Model-Driven Verifying Compilation of Synchronous Distributed Applications*

Sagar Chaki James Edmondson

Carnegie Mellon University, Pittsburgh, USA
{chaki,jredmondson}@sei.cmu.edu

Abstract. We present an approach, based on model-driven verifying compilation, to construct distributed applications that satisfy user-specified safety specifications, assuming a ”synchronous network” model of computation. Given a distributed application P_d and a safety specification φ in a domain specific language DASL (that we have developed), we first use a combination of sequentialization and software model checking to verify that P_d satisfies φ. If verification succeeds, we generate an implementation of P_d that uses a novel barrier-based synchronizer protocol (that we have also developed) to implement the synchronous network semantics. We present the syntax and semantics of DASL. We also present, and prove correctness of, two sequentialization algorithms, and the synchronizer protocol. Finally, we evaluate the two sequentializations on a collection of distributed applications with safety-critical requirements.

1 Introduction

Distributed applications (i.e., software implementing distributed algorithms) play a critical, often silent, role in our day-to-day lives. Increasingly, they are being used in safety-critical domains. For example, Cyber-Physical intersection protocols [4] have been developed for ground-based vehicles that rely on vehicle-to-vehicle (V2V) communication. Safety-critical distributed applications must be subjected to rigorous verification & validation (V&V) before deployment. Indeed, incorrect operation of such applications can lead to damage or destruction of property, personal injury, and even loss of life.

The state-of-the-art in V&V of distributed applications relies heavily on testing. This has two problems. *First*, testing has poor coverage. This is particularly severe for distributed applications, since concurrency enables a large number of possible executions . *Second*, safety-critical applications are often produced via model-driven development (MDD), e.g., using Simulink in the automotive domain. While some form of testing is applied at each level of MDD, the assurance obtained at one level is not transferred to the next. In this paper, we

* 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 material has been approved for public release and unlimited distribution. DM-0001118

present and empirically evaluate an approach, called DIVER, for producing verified distributed applications, that addresses both these challenges. Specifically, DIVER uses software model checking, an exhaustive and automated technique, for verification. It also uses a single “model” of the application to perform both verification and code generation, thus transferring the results of one to the other.

DIVER targets the *synchronous network* model of computation [16], or SNMOC, where each node executes in rounds. Nodes communicate via single-writer-multiple-reader shared variables[1]. The final value of a variable at its writer node in any round (i) becomes visible to its reader nodes in the next round ($i+1$). SNMOC makes both programming and verification simpler, and is used in safety-critical domains, e.g., it reduced [17] verification time of an active-standby protocol (used in avionics systems) from 35 hours to 30 seconds.

DIVER is a *verifying compiler* [11]. The input to DIVER is a program P_d written in a domain specific language we have developed called Distributed Application Specification Language (DASL). P_d describes both a distributed application *App* and its correctness specification φ. DIVER outputs an executable for each node of *App* but only if it satisfies φ. It works in two steps:

1. *Verification*: Verify whether *App* satisfies φ. The verification is automated and exhaustive, and consists of two sub-steps:
   (a) *Sequentialization*: Construct a sequential (i.e., single threaded) program P_s that is semantically equivalent to *App* w.r.t. φ. Specifically, P_s is a C program containing an assertion α such that $P_s \models \alpha \iff P_d \models \varphi$, i.e., all legal executions of P_s satisfy α iff P_d satisfies φ.
   (b) *Model Checking*: Verify whether $P_s \models \alpha$ using software model checking [13] (SMC). We chose C and assertions for expressing P_s and α since these are the de-facto standards for describing SMC problems, and supported by state-of-the-art SMC engines. If SMC successfully verifies that $P_s \models \alpha$ then proceed to Step 2, otherwise declare $App \not\models \varphi$ and abort.
2. *Code Generation*: Generate C++ code for each node of *App* that relies on the MADARA [8] middleware for communication. We choose MADARA due to prior expertise, and our ability to implement SNMOC on top of its primitives. However, DIVER is compatible with other middleware that either support SNMOC natively, or provide an API on top of which SNMOC is implementable.

Our ultimate goal is to verify distributed applications running on mobile robots communicating over wireless networks. Such networks are not only asynchronous but have unbounded message delay. Therefore, we have also developed a protocol, called 2BSYNC, that implements SNMOC over asynchronous networks without relying on clock synchronization. To our knowledge, it is a new synchronizer protocol for wirelessly connected systems, and of independent interest.

The rest of this paper is organized as follows. After surveying related work (Sec. 2), we focus on our specific contributions. In Sec. 3, we present the syntax and semantics of DASL. The semantics leads immediately to a sequentialization we call SEQSEM. However SEQSEM produces a program with $\mathcal{O}(n^2)$ variables

[1] A version of SNMOC based on message-passing also appears in the literature.

(where n = number of nodes). This is undesirable from a verification perspective since the statespace of a program grows exponentially with the number of variables. Therefore, in Sec. 4 we develop, and prove correctness of, a more sophisticated sequentialization, called SEQDBL, that only requires $\mathcal{O}(n)$ variables. In Sec. 5, we present and prove correctness of our synchronizer protocol 2BSYNC. In Sec. 6, we present code generation from DASL to MADARA/C++. In Sec. 7, we compare SEQSEM and SEQDBL on a collection of distributed applications. Our results indicate that while SEQDBL is clearly better overall, for some applications, SEQSEM produces programs that are verified more quickly despite having many more variables. Finally, Sec. 8 concludes the paper.

2 Related Work

This work spans multiple disciplines – verification, distributed systems, middleware technology and code generation – which we briefly survey.

Verification. Most work in model checking concurrent software [2] use an asynchronous model of computation, based on either shared memory [1] or message-passing [7]. Some of these projects are also based on sequentialization [14, 24]. Synchronous programming languages, such as Lustre [5], are not suitable for distributed applications, since they can only describe systems with a fixed number of nodes. DIVER is a verifying compiler for synchronous distributed applications that does both model-driven verification and code generation from a single DASL program. Humphrey et al. [12] use LTL to specify and synthesize correct multi-UAV missions. In contrast, our approach is based on verification. Process calculi, such as CCS [18] and CSP [10], use asynchronous message-passing communication and are verified via refinement checking. DASL uses synchronous shared-variable based communication, and its verification is based on model checking user-specified assertions. The synchronous programming language Lustre [5] differs from DASL in that there can be no cyclic-dependency (i.e., causality loops) between nodes, and each Lustre program has a fixed number of nodes. Note, however, that every "instance" of a DASL program can be represented in Lustre using unit-delay nodes to break causality.

Distributed Systems. Distributed algorithms are typically verified at the pseudo-code level manually using invariants and simulation relations [16]. Distributed systems are also heavily simulated [23] and tested, which are incomplete. DIVER is based on model checking, which is automated and exhaustive. Synchronizer protocols [3] have also been widely studied. Many rely on clock synchronization [17] – which is inappropriate for wireless communication – or direct message passing. 2BSYNC uses barriers, which is more appropriate for middleware, like MADARA [8], that provide a shared memory abstraction.

Middleware and Code Generation. For our code generation target, we chose MADARA [8]. There are multiple middleware solutions that provide infrastructure support for control and communication between distributed applications. CORBA [22] is an OMG standard for component-based distributed application development, but requires definition of component interactions and precise management of transportation options. OMG has another standard called the Data Distribution Service [19] which facilitates quality-of-service contracts between publishers and subscribers and a complex but robust networking feature set. Tripakis et al. [25] have also explored implementing synchronous models via reduction to Kahn Process networks.

Several toolkits – e.g., COSMIC [20], AUTOSAR [9], and OCARINA [15] – provide verification and code generation for distributed applications, often with requirements of real-time support from underlying hardware, network connections, and operating systems. They force component paradigms or complex deployment configurations and metadata that is unnecessary for synchronous application specification and hinders verification. MADARA provides a more direct mapping for distributed algorithm logic, specializes in wireless communication – which is more appropriate for our target domain – and enforces Lamport clock-based consistency which provides a clean semantics and supports verification.

3 The DASL Language

A DASL program P_d describes a distributed application *App*, as well as its specification. The application consists of a number of nodes communicating via global variables over a synchronous network. Recall that each node executes in rounds. Formally, P_d is a 5-tuple $(GV, LV, \rho, n, \varphi)$ where: (i) GV is the set of global variables; (ii) LV is the set of node-local variables whose values persist across rounds; (iii) ρ is a function executed by each node in every round; (iv) n is the number of nodes; and (v) φ is the specification defined by a pair of functions *Init* and *Safety* that, respectively, establish a valid initial state, and check for violations of the desired safety property. The specification φ is used for verification only. The rest of P_d is used both for verification and code generation.

Syntax of DASL. 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 ρ is a statement. The "abstract" syntax of statements, lvalues and expressions is given by the following BNF grammar:

$$\textbf{(Statements)}\ stmt := skip \mid lval = exp \mid \textsc{ite}(exp, stmt, stmt) \mid \textsc{while}(exp, stmt) \mid \textsc{all}(IV, stmt) \mid \langle stmt\ ; \ldots ;\ stmt \rangle \mid \nu(TV, stmt)$$

$$\textbf{(LValues)}\ lval := GV[exp] \mid LV \mid TV$$

$$\textbf{(Expressions)}\ exp := \mathbb{Z} \mid lval \mid id \mid IV \mid \sim exp \mid exp \diamond exp$$

Intuitively, *skip* is a nop, $l = e$ is an assignment, ITE is an "if-then-else", WHILE is a while loop, $\textsc{all}(v, st)$ executes st iteratively by substituting v with 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, $\sim \in \{-, \neg\}$ is an unary operator, and $\diamond \in \{+, -, *, /, \wedge, \vee\}$ is a binary operator. ALL enables iteration over all nodes of *App* without knowing the exact number of such nodes a-priori.

Scoping and Assumptions. All global variables are arrays. We assume that: (i) each element of a global array has a single writer node; the mechanisms to

```
CONST OUTSIDE = 0;
CONST TRYING = 1;
CONST INSIDE = 2;

NODE node(id) {
  GLOBAL _Bool lock[#N];
  LOCAL unsigned char state;

  void ROUND() {
    _BOOL c;
    if(state == OUTSIDE) {
      c = should_enter();
      if(c) {
        if(EXISTS_LOWER(idp,lock[idp]))
          return;
        lock[id] = 1; state = TRYING;
      }
    } else if(state == TRYING) {
        if(EXISTS_HIGHER(idp,lock[idp]))
          return;
        state = INSIDE;
    } else if(state == INSIDE) {
      if(in_cs()) return;
      lock[id] = 0; state = OUTSIDE;
    }
  }
}
PROGRAM = node(0) || node(1);


void INIT()
{
  FORALL_NODE(id) {
    ND(state.id); ND(lock[id]);
    ASSUME(state.id == OUTSIDE &&
      lock[id] == 0 ||
      state.id == INSIDE &&
      lock[id] == 1);
  }
  FORALL_DISTINCT_NODE_PAIR
    (id1,id2) {
    ASSUME(state.id1 != INSIDE ||
           state.id2 != INSIDE);
  }
}

void SAFETY()
{
  FORALL_DISTINCT_NODE_PAIR
    (id1,id2) {
    ASSERT(state.id1 != INSIDE ||
           state.id2 != INSIDE);
  }
}
```

Fig. 1. Example DASL program with 2 nodes using an id-based mutex protocol.

enforce this are discussed later; (ii) variables in $GV \cup LV \cup \{id\}$ are always in scope; (iii) for each statement $\textsc{all}(v, st)$ and $\nu(v, st)$, variable v is in scope of st; (iv) scoping is unambiguous, and only variables in scope are used in expressions; (v) id and id variables do not appear on the LHS of assignments, i.e., they are read-only; (vi) in any execution of ρ, a global array element is written atmost once. Note that these assumptions do not limit expressivity.

Init and Safety. The body of *Init* is a statement whose syntax is the same as *stmt* except that *lval* and *exp* are defined as:

$$\textbf{(LValues)}\ lval := GV[exp] \mid LV.IV \mid TV$$
$$\textbf{(Expressions)}\ exp := \mathbb{Z} \mid lval \mid IV \mid \sim exp \mid exp \diamond exp$$

Thus the key differences of *Init* with ρ are: (i) variable id is no longer in scope; and (ii) it is able to refer to local variables of nodes – specifically, the lvalue $v.i$ refers to local variable v of node with id i. Function *Safety* is the same as *Init* except: (i) it cannot access global variables; and (ii) it cannot modify local variables. Formally, the body of *Safety* is a statement whose syntax is the same as *stmt* except that *lval* and *exp* are defined as:

$$\textbf{(LValues)}\ lval := TV$$
$$\textbf{(Expressions)}\ exp := \mathbb{Z} \mid lval \mid LV.IV \mid IV \mid \sim exp \mid exp \diamond exp$$

Concrete Syntax. The "concrete" syntax of P_d consists of declarations for GV and LV, definitions of ρ, *Init*, and *Safety*, and the value of n. For example,

Figure 1 shows a DASL program with 2 nodes that use a protocol based on their ids to ensure mutual exclusion. The program consists of constant definitions (lines 1–3), the nodes and their ids (line 28), definition of function *Init* (lines 31–45), function *Safety* (lines 47–54), declarations of GV (line 6), LV (line 7), and the definition of function ρ (lines 9–26). Note that:

1. The concrete syntax is similar to C. This provides familiarity to practitioners, and simplifies sequentialization and code generation.
2. Constant definitions (lines 1–3) are allowed for readability.
3. Multi-dimensional global arrays are supported. Dimension `#N` denotes the number of nodes. Thus, there is one element of `lock` for each node. This supports a programming pattern where a node always writes to a global array element whose index equals its id (lines 16 and 24), ensuring that every global array element has one writer node.
4. Function ρ is called `ROUND`, and variable *id* is called `id`.
5. A node can invoke external functions (e.g., `should_enter` on line 12 and `in_cs` on line 23) as needed. External functions are assumed to be "pure" (i.e., they do not modify global, local, or temporary variables) and to return integer values non-deterministically.
6. There are three built-in functions to aid specification: (i) `ND(v)` sets variable `v` to a value non-deterministically; (ii) `ASSUME(e)` blocks all executions where `e` is FALSE; and (iii) `assert(e)` aborts all executions where `e` is FALSE. `ASSUME` and `ND` help specify legal initial states (lines 34, 35 and 42). `ASSERT` helps (line 51) to check for a violation of the safety property.
7. Iterators are available to: (i) execute a statement over all nodes (`FORALL_NODE` at line 33), all pairs of distinct nodes (`FORALL_DISTINCT_NODE_PAIR` at line 40 and 49), etc.; and (ii) evaluate an expression disjunctively over nodes that have a lower id (`EXISTS_LOWER` at line 14), a higher id (`EXISTS_HIGHER` at line 19), etc. They are all "syntactic sugar" defined formally using ALL in a natural manner.

Example 1. The DASL program in Figure 1 uses global variable `lock` to ensure mutual exclusion. Specifically, the node with id `id` enters the critical section (CS) if `id` is the largest index for which `lock[id]` is TRUE. To enter the CS, a node first checks (line 14) if the CS is available (i.e., not occupied by another node with smaller id). If this is not the case, it retries in the next round (line 15). Otherwise, it requests the CS (line 16). In the next round, the node checks (line 19) if it can enter the CS. If not, it retries (line 20) in the next round. Otherwise, it enters the CS (line 21). Once in the CS, the node performs arbitrary computation (line 23), releases the lock and exits (line 24). Note that since `in_cs` (line 23) returns a non-deterministic value, the node remains in the CS for arbitrary many rounds. *Init* ensures that initially each node is either inside or outside the CS (lines 33–39) with atmost one node being inside (lines 40–44). Function *Safety* aborts (lines 49–53) if multiple nodes are in the CS simultaneously.

Semantics of DASL. Consider a DASL program $P_d = (GV, LV, \rho, n, \varphi)$. We define the semantics of P_d in terms of a "sequential" (i.e., single-threaded) pro-

$$\Delta(\epsilon_1, \epsilon_2, skip) \equiv skip \qquad \Delta(\epsilon_1, \epsilon_2, l = e) \equiv \epsilon_1(l) = \epsilon_2(e)$$
$$\Delta(\epsilon_1, \epsilon_2, \text{ITE}(e, s, s')) \equiv \text{ITE}(\epsilon_2(e), \Delta(\epsilon_1, \epsilon_2, s), \Delta(\epsilon_1, \epsilon_2, s'))$$
$$\Delta(\epsilon_1, \epsilon_2, \text{WHILE}(e, s)) \equiv \text{WHILE}(\epsilon_2(e), \Delta(\epsilon_1, \epsilon_2, s))$$
$$\Delta(\epsilon_1, \epsilon_2, \text{ALL}(v, s)) \equiv \langle \Delta(\epsilon_1 \oplus (v, 0), \epsilon_2 \oplus (v, 0), s); \ldots; \Delta(\epsilon_1 \oplus (v, n-1), \epsilon_2 \oplus (v, n-1), s)\rangle$$
$$\Delta(\epsilon_1, \epsilon_2, \langle s; s'\rangle) \equiv \langle \Delta(\epsilon_1, \epsilon_2, s); \Delta(\epsilon_1, \epsilon_2, s')\rangle \qquad \Delta(\epsilon_1, \epsilon_2, \nu(v, s)) \equiv \nu(v, \Delta(\epsilon_1, \epsilon_2, s))$$

Fig. 2. The statement transformer mapping Δ.

gram. Recall that P_d consists of n nodes executing concurrently and communicating via the shared variables GV. Each node is assigned a unique id between 0 and $n-1$, with N_i denoting the node with id i. We first create n copies of GV and LV, one for each node. For any $v \in GV \cup LV$, let v_i denote its copy made for N_i. Next, for each node N_i we create a copy of ρ, denoted ρ_i, by: (i) replacing each $v \in GV \cup LV$ with v_i; and (ii) expanding out each statement of the form $\text{ALL}(v, st)$ appropriately. We now define this formally.

ID Instantiation. An id instantiation is a partial mapping from $id \cup IV$ to $\mathbb{Z}$. Let $IdInst$ be the set of id instantiations. Let $\mu_\perp$ denote the empty id instantiation, i.e., $Domain(\mu_\perp) = \emptyset$. Given an id instantiation μ, a variable $v \notin Dom(\mu)$ and an integer z, $\mu \oplus (v, z)$ is the id instantiation that extends μ by mapping v to z.

Expression Transformer. An expression transformer is a mapping from expressions to expressions. Let $ExpTrans$ be the set of all expression transformers. Every id instantiation induces an expression transformer as follows.

Definition 1. *Define a mapping $\epsilon : IdInst \mapsto ExpTrans$ such that for any $\mu \in IdInst$ and $e \in exp$, $\epsilon(\mu, e)$ is obtained from e by replacing: (i) each $v \in GV \cup LV$ with $v_{\mu(id)}$; and (ii) each $v.i \in LV.IV$ with $v_{\mu(i)}$.*

A pair of expression transformers (ϵ_1, ϵ_2) induces a statement transformer that uses ϵ_1 to transform lvalues, ϵ_2 to transform expressions, and expands ALL statements. Formally, this defined as follows.

Definition 2 (Statement Transformer). *Define a mapping $\Delta : ExpTrans \mapsto ExpTrans \mapsto stmt \mapsto stmt$ as shown in Figure 2.*

Often, the two expression transformer arguments of Δ are equal. Therefore, for simplicity we write $\Delta(\epsilon, s)$ to mean $\Delta(\epsilon, \epsilon, s)$. Let the body of any function f be denoted by the statement $f()$. Then the semantics of node N_i in each round is given by the function ρ_i such that:

$$\rho_i() = \Delta(\epsilon(\mu_\perp \oplus (id, i)), \rho())$$

Thus, the body of ρ_i is obtained by transforming the body of ρ, starting with an id instantiation that maps id to i. Also, define functions $\tilde{Init}$ and $\tilde{Safety}$ as:

$$\tilde{Init}() = \Delta(\epsilon(\mu_\perp), Init()) \qquad \tilde{Safety}() = \Delta(\epsilon(\mu_\perp), Safety()) \tag{1}$$

Thus, when transforming *Init* and *Safety*, variable *id* is not in scope. Also, every lvalue $v.i$ is transformed to $v_{\mu(i)}$ since it refers to the local variable v of node N_i.

Semantics of P_d. The semantics of P_d is the sequential program that: (i) initializes variables by executing $\tilde{Init}()$; and then (ii) executes rounds. Each round consists of the following steps: (a) for every global array element $v[j]$, copy its value at its writer node to all its reader nodes; (b) check the property by executing $\tilde{Safety}()$; and (c) execute the sequence of statements $\langle \rho_0(); \ldots ; \rho_{n-1}() \rangle$.

Recall that every global variable is an array. For a global variable $v \in GV$, let $Dim(v)$ denote its size. For each $j \in [1, Dim(v)]$, let $\mathcal{W}(v, j)$ denote the index of the node that writes to the element $v[j]$. Note that $\mathcal{W}(v, j)$ is well-defined due to our assumption that all global variables have a single writer node.

Definition 3 (Semantics). *The semantics of a* DASL *program* $P_d = (GV, LV, \rho, n, \varphi)$*, denoted* $[\![P_d]\!]$*, is the sequential program:*

$$[\![P_d]\!] = \langle \tilde{Init}(); \text{WHILE}(\text{TRUE}, Round)\rangle, \text{ where}$$
$$Round = \langle CopyGlobals; \tilde{Safety}(); \rho_0(); \ldots ; \rho_{n-1}()\rangle, \text{ where}$$
$$CopyGlobals = \forall v \in GV \centerdot \forall j \in [1, Dim(v)] \centerdot \forall i \in [0, n) \centerdot v_i[j] = v_{\mathcal{W}(v,j)}[j]$$

Note that the quantifiers in the definition of CopyGlobals are finitely instantiable. Hence, CopyGlobals expands to a finite sequence of assignments.

The semantics of P_d (Definition 3) is a sequential program. Thus, the procedure to construct $[\![P_d]\!]$, denoted SEQSEM, is a valid sequentialization for DASL. Note that $[\![P_d]\!]$ has $\mathcal{O}(n^2)$ global variables since there are $\mathcal{O}(n)$ global arrays, and each global array has $\mathcal{O}(n)$ elements. In Sec. 4 we present a more advanced sequentialization, SEQDBL, that produces programs with $\mathcal{O}(n)$ global variables.

4 Sequentializing DASL Programs

SEQDBL uses only two copies of GV, $GV^{\mathbf{1}}$ and $GV^{\mathbf{0}}$, where: (i) $GV^{\mathbf{1}}$ is used as input in odd rounds and output in even rounds, while (ii) $GV^{\mathbf{0}}$ is used as input in even rounds and output in odd rounds. More specifically, SEQDBL constructs the program P_s that: (i) initializes $GV^{\mathbf{1}}$ and LV by executing *Init*(); and (ii) executes rounds. An odd round consists of the following steps: (a) check the property by executing *Safety*(); (b) copy $GV^{\mathbf{1}}$ to $GV^{\mathbf{0}}$; (b) execute the sequence of statements $\langle \rho_0(); \ldots ; \rho_{n-1}() \rangle$, reading from $GV^{\mathbf{1}}$ and writing to $GV^{\mathbf{0}}$. An even round is the same as an odd round except that the roles of $GV^{\mathbf{1}}$ and $GV^{\mathbf{0}}$ are reversed. We now define P_s formally. For a global variable $v \in GV$, let $v^{\mathbf{1}}$ and $v^{\mathbf{0}}$ be its copy in $GV^{\mathbf{1}}$ and $GV^{\mathbf{0}}$, respectively. We begin with two expression transformers, $\epsilon^{\mathbf{1}}$ and $\epsilon^{\mathbf{0}}$. Then, we use them to transform functions *Init*, *Safety*, and $\rho_0, \ldots, \rho_{n-1}$. Finally, we define P_s in terms of these transformed functions.

Definition 4. *Define a mapping* $\epsilon^{\mathbf{1}} : IdInst \mapsto ExpTrans$ *such that for any* $\mu \in IdInst$ *and* $e \in exp$*,* $\epsilon^{\mathbf{1}}(\mu, e)$ *is obtained from* e *by replacing: (i) each* $v \in GV$

with $v^{\mathbf{1}}$; (ii) each $v \in LV$ with $v_{\mu(id)}$; and (iii) each $v.i \in LV.IV$ with $v_{\mu(i)}$. Define mapping $\epsilon^{\mathbf{0}} : IdInst \mapsto ExpTrans$ to be the same as $\epsilon^{\mathbf{1}}$, except that every $v \in GV$ is replaced by $v^{\mathbf{0}}$.

Note that the only difference between $\epsilon^{\mathbf{1}}$ and $\epsilon^{\mathbf{0}}$ is in the treatment of global variables. For $i \in [0, n)$ define functions $\rho_i^{\mathbf{1}}$ and $\rho_i^{\mathbf{0}}$ such as:

$$\begin{aligned} \rho_i^{\mathbf{1}}() &= \Delta(\epsilon^{\mathbf{0}}(\mu_\perp \oplus (id, i)), \epsilon^{\mathbf{1}}(\mu_\perp \oplus (id, i)), \rho()) \\ \rho_i^{\mathbf{0}}() &= \Delta(\epsilon^{\mathbf{1}}(\mu_\perp \oplus (id, i)), \epsilon^{\mathbf{0}}(\mu_\perp \oplus (id, i)), \rho()) \end{aligned} \tag{2}$$

Note that $\rho_i^{\mathbf{1}}$ uses $GV^{\mathbf{0}}$ for LHS of assignments, and $GV^{\mathbf{1}}$ for other expressions. Thus, $\rho_i^{\mathbf{1}}$ reads $GV^{\mathbf{1}}$ and modifies $GV^{\mathbf{0}}$. Similarly, $\rho_i^{\mathbf{0}}$ reads $GV^{\mathbf{0}}$ and modifies $GV^{\mathbf{1}}$. Also, define functions $\ddot{Init}$, $Safety^{\mathbf{1}}$ and $Safety^{\mathbf{0}}$ as:

$$\begin{gathered} \ddot{Init}() = \Delta(\epsilon^{\mathbf{1}}(\mu_\perp), Init()) \qquad Safety^{\mathbf{1}}() = \Delta(\epsilon^{\mathbf{0}}(\mu_\perp), \epsilon^{\mathbf{1}}(\mu_\perp), Safety()) \\ Safety^{\mathbf{0}}() = \Delta(\epsilon^{\mathbf{1}}(\mu_\perp), \epsilon^{\mathbf{0}}(\mu_\perp), Safety()) \end{gathered} \tag{3}$$

Note that, $\ddot{Init}$ reads and modifies $GV^{\mathbf{1}}$, $Safety^{\mathbf{1}}$ reads $GV^{\mathbf{1}}$ and modifies $GV^{\mathbf{0}}$, while $Safety^{\mathbf{0}}$ reads $GV^{\mathbf{0}}$ and modifies $GV^{\mathbf{1}}$. We now define P_s formally.

Definition 5 (Sequentialization). *The sequentialization of a* DASL *program $P_d = (GV, LV, \rho, n, \varphi)$, denoted P_s, is the sequential program:*

$$\begin{aligned} P_s &= \langle \ddot{Init}(); \text{WHILE}(\text{TRUE}, \langle Round^{\mathbf{1}}; Round^{\mathbf{0}} \rangle) \rangle, \text{ where} \\ Round^{\mathbf{1}} &= \langle Safety^{\mathbf{1}}(); CopyFwd; \rho_0^{\mathbf{1}}(); \dots; \rho_{n-1}^{\mathbf{1}}() \rangle, \text{ where} \\ CopyFwd &= \forall v \in GV \centerdot \forall j \in [1, Dim(v)] \centerdot v^{\mathbf{0}}[j] = v^{\mathbf{1}}[j], \text{ and} \\ Round^{\mathbf{0}} &= \langle Safety^{\mathbf{0}}(); CopyBwd; \rho_0^{\mathbf{0}}(); \dots; \rho_{n-1}^{\mathbf{0}}() \rangle, \text{ where} \\ CopyBwd &= \forall v \in GV \centerdot \forall j \in [1, Dim(v)] \centerdot v^{\mathbf{1}}[j] = v^{\mathbf{0}}[j] \end{aligned}$$

Note that CopyFwd and CopyBwd expand to a finite sequence of assignments.

Correctness of SEQDBL. We now show that $[\![P_d]\!]$ and P_s are semantically equivalent, i.e., there is an execution of $[\![P_d]\!]$ that aborts iff there is an execution of P_s that aborts. For brevity, we only give a proof sketch. First, recall that $[\![P_d]\!]$ has n copies of GV, while P_s has just two. For simplicity, let $\mathbb{D}$ be the domain of values of all variables. Given a set of variables X, let $\mathcal{V}(X)$ be the set of mapping from X to $\mathbb{D}$. We write $\mathcal{V}_d$ to mean $\mathcal{V}(GV_1 \cup \dots \cup GV_n)$, $\mathcal{V}^{\mathbf{1}}$ to mean $\mathcal{V}(GV^{\mathbf{1}})$, $\mathcal{V}^{\mathbf{0}}$ to mean $\mathcal{V}(GV^{\mathbf{0}})$, and $\mathcal{V}_l$ to mean $\mathcal{V}(LV)$. Thus, for example, an element of $\mathcal{V}_l$ maps local variables to values.

To relate $[\![P_d]\!]$ and P_s, we relate valuations of global variables of one to global variables of the other. Formally, we define a relation $\approx \subseteq \mathcal{V}_d \times (\mathcal{V}^{\mathbf{1}} \cup \mathcal{V}^{\mathbf{0}})$ as follows:

$$V \approx V' \iff \forall v \in GV \centerdot \forall j \in [1, Dim(v)] \centerdot V(v_{\mathcal{W}(v,j)}[j]) = V'(v[j])$$

In other words, V and V' are related iff for every global array element $v[j]$, the value of $v[j]$ at its writer node $\mathcal{W}(v, j)$ according to V is the same as the value of $v[j]$ according to V'. A state of $[\![P_d]\!]$ is a pair $(v_g, v_l) \in \mathcal{V}_d \times \mathcal{V}_l$. Similarly, a state of P_s is a triple $(v^{\mathbf{1}}, v^{\mathbf{0}}, v_l) \in \mathcal{V}^{\mathbf{1}} \times \mathcal{V}^{\mathbf{0}} \times \mathcal{V}_l$. Then, the following holds.

Theorem 1. *For every $i \geq 1$, state (v_g, v_l) is reachable at the start of the i-th execution of $\widetilde{Safety}()$ in $[\![P_d]\!]$ iff: (a) i is odd and state $(v^{\mathbf{1}}, v^{\mathbf{0}}, v_l)$ is reachable at the start of the $\lceil \frac{i}{2} \rceil$-th execution of $Safety^{\mathbf{1}}()$ in P_s such that $v_g \approx v^{\mathbf{1}}$; or (b) i is even and state $(v^{\mathbf{1}}, v^{\mathbf{0}}, v_l)$ is reachable at the start of the $\frac{i}{2}$-th execution of $Safety^{\mathbf{0}}()$ in P_s such that $v_g \approx v^{\mathbf{0}}$.*

Proof. The proof is by induction over i. For brevity, we only give an outline. The base case ($i = 1$) follows from the definitions of $\widetilde{Init}()$, *CopyGlobals* (cf. (1)) and $\ddot{Init}()$ (cf. (3)). For the inductive step, suppose i is odd and (v_g, v_l) is reachable at the start of the i-th execution of $\widetilde{Safety}()$ in $[\![P_d]\!]$. By inductive hypothesis, $(v^{\mathbf{1}}, v^{\mathbf{0}}, v_l)$ is reachable at the start of the $\lceil \frac{i}{2} \rceil$-th execution of $Safety^{\mathbf{1}}()$ in P_s such that $v_g \approx v^{\mathbf{1}}$. Since, *Safety* does not modify global or local variables, $[\![P_d]\!]$ next executes statement $X_d = \langle \rho_0; \ldots; \rho_{n-1}; CopyGlobals \rangle$ from state (v_g, v_l). Suppose it reaches state (v'_g, v'_l). Also, from the definition of *CopyFwd*, we know that P_s next executes statement $X_s = \langle \rho^{\mathbf{1}}_0(); \ldots; \rho^{\mathbf{1}}_{n-1}() \rangle$ from state $(v^{\mathbf{1}}, v^{\mathbf{1}}, v_l)$. It can be shown that after executing X_s, P_s can also reach a state $(v^{\mathbf{1}}, v'^{\mathbf{1}}, v_l)$ such that $v'_g \approx v'^{\mathbf{1}}$. Similarly, suppose that after executing statement X_s, P_s reaches state $(v^{\mathbf{1}}, v'^{\mathbf{1}}, v_l)$. Again it can be shown that after executing statement X_d, $[\![P_d]\!]$ can also reach state (v'_g, v'_l) such that $v'_g \approx v'^{\mathbf{1}}$. This establishes the result for $i+1$. By a symmetric argument, we can show that the result holds for the case when i is even as well. ☐

Correctness of SEQDBL. Recall that function *Safety* reads local variables only. Thus, $\widetilde{Safety}() = Safety^{\mathbf{1}}() = Safety^{\mathbf{0}}()$. By Theorem 1, $[\![P_d]\!]$ executes $\widetilde{Safety}$ from a state (v_g, v_l) iff P_s executes $Safety^{\mathbf{1}}$ or $Safety^{\mathbf{0}}$ from a state $(v^{\mathbf{1}}, v^{\mathbf{0}}, v_l)$. Hence, $[\![P_d]\!]$ aborts iff P_s also aborts, proving that SEQDBL is correct.

Note that both SEQSEM and SEQDBL rely crucially on our assumption of SNMOC. However, in practice, networks in our domain of interest are asynchronous with unbounded message delays, and SNMOC must be implemented on top of it in order to deploy DASL applications. This is the topic of Section 5.

5 Implementing SNMOC

The synchronous network abstraction (SNMOC) is implemented on top of an asynchronous network via a "synchronizer" [3] protocol. In the literature, several synchronizers [16] have been proposed. Many, such as PALS [17], rely on clock synchronization. However, this is not appropriate for our target domain where networks have unbounded latency. To address this challenge, we have developed a new synchronizer that does not rely on any clock synchronization. Instead, our protcol, called 2-Barrier-Synchronization (2BSYNC), uses global variables to enforce a *barrier* before and after each round, thereby synchronizing rounds across all the application nodes. We now present 2BSYNC in more detail.

Consider a DASL program $P_d = (GV, LV, \rho, n, \varphi)$. Let W_i be the set of global variables written by node N_i, i.e., $W_i = \{v[j] \mid \mathcal{W}(v, j) = i\}$. We introduce n additional global "barrier" variables – $b_0, \ldots, b_{n-1}$ – each initialized to 0. For any set of global variables X, let $(X)!$ denote the *atomic broadcast* of the current value of all variables in X to other nodes. This means that the broadcasted values are received by other nodes atomically, i.e., at any point in time, either all of them are visible to a recipient node or none of them are. The atomic broadcast capability is crucial for implementing 2BSYNC, and we discuss it further later. Then, node N_i is implemented by the program $Node_i$ defined as follows (b_i++ is a shorthand for $b_i = b_i + 1$):

$$\begin{aligned} &Node_i = \text{WHILE}(\text{TRUE}, Round_i), \text{ where} \\ &Round_i = \langle b_i\text{++}; (W_i, b_i)!; Barr_i(); \rho_i(); b_i\text{++}; (b_i)!; Barr_i()\rangle, \text{ where} \\ &Barr_i = \text{WHILE}(b_0 < b_i \lor \cdots \lor b_{n-1} < b_i, skip) \end{aligned} \tag{4}$$

Note that $Barr_i$ implements a barrier since it forces $Node_i$ to wait till the values of the barrier variables at all other nodes have "caught up" with the value of its own barrier variable b_i.

Correctness of 2BSYNC. For any global array element $v[j]$, let $r(v[j], i, k)$ and $w(v[j], i, k)$ be the value of $v[j]$, before and after respectively, the execution of $\rho_i()$ during the k-th iteration of the outermost WHILE loop of $Node_i$. Let $\mathcal{I}(v[j])$ be the initial value of global array element $v[j]$ at its writer node. Thus, 2BSYNC is correct iff the following two conditions hold:

$$\forall v[j] \centerdot \forall i \in [0, n) \centerdot r(v[j], i, 1) = \mathcal{I}(v[j]) \tag{5}$$

$$\forall v[j] \centerdot \forall i \in [0, n) \centerdot \forall k > 1 \centerdot r(v[j], i, k) = w(v[j], \mathcal{W}(v, j), k - 1) \tag{6}$$

Let $\mathcal{B}(v[j], i, k)$ be the value of $v[j]$ broadcast atomically during the k-th iteration of the outermost WHILE loop of $Node_i$ in (4). Note that $\mathcal{B}(v[j], i, 1) = \mathcal{I}(v[j])$ and $\forall k > 1 \centerdot \mathcal{B}(v[j], i, k) = w(v[j], i, k - 1)$. Thus, (5) and (6) hold iff:

$$\forall v[j] \centerdot \forall i \in [0, n) \centerdot \forall k \geq 1 \centerdot r(v[j], i, k) = \mathcal{B}(v[j], \mathcal{W}(v, j), k) \tag{7}$$

Then, (7) follows from two observations. Due to the first $Barr_i$:

$$\forall v[j] \centerdot \forall i \in [0, n) \centerdot \forall k \geq 1 \centerdot r(v[j], i, k) = \mathcal{B}(v[j], \mathcal{W}(v, j), k') \implies k \leq k'$$

Again, due to the second $Barr_i$, we have:

$$\forall v[j] \centerdot \forall i \in [0, n) \centerdot \forall k \geq 1 \centerdot r(v[j], i, k) = \mathcal{B}(v[j], \mathcal{W}(v, j), k') \implies k' < k + 1$$

This completes the proof. Note that the 2BSYNC protocol must be implemented over a middleware that supports global variables as well as atomic broadcast. For this research, we use MADARA [8], a middleware developed for distributed AI applications. The support for global variables was already available in MADARA. We augmented it by implementing the atomic broadcast capability. In Section 6 we describe the process of generating C++ code for each node of a DASL program against the MADARA API.

6 Code Generation: From DASL to MADARA/C++

Once a DASL program P_d has been successfully verified, it is converted into an equivalent MADARA application P_m. MADARA is an open-source[2] middleware developed for distributed AI applications. It has been ported to a variety of real-world platforms and architectures (e.g., ARM and Intel) and operating systems (e.g., Linux, Windows, Android and iOS). MADARA applications can communicate via IP-based protocols like UDP, IP broadcast and IP multicast or the Data Distribution Service (DDS). These advantages are inherited by P_m by virtue of its use of MADARA. MADARA ensures consistency of global variables (GV) within P_m through a distributed context that maps variables to values, with each $v \in GV$ controlled by a private Lamport clock v_t, which enforces temporal consistency. This type of consistency is inherent in the underlying MADARA subsystems, and is useful for encoding the 2BSYNC protocol into the P_m program.

MADARA has two additional features crucial for implementing 2BSYNC. First, as part of this research, we augmented MADARA with a `sendlist` mechanism that allows application nodes to dynamically specify, at runtime, which variables in GV are disseminable immediately, and which variable disseminations should be delayed until later. This `sendlist` mechanism maps directly to the requirements of the 2BSYNC protocol (cf. Sec. 5). Specifically, we use it to enable barrier variable updates while actively suppressing the dissemination of other values written by node N_i until the time is appropriate. This is required to perform the atomic broadcast operation $(b_i)!$ in (4). Second, MADARA allows an application node to broadcast values of multiple context variables to other nodes as a "packet". MADARA ensures that the packet is received by other nodes "atomically", i.e., at any point in time, either all the values in the packet are observed by a receiver node, or none is. This is required to perform the atomic broadcast operation $(W_i, b_i)!$ in (4).

The generated program P_m preserves the semantics of the DASL program P_d that has been verified via sequentialization to P_s. The differences between P_m and P_d revolve around the following limitations and features of MADARA:

1. MADARA supports several first class types like strings, doubles, raw binary, and images but only one type of integer (a 64 bit integer). Consequently, Booleans and integers in P_d are encoded as 64 bit integers in P_m.
2. MADARA includes an efficient scripting environment for manipulating global variables (GV). It also provides classes – *Integer*, *Array*, *Array_N*, etc. – that allow direct access to GV. We use the scripting environment wherever applicable, such as in the implementation of the 2BSYNC protocol. However, for user-defined functions, we generate code that uses the classes. This leads to a more direct mapping from P_d to P_m, especially for control statements such as if/then/else and switch statements. The MADARA equivalents of these control structures use logical operators like && and ||, and the class facades into the MADARA context yields P_m code that is easier to debug and modify, without requiring expertise about MADARA internals.

[2] `http://madara.googlecode.com`

```
// Source model in P_d
EXISTS_LOWER(idp,lock[idp])
...
// Generated code in (*$P_m$*)
(id == 1 && lock[0]) ||
(id == 2 && (lock[0] ||  lock[1])) ||
(id == 3 && (lock[0] ||
lock[1] || lock[2]))
```

```
// Source model in P_d
2BSYNC for 2 processes
...

// Generated code in P_m
if (id == 0)
  settings.send_list ["B.0"]
    = true;
else
  settings.send_list ["B.1"]
    = true;

// Continued on the right
while (1)
{
  knowledge.evaluate("++B.{.id}");
  if (id == 0)
    knowledge.wait("B.1 >= B.0");
  else
    knowledge.wait("B.0 >= B.1");

  ROUND ();

  knowledge.evaluate("++B.{.id}");
  if (id == 0)
    knowledge.wait("B.1 >= B.0",settings);
  else
    knowledge.wait("B.0 >= B.1",settings);
}
```

Fig. 3. P_m code generated from: (top) EXISTS_LOWER; (bottom) 2BSYNC.

3. The MADARA context is appropriate for storing GV and LV but does not contain primitives that allow a node to perform omniscient variable accesses (i.e., to variables of other nodes) present in DASL programs, specifically in the *Init* and *Safety* functions (cf. Fig. 1). Because each node of P_m only has access to its own local variables, P_m does not contain code for *Init* or *Safety*. This makes sense since these two functions are meant for verification only. Still, for verification results to be valid, the initial state of P_m must be consistent with that constructed by *Init*. Currently, this is ensured manually.
4. Unlike the sequentialized program P_s, MADARA allows us to build a P_m that is *id*-neutral at compilation time. Through the usage of MADARA's object-oriented facades into the GV and LV contexts, a more direct mapping of the source P_d to P_m takes place. While the sequentialized program P_s contains separate code for each node of P_d, the application P_m consists of code for a single node whose *id* is supplied via a command line argument.

Fig. 3 illustrates examples of the code generation from sections of the P_d defined in Fig. 1. The examples outline the code unrolling of `EXISTS_LOWER` (top) and 2BSYNC (bottom), respectively. Note that variables `B.0` and `B.1` in Fig. 3 correspond to variables b_0 and b_1 in (4).

7 Empirical Evaluation

We implemented DIVER in a verifying compiler called DASLC, and used it to compare SEQSEM and SEQDBL on a set of synchronous distributed applications. All our experiments were done on a 8 core 2GHz machine running Ubuntu 12.04 with a time limit of 1 hour and a memory limit of 16GB. The parser for DASL programs was generated using `flex/bison`. The rest of DASLC was implemented in C++. DASLC generates ANSI C code – the safety property is

MUTEX-OK						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n=6$		$n=8$		$n=10$	
60	406	396	1116	1051	2388	2268
80	850	806	2268	1967	4525	4249
100	1404	1381	3584	3452	7092	6764
	μ=1.040 σ=0.038					

MUTEX-BUG1					
T_S	T_D	T_S	T_D	T_S	T_D
$n=6$		$n=8$		$n=10$	
184	175	517	439	1068	959
402	372	1013	925	2203	1812
734	686	1726	1566	3513	3287
μ=1.056 σ=0.060					

MUTEX-BUG2					
T_S	T_D	T_S	T_D	T_S	T_D
$n=6$		$n=8$		$n=10$	
233	216	637	553	1292	1167
500	462	1218	1112	2602	2139
890	838	2056	1860	4216	3742
μ=1.065 σ=0.056					

3DCOLL-OK-4x4						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n=2$		$n=4$		$n=6$	
10	13	10	59	40	219	96
20	37	31	351	123	1014	480
30	48	48	406	202	–	–
	μ=2.213 σ=0.715					

3DCOLL-OK-7x7					
T_S	T_D	T_S	T_D	T_S	T_D
$n=2$		$n=4$		$n=6$	
31	35	323	148	1099	323
73	72	1262	401	–	–
142	113	–	–	–	–
μ=2.294 σ=0.763					

3DCOLL-BUG-4x4					
T_S	T_D	T_S	T_D	T_S	T_D
$n=2$		$n=4$		$n=6$	
8	9	49	36	123	96
24	36	119	101	410	210
42	44	206	155	–	–
μ=1.615 σ=0.425					

3DCOLL-BUG-7x7					
T_S	T_D	T_S	T_D	T_S	T_D
$n=2$		$n=4$		$n=6$	
22	23	194	114	–	–
57	76	–	–	–	–
117	134	–	–	–	–
μ=1.514 σ=0.344					

2DCOLL-OK-4x4						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n=2$		$n=4$		$n=6$	
10	17	25	87	262	280	831
20	123	271	1474	2754	–	–
30	863	1301	–	–	–	–
	μ=0.446 σ=0.118					

2DCOLL-BUG1-4x4					
T_S	T_D	T_S	T_D	T_S	T_D
$n=2$		$n=4$		$n=6$	
3	2	12	11	30	22
8	7	36	29	80	75
12	15	57	51	144	105
μ=1.282 σ=0.264					

2DCOLL-BUG2-4x4					
T_S	T_D	T_S	T_D	T_S	T_D
$n=2$		$n=4$		$n=6$	
4	3	13	11	30	29
8	9	33	33	76	66
16	21	57	77	150	120
μ=1.056 σ=0.266					

2DCOLL-OK-7x7						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n=2$		$n=4$		$n=6$	
10	74	146	395	1016	1707	–
20	1726	3096	–	–	–	–
30	–	–	–	–	–	–
	μ=0.598 σ=0.202					

2DCOLL-BUG1-7x7					
T_S	T_D	T_S	T_D	T_S	T_D
$n=2$		$n=4$		$n=6$	
7	7	32	24	101	70
15	22	94	55	345	150
40	35	180	91	–	223
μ=1.382 σ=0.517					

2DCOLL-BUG2-7x7					
T_S	T_D	T_S	T_D	T_S	T_D
$n=2$		$n=4$		$n=6$	
5	10	26	36	188	113
19	22	71	113	207	166
46	68	124	295	416	235
μ=0.906 σ=0.393					

Fig. 4. Experimental Results; T_S, T_D = verification time with SEQSEM, SEQDBL; n = no. of nodes; R = no. of rounds; $G \times G$ = grid size; μ, σ = mean, standard deviation of T_S/T_D for all experiments in that category; – denotes out of time/memory.

encoded by assertions – which we verify using the model checker CBMC [6] v4.7. CBMC converts the target C program *Prog* and assertion *Asrt* into a propositional formula φ such that *Prog* violates *Asrt* iff φ is satisfiable. It then solves φ using an off-the-shelf SAT solver. We use the parallel SAT solver PLINGELING (`http://fmv.jku.at/lingeling`) to utilize multiple cores. Since CBMC only verifies bounded programs, we fixed the number of rounds of execution of the target application for each verification run. Due to lack of space, we only present a subset of results that suffice to illustrate our main conclusions. Our tools, benchmarks, and complete results are available at `http://www.contrib.andrew.cmu.edu/~schaki/misc/models14.zip`. We verified several applications, varying number of nodes (n) and rounds (R), and using both SEQSEM and SEQDBL. We now present our results in detail.

Mutual Exclusion. The first application implemented a distributed mutual exclusion protocol. The DASL program for the correct version of this protocol is in Fig. 1. We also implemented two buggy versions of this protocol by omitting important checks (at lines 14–15 and lines 19–20 in Fig. 1). Results of verifying all three versions are shown in Fig. 4. As expected, verification time increases both with n and R. However, it is almost the same between SEQSEM and SEQDBL for a fixed n and R, as shown by the values of μ and σ. This indicates that the techniques implemented in CBMC and PLINGELING effectively eliminate the complexity due to additional variables produced by SEQSEM.

3-Dimensional Collision Avoidance. The next application implemented a collision avoidance protocol where nodes (denoting robots flying over an area demarcated by a two-dimensional grid) are able to change their height to avoid colliding with each other. We implemented a correct and a buggy version of this protocol. The results of verifying the two versions are shown in Fig. 4. Again, verification time increases with n, R, and G (where grid-size = $G \times G$). In addition, programs generated by SEQDBL are verified faster (over 100% for the correct version and 50% for the buggy version) than those generated by SEQSEM, for a fixed n, R and G. This supports our intuition that the $\mathcal{O}(n)$ variables used by SEQDBL is better for verification.

2-Dimensional Collision Avoidance. The final application implemented a collision avoidance protocol where nodes can only move in two dimensions. We implemented a correct and two buggy versions of this protocol. The results of verifying them are shown in Fig. 4. Again, verification time increases with n, R, and G. However, the difference between SEQDBL and SEQSEM is subtle. For the BUG2 version, they are almost identical. For BUG1, SEQDBL leads to over 30% faster verification. However, for the correct version, SEQSEM allows verification to be 40% faster, even though it generates programs with more variables.

In summary, while SEQDBL is clearly the better option overall, there are cases where SEQSEM is more efficient. We believe that the optimizations and symbolic algorithms used by modern model checkers means that verification time is not just determined by the number of variables. While these results were obtained using CBMC, we believe that they are representative of symbolic model checkers. For example, similar non-monotonic performance has also been observed in other contexts, e.g., when comparing [21] BDD and SAT-based LTL model checkers. Note that, in general, model checking a buggy application is easier than a correct one since the latter requires complete statespace exploration.

8 Conclusion

We presented an approach for model-driven verifying compilation of distributed applications written in a domain-specific language, called DASL, against user-provided safety specifications. We assume a "synchronous network" model of computation. Our verification is based on sequentialization followed by software model checking. We develop two sequentialization techniques – SEQSEM and SEQDBL– and compare them on a set of applications. SEQDBL produces programs with fewer variables, and empirically is more efficient for verification in most cases. We also develop a protocol to implement a synchronous network abstraction over an asynchronous network. This protocol does not require clock synchronization and is of independent interest. We believe that extending our approach to handle asynchronous and fault-tolerant programs, and proving correctness of code generation and middleware, are important directions to pursue.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In: Sharygina, N., Veith, H. (eds.) Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13). Lecture Notes in Computer Science, vol. 8044, pp. 141–157. Springer-Verlag, Saint Petersburg, Russia (July 2013)
2. Andrews, T., Qadeer, S., Rajamani, S., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: Alur, R., Peled, D. (eds.) Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04). Lecture Notes in Computer Science, vol. 3114, pp. 484–487. Springer-Verlag, Boston, MA (July 2004)
3. Awerbuch, B.: Complexity of Network Synchronization. Journal of the ACM (JACM) 32(4), 804–823 (October 1985)
4. Azimi, S.R., Bhatia, G., Rajkumar, R., Mudalige, P.: Reliable intersection protocols using vehicular networks. In: Lu, C., Kumar, P.R., Stoleru, R. (eds.) Proceedings of the 4th International Conference on Cyber-Physical Systems (ICCPS '13). pp. 1–10. Association for Computing Machinery, Philadelphia, PA, USA (April 2013)
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A Declarative Language for Programming Synchronous Systems. In: Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '87). pp. 178–188. Association for Computing Machinery, Munich, Germany (January 1987), http://doi.acm.org/10.1145/41625.41641
6. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer-Verlag, Barcelona, Spain, March 29–April 2, 2004. New York, NY (March–April 2004)
7. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning Assumptions for Compositional Verification. In: Garavel, H., Hatcliff, J. (eds.) Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03). Lecture Notes in Computer Science, vol. 2619, pp. 331–346. Springer-Verlag, Warsaw, Poland, April 7–11, 2003. New York, NY (April 2003)
8. Edmondson, J.R., Gokhale, A.S.: Design of a Scalable Reasoning Engine for Distributed, Real-Time and Embedded Systems. In: Xiong, H., Lee, W.B. (eds.) Proceedings of the 5th International Conference on Knowledge Science, Engineering and Management (KSEM '11). Lecture Notes in Computer Science, vol. 7091, pp. 221–232. Springer-Verlag, Irvine, CA, USA, December 12-14, 2011 (December 2011)
9. Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K., Lange, K.: AUTOSAR–A Worldwide Standard is on the Road. In: Proceedings of the 14th International VDI Congress Electronic Systems for Vehicles. Baden-Baden, Germany (2009)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, London (1985)
11. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. Journal of the ACM (JACM) 50(1), 63–69 (January 2003)
12. Humphrey, L., Wolff, E., Topcu, U.: Formal Specification and Synthesis of UAV Mission Plans. In: Proc. of AAAI Symposium. Baden-Baden, Germany (November 2014)

13. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys (CSUR) 41(4) (2009)
14. Lal, A., Reps, T.W.: Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In: Gupta, A., Malik, S. (eds.) Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08). Lecture Notes in Computer Science, vol. 5123, pp. 37–51. Springer-Verlag, Princeton, NJ, USA, July 7-14, 2008. New York, NY (July 2008)
15. Lasnier, G., Zalila, B., Pautet, L., Hugues, J.: Ocarina : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In: Kordon, F., Kermarrec, Y. (eds.) Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies - Ada-Europe 2009. Lecture Notes in Computer Science, vol. 5570, pp. 237–250. Springer-Verlag, Brest, France (June 2009)
16. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
17. Miller, S.P., Cofer, D.D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proceedings of the 28th Digital Avionics Systems Conference (DASC '09). pp. 1.A.3–1–1.A.3–12. IEEE Computer Society, Orlando, FL, USA (October 2009)
18. Milner, R.: Communication and Concurrency. Prentice-Hall International, London (1989)
19. Pardo-Castellote, G.: OMG Data-Distribution Service: Architectural Overview. In: Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003 Workshops). pp. 200–206. IEEE Computer Society, Providence, RI, USA (May 2003)
20. Schmidt, D.C., Gokhale, A., Natarajan, B., Neema, S., Bapty, T., Parsons, J., Gray, J., Nechypurenko, A., Wang, N.: CoSMIC: An MDA generative tool for distributed real-time and embedded component middleware and applications. In: Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture. Seattle, WA, USA (2002)
21. Schuppan, V., Darmawan, L.: Evaluating LTL Satisfiability Solvers. In: Bultan, T., Hsiung, P.A. (eds.) Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA '11). Lecture Notes in Computer Science, vol. 6996, pp. 397–413. Springer-Verlag, Taipei, Taiwan (October 2011)
22. Siegel, J.: CORBA 3 fundamentals and programming, vol. 2. John Wiley & Sons Chichester (2000)
23. Sulistio, A., Yeo, C.S., Buyya, R.: A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. Softw., Pract. Exper. 34(7), 653–673 (June 2004)
24. Torre, S.L., Madhusudan, P., Parlato, G.: Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: Bouajjani, A., Maler, O. (eds.) Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09). Lecture Notes in Computer Science, vol. 5643, pp. 477–492. Springer-Verlag, Grenoble, France, June 26 - July 2, 2009. New York, NY (July 2009)
25. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincentelli, A.L., Caspi, P., Natale, M.D.: Implementing Synchronous Models on Loosely Time Triggered Architectures. IEEE Transactions on Computers (TC) 57(10), 1300–1314 (October 2008)