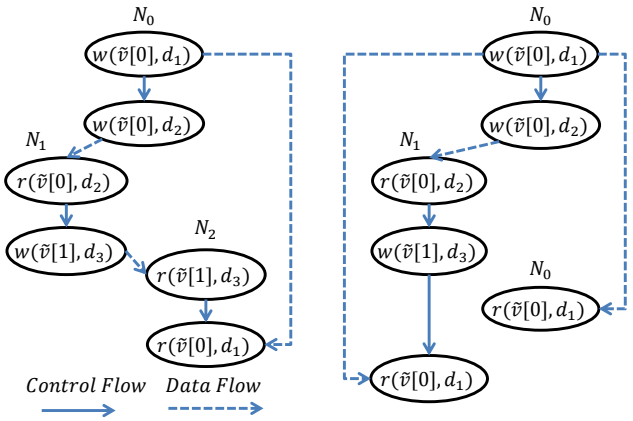

Figure 4.5: Two executions of a program instance P_3. Time flows from top to bottom; $r(\widetilde{v}[i], d)$ = value d was read from $\widetilde{v}[i]$; $w(\widetilde{v}[i], d)$ = value d was written to $\widetilde{v}[i]$. The execution on the left is legal even though N_2 reads a stale value d_1, since it observed no previous writes to v. The execution on the right is illegal for two reasons: (i) once N_1 reads value d_2, it can no longer read the older value d_1; and (ii) N_0 must always read the most recent write to v.

```
%%{
#include <map>
#include "adaptation_manager.h"
double accumulated_risk = 0.0;
map<int,map<int,double> > hazard_map;

void INIT_HAZARDS(int _X,int _Y,
                  int tx,int ty) {...}
double GET_HAZARD(int x,int y) {...}
double update_risk(double risk) {...}
%%}

/*** constants ***/
const X = 20; const Y = 20;
const INITS = 0; const NEXT = 1; ...

/*** external function declarations */
extern void INIT_HAZARDS(...);
extern pure double GET_HAZARD(...);
extern double update_risk(...);
extern bool adaptation_manager(...);
extern int GRID_MOVE(unsigned char _xp,
       unsigned char _yp, double _z);

/* shared variables & initialization */
local unsigned char state = INITS;
//current, next and target coordinates
local unsigned char input x~(0<=x && x<X);
local unsigned char input y~(0<=y && y<Y);
local unsigned char xp = x, yp = y;
local unsigned char xt, yt;
//current formation
global bool input formation;
global bool lock[X][Y] = { lock[x][y] = 1; };

/*** internal functions ***/
//-- compute the next grid on path
bool NEXT_XY () { ... }
//-- stuff once next waypoint reached
void REACHED_NEXT_XY() { ... }

/*** threads ***/
@BarrierSync;
@Period(100000); @Criticality(4);
@WCExecTimeNominal(15000);
@WCExecTimeOverload(30000);
thread COLLISION_AVOIDANCE {
  if(state == INITS) { ... state = NEXT; }
  if(state == NEXT) {
    if(NEXT_XY()) return; state = REQUEST;
  } ...
  else if(state == MOVE) {
    if(GRID_MOVE(xp,yp,0.5)) return;
    REACHED_NEXT_XY(); ... state = NEXT;
  }
}

@Period(100000);
@Criticality(3);
thread WAYPOINT;

/*** roles ***/
role Leader {
  override local unsigned char input xt,
                                input yt;

  override void REACHED_NEXT_XY()
  {
    base.REACHED_NEXT_XY(); ...
  }

  thread COLLISION_AVOIDANCE;

  @WCExecTimeNominal(10000);
  @WCExecTimeOverload(20000);
  thread WAYPOINT { ... }

  @Period(4000000);
  @Criticality(2);
  @WCExecTimeNominal(10000);
  @WCExecTimeOverload(20000);
  thread ADAPTATION_MANAGER
  {
    ...
    formation = adaptation_manager(...);
    ...
  }
}

role Protector {
  thread COLLISION_AVOIDANCE;

  @WCExecTimeNominal(10000);
  @WCExecTimeOverload(20000);
  thread WAYPOINT { ... }
}

/*** properties ***/
pure bool NoCollisions ()
{
  forall_distinct_node_pair (id1,id2) {
    if (x@id1 == x@id2 && y@id1 == y@id2)
      return false;
  }
  return true;
}

require COLLISION_AVOIDANCE => NoCollisions;
```

Figure 4.6: DMPL *program for 5-robot reconnaissance example*

5 Code Generation

In this section, we describe how C++ code can be generated from a DMPL program, given a mapping from node ids to roles. Figure 5.1 shows a fragment of the C++ code generated from the DMPL program shown in Figure 4.6. Notable features of this C++ code are:

- Code within the `%%{...%%}` block is emitted verbatim.
- Constant definitions are converted to `#define` macros.
- External function declarations are eliminated since these functions are assumed to be declared or defined within the code included via the `%%{...%%}` block.
- The number of nodes (available from the mapping from ids to roles) is defined as a macro (line 18).
- A new variable `id` is introduced to store the value of the node's id. The value of this variable is supplied from the command line.
- Shared variables are represented using containers provided by the MADARA middleware [12]. For example a local variable of type `unsigned char` is represented as a MADARA variable of type `madara::UnsignedChar` (line 23). Global variables are expanded to arrays, again using MADARA containers. For example, variables `formation` and `lock` are represented as 1-dimensional (line 25) and 3-dimensional (line 26) vectors, respectively.
- A function (line 29) is generated to initialize input variables and the node's id from command line arguments. Recall that whether a variable is an input or not may depend on the role.
- A function (line 35) is generated to initialize the remaining shared variables.
- Internal functions defined are generated (lines 42-43). The translation from the DMPL function is syntax directed. However, (i) iterators such as `forall_node`, `forall_higher`, `exists_lower` etc. are appropriately based on the value of `id` and `N`; and (ii) access to global variable `v` is replaced by access to `v[id]`. Function names are used to distinguish between those defined at the top level and those defined within roles. A good example is `REACHED_NEXT_XY` and `Leader_REACHED_NEXT_XY`.
- A MADARA context is declared. The context is a collection of variables of which MADARA maintains distributed copies on each node, and provides the consistency required by the semantics of DMPL. In particular, MADARA attaches a Lamport clock with each variable v. The clock is initialized to 0 and incremented with each assignment to v. If v is a global variable, and it is assigned, then the new value of v and its clock are sent to other nodes via network messages. Each receiver node records the Lamport clock of the last version of v received, and discards a received value of v if it comes with an older Lamport clock. Note that this implements the consistency of global variables required by the semantics of DMPL. For local variables, the sequential consistency provided by the hardware directly implements DMPL's semantics.
- Thread functions are generated. For asynchronous threads, the function repeatedly uses the ZSRM scheduler (via `wait_for_next_period`) to suspend till the arrival time of the next job, reads the shared variables atomically (via `read_context`), executes the job, and writes the shared variables back atomically (via `write_context`). Thread functions are named based on whether they are defined globally or within a role. A good example of an asynchronous thread function is `Leader_ADAPTATION_MANAGER`. The functions `read_context` and `write_context` use a special mutex provided by the ZSRM scheduler

```
1  #include <map>
2  #include "adaptation_manager.h"
3  double accumulated_risk = 0.0;
4  map<int,map<int,double> > hazard_map;
5
6  void INIT_HAZARDS(int _X,int _Y,
7                    int tx,int ty) {...}
8  double GET_HAZARD(int x,int y) {...}
9  double update_risk(double risk) {...}
10
11 /*** constants ***/
12 #define X 20
13 #define Y 20
14 #define INITS 0
15 #define NEXT 1
16 ...
17
18 #define N 5 //number of nodes
19 int id; //variable to store node id
20
21 /*** shared variables declaration */
22 #include <madara.h>
23 madara::UnsignedChar state = INITS;
24 madara::UnsignedChar x, y, xp, yp, xt, yt;
25 madara::Vector<Bool> formation(N);
26 madara::Vector3<Bool> lock(X,Y,N);
27
28 /** get input variable values from command line */
29 void initInputs() {
30   /* Initialize id, x, y, formation. Initialize
31      xt, yt if id corresponds to a leader role. */
32 }
33
34 /** shared variable initialization */
35 void initShared()
36 {
37   state = INITS; xp = x; yp = y;
38   lock.set(x,y,id,1);
39 }
40
41 /*** internal functions ***/
42 bool NEXT_XY () { ... }
43 void REACHED_NEXT_XY() { ... }
44 void Leader_REACHED_NEXT_XY() {
45   ... REACHED_NEXT_XY(); ...
46 }
47
48 madara::Context context; //the MADARA context
49
50 /*** threads ***/
51 void COLLISION_AVOIDANCE() {
52   int s = 0;
53   for(;;) {
54     wait_for_next_period();
55     if(s == 0 && barrier()) s = 1;
56     if(s == 1) { read_context(); s = 2; }
57     if(s == 2) {
58       //-- code generated from body of thread
59       //-- from DMPL program goes here
60       if(!job_missed_deadline()) s = 3;
61     }
62     if(s == 3) { write_context(); s = 4; }
63     if(s == 4 && barrier()) s = 0;
64   }
65 }
66
67 void Leader_ADAPTATION_MANAGER()
68 {
69   for(;;) {
70     wait_for_next_period();
71     read_context();
72     ... formation[id] = adaptation_manager(...); ...
73     if(job_missed_deadline()) continue;
74     write_context();
75   }
76 }
77
78 int main()
79 {
80   initInputs(); initShared();
81   registerThreads();
82   startThreads();
83   waitForThreads(); return 0;
84 }
```

Figure 5.1: Generated C++ *code for example* DMPL *program. In practice, local variables (lines 23–26) are duplicated for each thread.*

to protect access to the MADARA context. This mutex is designed to prevent priority-and-criticality inversion (i.e., a higher priority thread being blocked by a lower priority thread for unbounded time on trying to acquire the lock).

- For synchronous thread functions (e.g., `COLLISION_AVOIDANCE`) we develop and use a real-time implementation of the 2BSYNC protocol [4]. The key idea behind this protocol is to use two barriers, at the beginning and end of each job respectively, across all the nodes to enforce the synchronous model of computation. However, in our case, since threads must respect their WCET estimates, a barrier cannot be implemented by an unbounded loop within a job function. Instead, we abort a job whenever the barrier condition is violated (lines 55 and 63) and retry at the next period. Logically this is equivalent to an unbounded loop, and hence the correctness of our implementation follows from that of the original 2BSYNC protocol [4]. Note also that we use the ZSRM scheduler API (line 60) to detect if the current job missed its deadline due to overload, and abort it in this case.
- Finally, the `main` function (lines 80–83) initializes variables from command line arguments, initializes remaining shared variables, registers all threads with the ZSRM scheduler using timing parameters specified in the DMPL file and the Zero-Slack Instants computed via ZSRM schedulability analysis, starts all the threads, and waits for them to complete. In addition to the threads corresponding to the role specified in the DMPL program, function `registerThreads` (line 81) also registers a (high priority) MADARA reader thread that

periodically receives new global variable values and clocks from other nodes and updates the MADARA context appropriately.

6 Verification via Sequentialization

In this section, we present an approach for verifying the reachability properties of synchronous threads. Consider such a property $\varphi = (t, f)$ where t is a synchronous thread and f is a property function. Since t respects the synchronous model of computation, it can be shown [4] that P_n is over-approximated by a sequential program that:

1. Maintains two copies of all global shared variables – $\widetilde{GV}_0$ and $\widetilde{GV}_1$, and one copy of local shared variables per node $LV_0, \ldots, LV_1$.
2. For each DMPL function and thread f, for each node n, produces two versions: $f^n_{\rightarrow}$ and $f^n_{\leftarrow}$. For each n, $f^n_{\rightarrow}$ reads from $\widetilde{GV}_0$ and writes to $\widetilde{GV}_1$, while function $f^n_{\leftarrow}$ reads from $\widetilde{GV}_1$ and writes to $\widetilde{GV}_0$. Both versions read and write the same set of local variables. A given version of a function or thread calls the same version of all sub-functions.
3. Initializes $\widetilde{GV}_0$, $\widetilde{GV}_1$, $LV_0, \ldots, LV_1$ as per constructors.
4. Executes an infinite loop. In each iteration, it: (i) executes the thread $\rightarrow$ version of t once for each node, i.e., n times with the node id set to $0, 1, \ldots, n-1$; (ii) sets the values of variables that can be modified by other threads non-deterministically (to over-approximate interference from them); and (iii) repeats (i), but with the $\leftarrow$ version of all functions.

Informally, $\widetilde{GV}_0$ stores the values of global variables at the start of a round; $\widetilde{GV}_1$ stores their values at the end. Thus, every program instance state $\hat{S}(\pi, t, i)$ is reachable at the start of the i-th iteration of the loop for some execution of this sequential program. To verify whether $\varphi = (t, f)$ holds, we can prove that f invoked at the beginning of the loop always returns TRUE.

6.1 Implementing Sequentialization

Our actual implementation of sequentialization has the following features:

- Local variables are expanded to arrays so we can represent their values for all nodes. This also simplifies generation of the internal and thread functions.
- Input variables are initialized non-deterministically. However, we assume that the property holds at startup. The `assume` command is supported in some form by most state-of-the-art software model checkers. It semantically cuts off all executions that reach the command in a state where the argument to the command evaluates to FALSE.
- This program is written in the C programming language and can therefore be verified using software model checkers that target C programs.

6.2 Bug Finding and Full Verification

For bug finding, we use bounded model checking. To this end, we change the infinite loop (Step 4 above) into one that executes k times for some target bound k. We verify the resulting C program with a software model checker. For complete verification, we prove that the property f (or as is typically the case, a strengthening of it) is inductive. To this end, we create a program that: (i) initializes all variables non-deterministically but assumes that f returns TRUE; (ii) executes the body of the loop (Step 4 above) once; and (iii) asserts that f returns TRUE. We then verify this program.

7 Evaluation

We have implemented a compiler, called DMPLC, that performs both code generation and sequentialization of DMPL programs. The size of the generated code is linear in the size of the input DMPL program. We validated DMPLC on five examples we developed, each consisting of a group of robots coordinating to achieve some goal. We used the V-REP [29] simulator to implement robots and their movement in a physical environment. In particular, we used quadcopter models provided by V-REP to represent each node. DMPLC generates appropriate glue code to use the V-REP API. A copy of DMPL, the examples, and instructions for generating and verifying code are available at https://db.tt/W153VDg6. All experiments were done on a quad-core 2.90GHz laptop with 16 gigabytes of RAM running Kubuntu 14.04 and a publicly-available implementation of the ZSRM schedulability analysis and scheduler [35]. The DMPLC-generated C++ code was compiled with `g++`.

7.1 Reconnaissance example

We implemented the five-node reconnaissance example in about 1300 lines of DMPL code. In addition, we used an external adaptation manager component that was implemented in about 1000 lines of C++ code. The adaptation technique used by this manager is described elsewhere [25] and is orthogonal to this paper. We also analyzed collision avoidance in two ways:

- *Bug Finding:* First, the CBMC [6] tool was used to perform bounded model checking of the sequentialized code. For this, the `assume` command was implemented by the `__CPROVER_assume` function supported by CBMC. This approach is incomplete, but was very effective for finding bugs. We found several bugs due to programming errors (e.g., mistyped variable names, missing checks) in a few seconds with a bound of 5.
- *Full Verification:* Next we proved correctness by showing that the property is inductive over each execution of the `COLLISION_AVOIDANCE` function. Again we used CBMC for model checking. Since the collision avoidance property is not inductive by itself, we manually strengthened it by adding extra invariants, and were able to prove inductiveness. In all, 10 new invariants were added over a 1-person-day effort, and verification by CBMC at the end took a few seconds.

7.2 Other examples

We implemented several other examples, including the following: (i) three robots moving from predefined starting points to ending points on a grid while avoiding collisions; (ii) five robots moving in a tight formation from a starting to an ending point while avoiding collisions; (iii) nine robots moving in a loose formation from a starting to an ending point while avoiding collisions; and (iv) nine robots moving from random initial positions and coming together at the grid center to form a 3×3 square structure. In each example, the generated code demonstrated expected behavior during simulation; collision avoidance was verified. In two cases, the collision avoidance implementation initially had bugs. (We had copied the DMPL code from other examples that we had previously verified, but subtle differences in other threads were unaccounted for). In all cases, our approach found a counter-example that we used to find and fix the errors.

Overall, we believe DMPL provides a good tradeoff between programmability and verifiability of DRTS. Additional examples and videos are available at the DMPL [11] and DART [8] web sites.

8 Future Work and Conclusion

We see several directions for ongoing and future work: (i) developing techniques to construct inductive invariants for synchronous threads; (ii) verification techniques for liveness properties, and properties over asynchronous threads; (iii) implementing an infrastructure to enforce the isolation between code generated by DMPL and external functions; currently since everything executes in the same process and the external functions can execute arbitrary code, we have no way to ensure this; (iv) handling faults and intrusions; this is important especially because one promising application of DRTS is in autonomous systems where human intervention is limited; (v) abstraction and compositional verification techniques.

In summary, DRTS is a promising area for pushing the frontier of computer science and engineering. A programming language that has well-defined semantics and also supports verification is critical to develop high-assurance software for such systems. The language must deal with timing and schedulability as a first class concern, and support tasks with different levels of criticality. Finally, to facilitate adoption, it must play well with existing legacy components since very few DRTS will be built from scratch. While by no means the final answer, we believe that DMPL represents a promising start in this direction.

References/Bibliography

URLs are valid as of the publication date of this document.

[1] Seyed (Reza) Azimi, Gaurav Bhatia, Ragunathan Rajkumar, and Priyantha Mudalige. Reliable intersection protocols using vehicular networks. In Chenyang Lu, P. R. Kumar, and Radu Stoleru, editors, *Proceedings of the 4th International Conference on Cyber-Physical Systems (ICCPS '13)*, pages 1–10, Philadelphia, PA, USA, April 2013. Association for Computing Machinery.

[2] Sanjoy K. Baruah, Alan Burns, and Robert I. Davis. Response-Time Analysis for Mixed Criticality Systems. In *Proceedings of the 32nd Real-Time Systems Symposium (RTSS '11)*, pages 34–43, Vienna, Austria, November–December 2011. IEEE Computer Society.

[3] Alan Burns and Robert I. Davis. Mixed Criticality Systems - A Review, 2015. http://www-users.cs.york.ac.uk/burns/review.pdf.

[4] Sagar Chaki and James Edmondson. Model-Driven Verifying Compilation of Synchronous Distributed Applications. In Juergen Dingel and Wolfram Schulte, editors, *Proceedings of the 17th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '14)*, Lecture Notes in Computer Science, Valencia, Spain, September 28 - October 3, 2014. New York, NY, September–October 2014. Springer-Verlag.

[5] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Time-Bounded Analysis of Real-Time Systems. In *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD '11)*, pages 72–80, Austin, TX, October–November 2011. IEEE Computer Society.

[6] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, Barcelona, Spain, March 29–April 2, 2004. New York, NY, March–April 2004. Springer-Verlag.

[7] Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 331–340, Waikiki, Honolulu , HI, USA, May 21-28, 2011. New York, NY, May 2011. Association for Computing Machinery.

[8] DART website. http://cps-sei.github.io/dart/videos.html.

[9] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proceedings of the 30th Real-Time Systems Symposium (RTSS '09)*, pages 291–300, Washington, DC, USA, December 2009. IEEE Computer Society.

[10] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In Hans-Juergen Boehm and Cormac Flanagan, editors, *Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation (PLDI '13)*, pages 321–332, Seattle, WA, June 2013. Association for Computing Machinery.

[11] DMPL website. https://github.com/cps-sei/dmplc.

[12] James R. Edmondson and Aniruddha S. Gokhale. Design of a Scalable Reasoning Engine for Distributed, Real-Time and Embedded Systems. In Hui Xiong and W. B. Lee, editors, *Proceedings of the 5th International Conference on Knowledge Science, Engineering and*

Management (KSEM '11), volume 7091 of *Lecture Notes in Computer Science*, pages 221–232, Irvine, CA, USA, December 12-14, 2011, December 2011. Springer-Verlag.

[13] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '14)*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, Grenoble, France, March 2014. Springer-Verlag.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, 1985.

[15] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4), 2009.

[16] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 201–209, Portland, OR, October 2013. IEEE Computer Society.

[17] David Kyle, Jeffery Hansen, and Sagar Chaki. Statistical Model Checking of Distributed Adaptive Real-Time Software. In *Proceedings of the 15th International Conference on Runtime Verification (RV '15)*, Vienna, Austria, September 2015.

[18] Akash Lal and Thomas W. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In Aarti Gupta and Sharad Malik, editors, *Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 37–51, Princeton, NJ, USA, July 7-14, 2008. New York, NY, July 2008. Springer-Verlag.

[19] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, January 1973.

[20] Damian M. Lyons, Ronald C. Arkin, Shu Jiang, Dagan Harrington, and T. Liu. Verifying and validating multirobot missions. In *Proceedings of 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1495–1502, Chicago, IL, September 2014. IEEE Computer Society.

[21] MADARA website. http://sourceforge.net/projects/madara.

[22] Steven P. Miller, Darren D. Cofer, Lui Sha, Jose Meseguer, and Abdullah Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Proceedings of the 28th Digital Avionics Systems Conference (DASC '09)*, pages 1.A.3–1–1.A.3–12, Orlando, FL, USA, October 2009. IEEE Computer Society.

[23] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.

[24] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[25] Gabriel Moreno, Javier Camara, David Garlan, and Bradley Schmerl. Proactive Self-Adaptation under Uncertainty: a Probabilistic Model Checking Approach. In *Proceedings of the 23rd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '15)*. Association for Computing Machinery, September 2015.

[26] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers (TC)*, 39(9):1175–1185, September 1990.

[27] SystemJ website. http://systemjtechnology.com.

[28] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 477–492, Grenoble, France, June 26 - July 2, 2009. New York, NY, July 2009. Springer-Verlag.

[29] V-REP website. http://www.coppeliarobotics.com.

[30] Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the 28th Real-Time Systems Symposium (RTSS '07)*, pages 239–243, Tucson, Arizona, USA, December 2007. IEEE Computer Society.

[31] Peter H. Welch, Jan Bækgaard Pedersen, Fred R. M. Barnes, Carl G. Ritson, and Neil C. C. Brown. Adding Formal Verification to occam-π. In *Proceedings of the 33th Communicating Process Architectures Conference (CPA'11)*, Concurrent Systems Engineering Series, page 379, Limerick, Ireland, June 2011. IOS Press.

[32] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

[33] Po-Liang Wu, Dhashrath Raguraman, Lui Sha, Richard B. Berlin Jr., and Julian M. Goldman. A Treatment Validation Protocol for Cyber-Physical-Human Medical Systems. In *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'14)*, pages 183–190, Verona, Italy, August 2014. IEEE Computer Society.

[34] Håkan L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2005. Technical report no. CMU-CS-05-105.

[35] ZSRM website. https://github.com/cps-sei/mzsrm.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	May 2016	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
DMPL: Programming and Verifying Distributed Mixed-Synchrony and Mixed-Critical Software	FA8721-05-C-0003

6. AUTHOR(S)

Sagar Chaki, David Kyle

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2016-TR-005

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

A language called DMPL for programming distributed real-time, mixed-criticality software is presented. DMPL supports a distributed system in which each node executes a set of periodic real-time threads that are scheduled on the basis of their priority and criticality. Both synchronous and asynchronous threads are allowed. The syntax and semantics of DMPL are formally described. A compiler that generates C++ code from a DMPL program is presented. Two methods of verification of properties of synchronous threads via sequentialization are proposed: fully-automated bounded model checking, and deductive verification with manually-supplied invariants. DMPL programming and verification are validated on several examples of collision avoidance in multi-robot systems.

14. SUBJECT TERMS	15. NUMBER OF PAGES
DMPL, multi-thread, synchronous, asynchronous, real-time, distributed systems, C++, multi-agent systems, mixed-critical software	39

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102