Assume-Guarantee Reasoning with Scheduled Components

Cong Liu[1], Junaid Babar[1], Isaac Amundson[1], Karl Hoech[1], Darren Cofer[1], and Eric Mercer[2][0000−0002−2264−2958]

[1] Applied Research and Technology, Collins Aerospace, USA
{cong.liu,junaid.babar,isaac.amundson,karl.hoech,darren.cofer}@collins.com
[2] Brigham Young University, USA
egm@cs.byu.edu

Abstract. Contract-based assume-guarantee reasoning can be used to improve the scalability of model checking by decomposing complex verification problems. In previous work, we demonstrated this approach for systems modeled using the Architecture Analysis and Design Language (AADL) assuming a synchronous model of computation. This allows nondeterministic ordering of parallel components and generally results in an over-approximation of real behavior. This paper describes an approach to incorporating an execution schedule in the assume-guarantee reasoning. We define our semantic interpretation of contracts when components are executed according to this schedule, more accurately reflecting the behavior of the system implementation. We introduce virtual scheduling events which tie AADL timing and execution semantics to contracts. A case study based on a simple unmanned air vehicle surveillance system is provided to illustrate our approach.

Keywords: assume-guarantee · compositional verification · model checking · model based system engineering · AADL · scheduling semantics

1 Introduction

Formal verification of cyber-physical systems can be a daunting task due to the *state explosion problem* [5]. We tackle this challenge from two angles. First, we use a compositional verification technique [23] [6] to decompose the reasoning on the global state space into a number of localized problems for each component separately. The system proof is constructed from the individual component proofs. Second, we assume that the components execute in a static sequential order. We do not consider all possible execution orders; in other words, nondeterminism due to scheduling decisions is excluded. In fact, in many safety-critical applications the actual implementation executes according to a predefined schedule [2] to achieve real-time performance requirements.

Previous work has not incorporated component execution times or ordering imposed by a component execution schedule. As a result, an analysis performed at the model level may produce results that deviate from the actual behavior of

the system implemented from the model. Our objective is to refine our compositional verification approach to capture this aspect of the design and ensure that analysis results faithfully represent the system implementation.

The Architecture Analysis and Design Language (AADL) was developed to capture the important design concepts in distributed real-time embedded systems [10]. AADL captures both the hardware and software architecture in a hierarchical format, offering a high degree of flexibility and supporting incremental development in which an architecture is refined to add increasing levels of detail.

In AADL, an architecture model includes component interfaces, connections, and execution characteristics, but not component implementations. It describes the interactions between components and their arrangement in the system, but the lowest level components themselves are "black boxes." Their implementations must be described separately using model-based behavioral specification languages or traditional programming languages, which may be included by reference in the architecture model.

In previous work, we developed the Assume Guarantee Reasoning Environment (AGREE) [8], a language and tool for compositional verification of AADL models. The behavior of a model is described by *contracts* [4] specified for each component. A contract contains a set of *assumptions* about the component's inputs and a set of *guarantees* about the outputs. The guarantees of a component must be true provided that the component's assumptions are true. The goal of an AGREE analysis is to prove that each component's contract is entailed by the contracts of its subcomponents. Guarantees on a leaf-level component must be verified to hold by its implementation.

AGREE was originally developed to reason about systems that execute synchronously. These systems have straightforward translations to *Lustre* [13], a synchronous dataflow language interpreted by the model checkers used by AGREE. However, many systems that are modeled in AADL do not behave synchronously. Ideally one can implement a communication protocol between components, such as Physically Asynchronous Logically Synchronous (PALS) [20], that allows the abstraction of synchronous communication to be sound. However, for many systems this is not the case.

In this paper, we extend the AGREE framework to enable the verification of *scheduled* AADL models. We introduce virtual scheduling events, which tie AADL timing and scheduling semantics to AGREE contracts. This enhancement enables AGREE to take the software execution schedule into account in the analysis. Furthermore, it enables formal verification of a new class of embedded system architectures.

This paper is organized as follows. First we illustrate the motivation of our work using simple examples in Section 2. We then provide an informal description of our interpretation of the scheduling semantics in Section 3, followed by formal definitions of the model in Section 4. We present the modeling of the semantics in the AGREE AADL annex and Lustre backend in Section 6 and Section 7, respectively. We demonstrate usage of the model in a case study in Section 8.

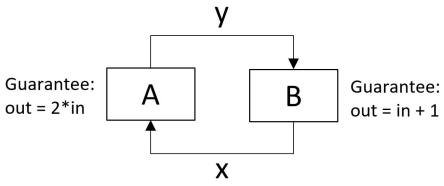

Fig. 1. A Simple Feedback System

Related work is presented in Section 9. We discuss our conclusion and future work in Section 10.

2 Motivating Examples

First, we will illustrate the key semantic difference between the synchronous model used in the original AGREE framework and the proposed model. Consider an AADL model that consists of two threads A and B, as shown in Figure 1. All ports are data ports. The behavior of each thread is indicated by its AGREE contract. The output of thread A is double its input and the output of thread B increments its input by one. By the synchronous semantics, the value of signal x and y at computation step n is defined by the solution to the two equations $y_n = 2x_n$ and $x_n = y_n + 1$, for all $n \in N$. This results in $x = (-1, -1, \dots)$, $y = (-2, -2, \dots)$ for all time. However, if the two threads execute in a sequential order ($ABAB$...), letting x_0, y_0 denote the initial value of x and y, respectively, an intuitive interpretation of the execution semantics is $y_1 = 2x_0, x_1 = y_1 + 1, y_2 = 2x_1$.... If $x_0 = 0$ and $y_0 = 0$, this results in $x = (0, 1, 3, 7, \dots)$ and $y = (0, 0, 2, 6, \dots)$. The example shows that the behavior of a synchronous model is defined by the solution(s) to systems of mathematical equations (or inequalities) at each instant, while the behavior of the scheduled components is defined through iterations over time.

We are aware that the Lustre compiler rejects all syntactic loops. A one-step delay (the pre operator) could be added between A and B, resulting in an implied schedule and legal Lustre code. Since an AGREE analysis does not compile the generated Lustre code but instead interprets it via one of the underlying model checkers, we do not face the same limitation and can compute a solution for synchronous execution whenever one exists.

Now consider an AADL model that consists of four threads A, B, C, D, as shown in Figure 2. Again, all ports are data ports. Thread A outputs the sequence of all natural numbers. Threads B and C simply copy their inputs to their outputs. Thread D subtracts the second (bottom) input value from the first (top) input value. Given a schedule $(ACABD)^*$, suppose we want to prove that the primary output d is a sequence of ones (ignoring the initial prefix). This can be achieved with the proposed model, since thread B only copies even numbers, and thread C only copies odd numbers. However, it cannot be proved directly

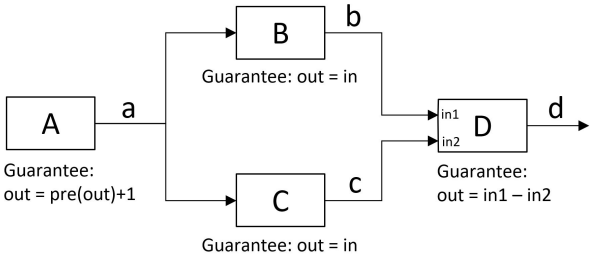

Fig. 2. A Simple Downsampling System

```
thread A
    features
        Input: in data port Base_Types::Integer;
        Output: out data port Base_Types::Integer;
    properties
        Dispatch_Protocol => Periodic;
        Period => 500ms;
        Compute_Execution_Time => 6ms .. 8ms;
    annex agree {**
        assume "A input": Input > prev(Input, 0);
        guarantee "A output": Output = Input + prev(Output, 0);
    **};
end A ;
```

Fig. 3. A Simple Integrator AADL Model in AGREE

with the synchronous model, where d is a sequence of zeroes. Note that in the example, threads B and C essentially downsample the data stream from thread A. To model this kind of behavior, we require a mechanism significantly more complex than *delays*.

Note that if the schedule is $(ABCD)^*$, the output d is a sequence of zeroes (ignoring the initial prefix), matching the behavior of the synchronous model. This indicates that the execution order could have an impact on the system behavior. As we will show later, our model is not a variant of Kahn Process Network [15], like Lee's Synchronous Dataflow [18], where any execution order results in the same system behavior. Therefore, it makes sense to tie a system-level property to a specific schedule of the components.

3 Overview of the Model

We now discuss in detail our semantic interpretation of AGREE contracts on scheduled components. Consider the AADL model of an integrator, shown in Figure 3. We assume that an execution time slot is assigned to the thread. The first question that we face is when the contracts shall hold. In a synchronous model, contracts hold at every instant. However, with scheduled execution, it is reasonable to assume that the contract may not hold when the component is not activated. But once it is activated, shall a contract hold throughout the entire execution or just at certain instants? Second, how shall *Input* (referred to in the

contract) be interpreted? One interpretation is that it refers to the input value at the time when the contract is evaluated, which may vary during the execution. Another interpretation is that it refers to the input value when the component starts its execution. In other words, there is a notion of *sample and hold*. This interpretation is consistent with the *frozen* inputs described in the AADL V2 standard. Third, how shall the *prev* operator be interpreted? In a synchronous model, it refers to the previous instant. However, with scheduled execution, it seems reasonable to interpret *prev* as the previous activation (i.e. the value when the component was last activated). If the contracts hold throughout the activation, a more sensible interpretation is that at the first instant during activation, it refers to the previous activation. Then at each following instant, it refers to the last updated value in the current activation. This interpretation is adopted in the *activation condition* in SCADE [9] and the *clock* mechanism in SIGNAL [3].

We believe that AGREE contracts are intended to model requirements [26], not implementations. Guarantees model the component requirements, and assumptions model the environmental constraints that are used to verify the component requirements. Following the AADL *input-compute-output* model, assumptions are said to hold at the start of the execution (i.e. *dispatch*) when the inputs are read. The guarantees shall be satisfied at the end of the execution (i.e. *complete*) when the outputs are written. This interpretation has a few implications. First, since we adopt the AADL *frozen inputs* concept, any reference to *Input* refers to the input value that was read in at dispatch. Second, a component's assigned time slot does not necessarily exactly match its execution time window. If the time slot is greater than its execution time, we interpret the start and end of the time slot as *dispatch* and *complete*, respectively. Otherwise, we claim that a *preemption* has occurred. Third, each contract is examined exactly once in each activation. Thus, we interpret the *prev* operator as the previous activation. Fourth, the guarantees are not models of the *transient* behavior during an execution. Instead, we interpret them as constraints on the *steady-state* outputs at the end of activation.

We assume that the requirements do not contain real-time constraints. Modeling such constraints in AGREE is discussed in [1]. However, this does not mean that AGREE contracts cannot model timer based requirements. In practice, a timer is usually implemented as a counter, whose limit (constant) is calculated based on the frequency of its execution. The counter is activated periodically and increments by only one during each activation, independent of the execution time. This is consistent with our interpretation.

Thus, for each component we introduce two distinctive events, *dispatch* and *complete*, to model the start and end of its activation, respectively. Similarly, for a system (consisting of components), the two events model the start and end of a scheduling cycle. The two events shall appear in pairs and alternate, with *dispatch* appearing before *complete*. We introduce the notion of *well-ordered* in Section 4 to capture this pattern.

In SCADE and SIGNAL, when a component is not activated, its outputs retain their previous values. We extend this output freeze time window to *complete* events, including activation. We understand that in practice the actual output values may change during activation. We choose this because we interpret the *guarantees* as steady-state requirements of outputs at *complete*. Output values between *dispatch* and *complete* are undefined. Thus, we model them using the last output values, so that the outputs are well-defined at every instant.

We inherit the same notion of composition used in the current AGREE framework. A connection between two components means their contracts refer to the same signal. This, combined with a schedule and the output freeze rule, essentially simulates communication based on *shared variables*. When the producer is not activated, its outputs hold the last values. When the consumer is activated, it reads the last values from the producer. The communication may also be viewed as a FIFO queue, where the queue size is one. This means the proposed model only supports limited AADL *event* data port communication.

We only consider single-machine schedules. The scheduler ensures that at most one component is activated at a time. For a preemptive schedule, we require that a component can only be preempted by another component if they do not have connections. Thus, there is no ambiguity on the order of read and write or the variable value referenced in the contracts.

We assume that the system-level inputs do not change values throughout a scheduling cycle. In practice, this means that there may exist a queue that holds the system-level input messages, which are periodically sampled by the components, or the inputs may come from another system, which is inactive while the system under consideration is active.

The input freeze rule may imply that the assumptions could be examined at *complete*, instead of *dispatch*. Thus, we may not really need the *dispatch* event. We keep it mainly for two reasons. First, the assumptions in general could depend on previous outputs. In our model, the outputs are updated at *complete*. So the output values at *dispatch* may be different from the values at the corresponding *complete*. Therefore, it is important to distinguish between the two events to avoid ambiguity of the output values. Second, keeping the pair (*dispatch*, *complete*) may help users to better understand the AGREE counterexample trace, particularly with a preemptive schedule.

The original schedule is often specified in the form of a sequence of time slots assigned to the components. The schedule could come from an AADL real-time scheduling tool such as Cheddar [25], or from a scheduler provided by an RTOS/Microkernel vendor, such as seL4 [17]. To properly model the schedule in AGREE, the component execution time has to be considered. Consider the example shown in Figure 4 with two scheduled components A and B. We refer to the original schedule and its model in AGREE as the real-time schedule and AGREE schedule, respectively. Given the same real-time schedule, due to the different execution time C_A of A, two different AGREE schedules are created. In Figure 4(a), since C_A is equal to the time slots assigned to A, the end of the each time slot is modeled as *complete*. In Figure 4(b), since the first time slot

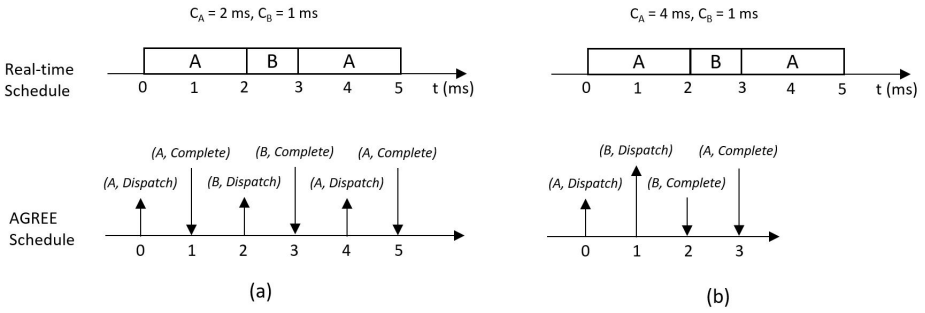

Fig. 4. Models of Real-Time Schedule in AGREE

assigned to A is less than its execution time, the end of the first time slot is interpreted as *preemption*, instead of *complete*.

4 Formal Definitions

In this section we formally define the proposed model.

Signal. A *signal* x is a function $x : N \to V$, where N is the set of natural numbers including zero, and V is a value set. A signal is *Boolean* if the value set is the Boolean domain. We use $x(n)$ to denote the value of signal x at instant n.

Port. Let Q be a set of ports. For each port $q \in Q$, a set V_q denotes the values that may be assigned to port q. A signal x_q at port q is a function $x_q : N \to V_q$. A *trace* σ_Q of Q is an assignment of a signal x_q to each port q in Q. We use Σ_Q to denote the set of all traces of Q. Given a set $Q' \subseteq Q$, the *projection* of a trace σ_Q onto Q' is the assignment of signal x_q to each port q in Q', as defined in σ_Q. We denote the projection as $\sigma_Q|_{Q'}$.

Dispatch and Complete. Two Boolean signals *dispatch* and *complete* are *well-ordered* if

1. $\forall n \in N, dispatch(n) \neq 1 \vee complete(n) \neq 1$,
2. $\forall n \in N, dispatch(n) = 1 \implies \exists m \in N, m > n, complete(m) = 1$,
3. $\forall m \in N, complete(m) = 1 \implies \exists n \in N, n < m, dispatch(n) = 1$,
4. $\forall n, m \in N, n < m, dispatch(n) = 1 \wedge dispatch(m) = 1 \implies \exists k \in N, n < k < m, complete(k) = 1$.
5. $\forall n, m \in N, n < m, complete(n) = 1 \wedge complete(m) = 1 \implies \exists k \in N, n < k < m, dispatch(k) = 1$.

The first condition requires that *dispatch* and *complete* are mutually exclusive. The second and third conditions state that *dispatch* and *complete* appear in pairs, and in each pair *dispatch* appears before *complete*. The fourth and fifth

conditions ensure that *dispatch* and *complete* appear alternately. From here on we only consider a well-ordered pair (*dispatch*, *complete*).

An *interval* δ of a pair (*dispatch*, *complete*) is a set of integers $[n, m] \cap N, n, m \in N, n < m$, satisfying:

1. $dispatch(n) = 1$,
2. $complete(m) = 1$,
3. $\forall k \in (n, m) \cap N, dispatch(k) = 0 \wedge complete(k) = 0$.

We denote the set of all such intervals as Δ.

Component. A (scheduled) *component* c is a tuple $(I_c, O_c, A_c, P_c, dispatch_c, complete_c)$, where:

- I_c is a finite set of ports, called *inputs*,
- O_c is a finite set of ports disjoint from I_c, called *outputs*,
- A_c and P_c are two past-time LTL formulas on a trace $\sigma_c \in \Sigma_{I_c \cup O_c}$, called *assumptions* and *guarantees*, respectively,
- ($dispatch_c$, $complete_c$) is a pair of *well-ordered* Boolean signals.

The *behaviors* of a component c are a set $\Sigma_c \subseteq \Sigma_{I_c \cup O_c}$, such that $\forall \sigma_c \in \Sigma_{I_c \cup O_c}, \sigma_c \in \Sigma_c$ if and only if the following propositions hold:

The *assumptions* hold at *dispatch*. That is,

$$dispatch_c(n) \implies (\sigma_c, n) \models A_c, \forall n \in N. \tag{1}$$

Inputs freeze between *dispatch* and *complete*. That is,

$$x(i) = x(j), \forall i, j \in \delta \cap N, \forall \delta \in \Delta, \forall x \in \sigma_c|_{I_c}. \tag{2}$$

The *guarantees* hold at *complete*. That is,

$$complete_c(n) \implies (\sigma_c, n) \models P_c, \forall n \in N. \tag{3}$$

Outputs freeze between *completes*. That is,

$$\neg complete_c(n) \implies y(n) = y(n-1), \forall n \in N, n > 0, \forall y \in \sigma_c|_{O_c}. \tag{4}$$

Equation 1, 2, 3, and 4 represent the specification of a scheduled component.

Connection. Two components $c, c', c \neq c'$ are said to be *connected* if

$$O_c \cap I_{c'} \neq \varnothing \vee O_{c'} \cap I_c \neq \varnothing. \tag{5}$$

Note that by definition the intersection of a component's inputs and outputs is empty. Thus, we forbid a component from connecting to itself.

Schedule. Let C be a finite set of components, a *schedule* ϕ of C with *length* $T \in N$ is a partial function $[1, T] \cap N \to C \times \{\mathsf{Dispatch}, \mathsf{Complete}\}$, where Dispatch and Complete are two strings, satisfying:

1. $\forall i \in \text{dom}\ \phi, c \in C, \phi(i) = (c, \mathsf{Dispatch}) \implies \exists j \in \text{dom}\ \phi, j > i, \phi(j) = (c, \mathsf{Complete})$,
2. $\forall j \in \text{dom}\ \phi, c \in C, \phi(j) = (c, \mathsf{Complete}) \implies \exists i \in \text{dom}\ \phi, i < j, \phi(i) = (c, \mathsf{Dispatch})$,
3. $\forall i, j \in \text{dom}\ \phi, i < j, c \in C, \phi(i) = (c, \mathsf{Dispatch}) \wedge \phi(j) = (c, \mathsf{Dispatch}) \implies \exists k \in \text{dom}\ \phi, i < k < j, \phi(k) = (c, \mathsf{Complete})$,
4. $\forall i, j \in \text{dom}\ \phi, i < j, c \in C, \phi(i) = (c, \mathsf{Complete}) \wedge \phi(j) = (c, \mathsf{Complete}) \implies \exists k \in \text{dom}\ \phi, i < k < j, \phi(k) = (c, \mathsf{Dispatch})$,
5. $\forall i, j \in \text{dom}\ \phi, i < j, c, c' \in C, c \neq c', \phi(i) = (c, \mathsf{Dispatch}) \wedge \phi(j) = (c', \mathsf{Dispatch}) \wedge \forall k \in \text{dom}\ \phi, i < k < j, \phi(k) \neq (c, \mathsf{Complete}) \implies c, c'$ are not connected,
6. $\forall i, j, k \in \text{dom}\ \phi, i < j < k, c, c' \in C, c \neq c', \phi(i) = (c, \mathsf{Dispatch}) \wedge \phi(j) = (c', \mathsf{Dispatch}) \wedge \phi(k) = (c, \mathsf{Complete}) \implies \exists n \in \text{dom}\ \phi, j < n < k, \phi(n) = (c', \mathsf{Complete})$.

The first four conditions ensure the pair (Dispatch, Complete) associated with a component is *well-ordered* in a schedule. The fifth condition allows a component to be *preempted* by another component if the two have no connection. The sixth condition ensures that the scheduling events of two components are interleaved in a proper order. A schedule is *minimal* if ϕ is a *total* function. This means that at each instant there is either a *dispatch* or a *complete*. A schedule is *fair* if ϕ is *surjective*. This means that every component is scheduled to execute at least once. If a schedule is minimal and non-preemptive, we could simplify the notation and denote the schedule as a function that maps $[1, |C|] \cap N$ to C, as shown in the previous examples.

Given a schedule ϕ of components C, the *dispatch* and *complete* signals of each component $c \in C$ are defined as follows: $\forall i \in N$,

$$dispatch_c^{\phi}(i) = \begin{cases} 1, & \text{if } \phi(i \mod T) = (c, \mathsf{Dispatch}) \\ 0, & \text{otherwise} \end{cases}, \tag{6}$$

$$complete_c^{\phi}(i) = \begin{cases} 1, & \text{if } \phi(i \mod T) = (c, \mathsf{Complete}) \\ 0, & \text{otherwise} \end{cases}. \tag{7}$$

System. A set C of components are said to be *compatible* if no two components share the same output. That is,

$$\forall c_i, c_j \in C, c_i \neq c_j, O_{c_i} \cap O_{c_j} = \varnothing. \tag{8}$$

A *system* S is a tuple $(C, \phi, I_s, O_s, A_s, P_s, dispatch_s, complete_s)$, where:

- C is a set of compatible, scheduled components,
- ϕ is a schedule of C,
- $I_s = \cup_{\forall c \in C} I_c - \cup_{\forall c \in C} O_c$,
- $O_s = \cup_{\forall c \in C} O_c$,
- A_s and P_s are two past-time LTL formulas on a trace $\sigma_s \in \Sigma_{I_s \cup O_s}$, called system-level *assumptions* and *guarantees*, respectively,

- $dispatch_s(i) = \begin{cases} 1, & \text{if } i \mod T = 1 \\ 0, & \text{otherwise} \end{cases}, \forall i \in N,$
- $complete_s(i) = \begin{cases} 1, & \text{if } i \mod T = 0 \\ 0, & \text{otherwise} \end{cases}, \forall i \in N, i > 0.$

We have $I_s \cup O_s = \cup_{\forall c \in C}(I_c \cup O_c)$. The *behaviors* of a system S are a set $\Sigma_s \subseteq \Sigma_{I_s \cup O_s}$, such that $\forall \sigma_s \in \Sigma_{I_s \cup O_s}$,

$$\sigma_s \in \Sigma_s \iff \forall c \in C, \exists \sigma_c \in \Sigma_c, \sigma_s|_{I_c \cup O_c} = \sigma_c. \tag{9}$$

Informally, a trace of a system's ports is a behavior of the system if and only if its projection onto any component's ports is a behavior of the component. This implies that a system behavior maps the connected ports to the same signal. We use δ_s to denote an *interval* of the pair $(dispatch_s, complete_s)$. And we use Δ_s to denote the set of all such intervals. Given a system S and a trace $\sigma_s \in \Sigma_{I_s \cup O_s}$, we define the following propositions:

The system-level *assumptions* hold at *dispatch*. That is,

$$dispatch_s(n) \implies (\sigma_s, n) \models A_s, \forall n \in N. \tag{10}$$

Inputs freeze between dispatch and complete. That is,

$$x(i) = x(j), \forall i, j \in \delta_s \cap N, \forall \delta_s \in \Delta_s, \forall x \in \sigma_s|_{I_s}. \tag{11}$$

The system-level *guarantees* hold at *complete*. That is,

$$complete_s(n) \implies (\sigma_s, n) \models P_s, \forall n \in N. \tag{12}$$

Equations 10–12 represent the system specification of a set of scheduled components. Our verification goal is to prove that the system behaviors satisfy the system specification. Note that we do not define the system output freeze rule. This is because (in our context) the system under consideration is always active. The rule would make sense in the assume-guarantee reasoning at a higher level in the hierarchy, where the system is viewed as a periodically activated component.

5 Assume-Guarantee Reasoning

Scheduled components lend themselves to hierarchical assume-guarantee reasoning in a manner similar to that in [26]. The verification conditions to prove a system of unscheduled components correct are formalized in *past-time linear temporal logic* (PLTL) [16]. The two PLTL operators necessary for the verification conditions are **G** (globally) and **H** (historically). These are defined over a trace of the system, π, and a moment of evaluation in the trace, i, as follows:

$$(\pi, i) \models \mathbf{G}(f) \iff \forall j \geq i, (\pi, j) \models f$$
$$(\pi, i) \models \mathbf{H}(f) \iff \forall 0 \leq j \leq i, (\pi, j) \models f$$

Globally is invariant from the current moment into the future and historically is invariant from the beginning to the current moment.

We define $\mathbb{I}_c$ to be the set of components providing input to some component c in the system, and we define $\mathbb{O}$ to be the set of components that provide the output for the system. An unscheduled system, $S = (I, O, A, P)$, is correct if and only if for all π and for all $i \geq 0$ the following holds:

$$\begin{aligned} \forall c \in C\ \mathbf{G}(\mathbf{H}(A \wedge \textstyle\bigwedge_{c' \in \mathbb{I}_c} P_{c'}) \implies A_c) \wedge \\ \mathbf{G}(\mathbf{H}(A \wedge \textstyle\bigwedge_{c' \in \mathbb{O}} P_{c'}) \implies P) \end{aligned}$$

The first condition checks the input assumptions on each component under the system assumptions and upstream component guarantees. The second checks the output guarantees of the system under the system assumptions and component guarantees providing the output. If both conditions hold, then the system is said to be *correct*, meaning that $\mathbf{G}(\mathbf{H}(A) \implies P)$ holds.

The verification conditions are extended to scheduled components by adding a notion of *dispatch* and *complete* to the verification conditions. We define a predicate $same(X)$ that is true in the first moment, and after that, true at any moment if and only if the signals in the set X are unchanged from the previous moment. We also define the predicate δ_c^ϕ to be true if the current moment is in a dispatch interval for the component c according the schedule.

The assumptions in a scheduled component must hold at dispatch, and the guarantees of the same component must hold at complete. A component also assumes that its inputs are invariant through the dispatch interval and it guarantees that the outputs are invariant between complete cycles. These requirements are captured in the following predicates where x is a component:

$$\begin{aligned} \mathbb{D}_x^\phi(A_x) &= \left[\left(dispatch_x^\phi \wedge A_x\right) \vee \left(\delta_x^\phi \wedge same(I_x)\right)\right] \\ \mathbb{C}_x^\phi(P_x) &= \left[\left(complete_x^\phi \wedge P_x\right) \vee \left(\neg complete_x^\phi \wedge same(O_x)\right)\right] \end{aligned}$$

$\mathbb{D}_x^\phi(A_x)$ relies on the scheduling interval, δ_x^ϕ, for the input assumption to hold. The guarantee on the output hold is more direct relying only on the current value of $complete_x^\phi$.

A scheduled system, $S = (C, \phi, I, O, A, P)$, is correct if and only if for all π and for all $i \geq 0$ the following holds:

$$\begin{aligned} \forall c \in C\ \mathbf{G}\left[\mathbf{H}\left(\mathbb{D}_S^\phi(A) \wedge \textstyle\bigwedge_{c' \in \mathbb{I}_c} \mathbb{C}_{c'}^\phi(P_{c'})\right) \implies \mathbb{D}_c^\phi(A_c)\right] \bigwedge \\ \mathbf{G}\left[\mathbf{H}\left(\mathbb{D}_S^\phi(A) \wedge \textstyle\bigwedge_{c' \in \mathbb{O}} \mathbb{C}_{c'}^\phi(P_{c'})\right) \implies \left(complete_s^\phi \wedge P_s\right)\right] \end{aligned}$$

Here the system itself has a dispatch cycle in the schedule as discussed in the prior section. The first set of verification conditions, one condition in the set for each component, checks compatibility between connected components. Component outputs that are consumed by downstream components as inputs must have

```
thread WaypointPlanManagerService_thr
    features
        AutomationResponse: in event data port CMASI::AutomationResponse.i;
        MissionCommand: out event data port CMASI::MissionCommand.i;
    annex agree {**
        eq Dispatch: bool;
        eq Complete: bool;

        guarantee Sem_WPM_Output_Event_Hold_MissionCommand "Output event freeze" :
            not Complete => (event(MissionCommand) = prev(event(MissionCommand), false));
        guarantee Sem_WPM_Output_Data_Hold_MissionCommand "Output data freeze" :
            not Complete => (true -> (MissionCommand = pre(MissionCommand)));

        assume Req_WPM_Good_Automation_Response "Input valid automation response":
            Dispatch => (event(AutomationResponse) => WELL_FORMED_AUTOMATION_RESPONSE(AutomationResponse));

        guarantee Req_WPM_Good_Mission_Command "Output valid mission commands" :
            Complete => (event(MissionCommand) => WELL_FORMED_MISSION_COMMAND(MissionCommand));
    **};
end WaypointPlanManagerService_thr;
```

Fig. 5. Modeling of Scheduling Semantics in AGREE

guarantees strong enough to satisfy input assumptions at dispatch. These must also respect the input freeze required by the consuming component.

The second condition is for the system outputs. Components producing system outputs must have guarantees strong enough to imply that the system guarantees hold at complete. Unlike components though, there is no output hold requirement for the system because outputs appear depending on when components producing those outputs complete. As before, if all of the verification conditions hold, then a scheduled system is said to be correct. Correct means that for the schedule $\phi\prime$, $\mathbf{G}\left[\mathbf{H}\left(\mathbb{D}_S^{\phi\prime}(A)\right) \implies \mathbb{C}_S^{\phi\prime}(P)\right]$ holds. Here the internal components of the system are completely abstracted away, and the system itself is just some scheduled component in $\phi\prime$ belonging to a larger system.

6 AGREE Model

The scheduling semantics can often be directly modeled in the AADL AGREE annex. At the component level, this requires introducing two Boolean variables *dispatch* and *complete*, augmenting the original *assumptions* and *guarantees* with *dispatch* and *complete*, respectively, and adding additional *guarantees* to enforce the output freeze rule. We often omit the assumptions of frozen inputs, as they are trivially satisfied by the schedule definition, the output freeze rule, and the system-level assumptions.

Figure 5 shows a simplified AADL model originally developed on the DARPA CASE program [19]. The first two *guarantees* are added to freeze the outputs between completions. Also the original contract (the assumption and the third *guarantee*) are augmented with *dispatch* and *complete*. In practice, we find that direct modeling is helpful to clarify the semantics with users. However, in general it could be a complex task, particularly if the contracts depend on past history. In the next section, we will discuss how the Lustre backend model is used to handle the general case.

```
node CircularCounter (init: int, incr: int, reset: bool)
returns (count: int);
let
    count = if reset then init
            else init-> (pre(count) + incr);
tel;

const PERIOD :   int = 10;

eq tick : int = CircularCounter(1, 1, prev(tick = PERIOD, false));

assume "Schedule ABACD" :
    A_Dispatch = (tick = 1 or tick = 5) and
    A_Complete = (tick = 2 or tick = 6) and
    B_Dispatch = (tick = 3) and
    B_Complete = (tick = 4) and
    C_Dispatch = (tick = 7) and
    C_Complete = (tick = 8) and
    D_Dispatch = (tick = 9) and
    D_Complete = (tick = 10);
```

Fig. 6. Modeling Schedule with Circular Counter in AGREE

At the system level, we use a circular counter to model a cyclic schedule in AGREE. The counter updates at every instant. Once it reaches the limit, it resets to one at the next instant. We set the limit to the period of the schedule.

Based on the current count, the counter triggers a corresponding scheduling event. Figure 6 shows an AGREE model of the schedule ($ABACD$).

7 LUSTRE Backend Model

AGREE translates an AADL model and its annotated contracts into a dialect [12] of the Lustre language, and then queries a user-selected model checker to perform the verification. The dialect includes an expression called *condact*, which is similar to the activation condition in SCADE. It clocks a node call expression as follows: $condact(cond, node(node\_inputs, node\_outputs), init\_outputs)$. If the Boolean signal *cond* is true, the clocked node *node* is activated and updates its local and output signals. Otherwise, the node keeps the previous value of the local and output signals. Before the first activation, the node outputs values are set to *init_outputs*. We are aware that the standard Lustre language introduced similar temporal operators like *when* and *current*. We use *condact* simply because it is supported by our default model checker JKind [12].

AGREE translates an AADL thread to a Lustre node in a *constraint* style, in which the thread input and output ports are both mapped to the node input signals. Thus, the *condact* expression does not automatically freeze the thread outputs. We add assertions to enforce the output freeze rule, and we use the thread *complete* signal to clock the node. The *complete* signal is triggered by the circular counter shown in Figure 6. Figure 7 shows an example of using *condact* to model a scheduled AADL thread.

```
assert condact(WPM_Complete, WPM(WPM_ASSUME_HIST, WPM_ASSUME0, WPM_AutomationResponse_EVENT,
            WPM_AutomationResponse, WPM_MissionCommand_EVENT, WPM_MissionCommand), true);

assert (not WPM_Complete) => (true -> (WPM_MissionCommand = pre(WPM_MissionCommand)));
assert (not WPM_Complete) => (WPM_MissionCommand_EVENT = (false -> pre(WPM_MissionCommand_EVENT)));
assert (not WPM_Complete) => (WPM_ASSUME0 = (true -> pre(WPM_ASSUME0)));
```

Fig. 7. Modeling Scheduling Semantics with *condact* in Lustre

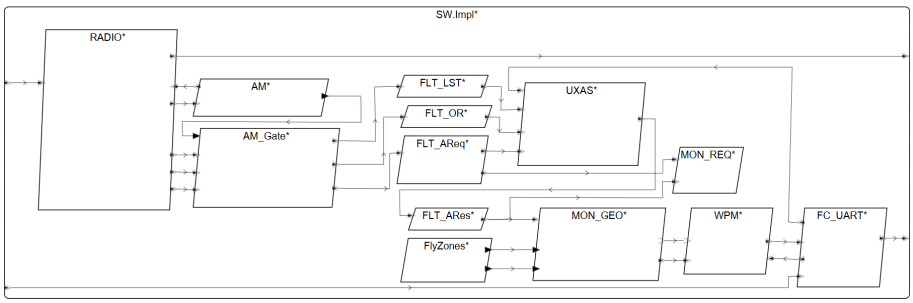

Fig. 8. UAV Software Architecture Model in AADL

8 Case Study

The approach presented in this paper was applied to the BriefCASE toolchain [7], which was developed on the DARPA CASE program to assist engineers in the design of inherently cyber-resilient embedded systems. As part of the demonstration effort, a UAV surveillance system architecture was modeled in AADL. Figure 8 shows the architecture of the UAV mission computer software, which receives commands from a ground station to conduct surveillance along a geographical feature, such as a river. The software generates a flight plan adhering to a set of keep-in and keep-out zones, which is then sent to the UAV flight controller.

The baseline design included the UxAS [24] flight planning component, waypoint plan manager, UART driver, radio driver, and fly-zone database. These components were associated with varying levels of trustworthiness. In particular, UxAS was treated as blackbox software and deemed potentially security-compromised since it was an open-source component developed by a third party. BriefCASE includes tools that analyze architecture models and generate requirements corresponding to vulnerabilities in the design. The BriefCASE cyber-resiliency tool was then used to address the requirements by transforming the model, thereby mitigating the corresponding vulnerabilities. The transformations inserted eight high-assurance components into the model including an attestation manager, attestation gate, two monitors, and four filters. AGREE behavioral specifications for these components were provided, describing their intended functionality.

The hardened model (baseline plus high-assurance components) contained 13 threads, all of which were mapped to a single mission computer processor running

```
fun WellformedCASE_RF_Msg(msg : CASE_RF_Msg.Impl, src : int64.i, dst : int64.i ) : bool =
    WellformedCASE_MsgHeader(msg.header, src, dst);
fun WellformedCASE_MsgHeader(hdr : CASE_MsgHeader.Impl, src : int64.i, dst : int64.i ) : bool =
    (hdr.src = src) and (hdr.dst = dst) and (hdr.trusted = true) and (hdr.HMAC = true);
fun WellformedCASE_UART_Msg(msg : CASE_UART_Msg.Impl) : bool =
    msg.crc = true;
assume "Radio receives well-formed messages" :
    event(radio_recv) => WellformedCASE_RF_Msg(radio_recv, GS_ID, UAV_ID);
assume "UART receives well-formed messages" :
    event(uart_recv) => WellformedCASE_UART_Msg(uart_recv);
guarantee "Radio sends well-formed messsages" :
    event(radio_send) => WellformedCASE_RF_Msg(radio_send, UAV_ID, GS_ID);
guarantee "UART sends well-formed messages" :
    event(uart_send) => WellformedCASE_UART_Msg(uart_send);
```

Fig. 9. Model of Assumptions and Well-formedness Properties in AGREE

the seL4 microkernel (chosen for its formally verified separation guarantees). An seL4 domain schedule was added to the model with all threads designated to run once per scheduling cycle with a period of 500 ms. The processor time allocated to each thread ranged from 2 ms (filters and monitors) to 100 ms (UxAS). The verification goal was to prove that the key system security properties were satisfied by the hardened model with the components executing according to the seL4 domain schedule.

We note that although *event* and *event data* ports were used in the UAV AADL model, they were intended to model the event-triggered execution of periodic threads. In addition, since each thread executed once every scheduling cycle, the number of queued events or data was always equal to or less than one, making this model suitable for the application of our modeling framework.

The following system-level security properties were to be verified in the presence of the seL4 domain schedule: (a) the output UART and RF messages are *well-formed*, (b) the system only responds to trusted sources, and (c) the waypoints generated are *geo-fenced*. The encoding of the well-formedness property and its assumptions is shown in Figure 9.

Our framework was able to prove these properties in less than 2 minutes on a PC with 2.6 GHz CPU and 32 GB RAM. The verification results are shown in Figure 10.

The case study is reflective of a development workflow in which we first verify that the component contracts hold under a synchronous dataflow model. As the design is refined and an execution schedule for each component is specified, we want to show that the system properties continue to hold. Our new framework enables such verification, providing assurance of intended behavior at runtime.

9 Related Work

The AADL standard by itself does not have a well-defined execution semantics. In order to formally verify an AADL model, it is often translated to a formal model, like timed automata [11], Lustre [14], and Real-Time Maude [22]. Then a formal method is applied to analyze the translated model.

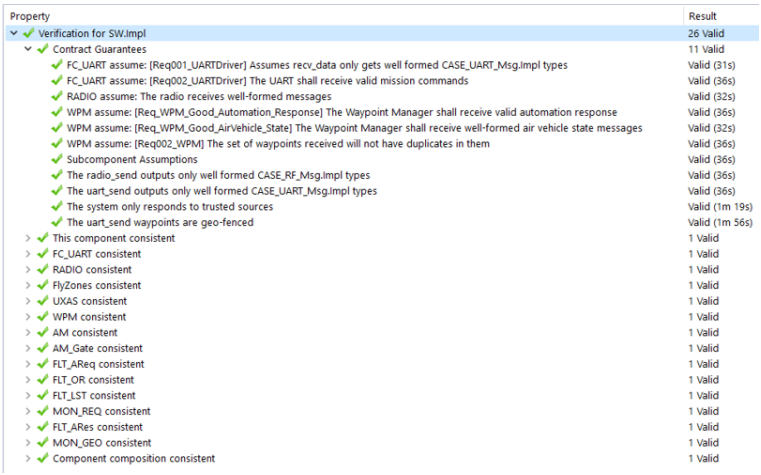

Fig. 10. Use Case Verification Results in AGREE

In *aadl2sync* [14], the AADL behavior models are translated to synchronous programs mainly for simulation. *aadl2sync* uses activation condition to model sporadic execution of software components. By contrast, our proposed framework focuses on simulating detailed timed behavior in the presence of clock drift. Moreover, we focus on the formal verification of system properties based on component requirements, which in general do not completely define component behavior.

Metzler et al. [21] use an iterative and incremental approach to prove safety properties of concurrent programs. Their technique starts with a proof under a specific schedule, and then in each following iteration gradually relaxes the scheduling constraints. The iteration stops when all possible executions are explored or a counterexample is generated. Unlike our component model, their programs are "white boxes", allowing their schedule to interleave instructions between programs. In comparison, our basic scheduling unit is a software thread. In each iteration, the model checking problem is still challenging. In this context, our compositional verification approach makes sense.

10 Conclusion and Future Work

Based on the AGREE framework, we presented an approach to assume-guarantee reasoning with scheduled components. The proposed model of computation differs from the synchronous model used in the current framework. We introduced virtual scheduling events to tie the AADL execution semantics to AGREE contracts. Our approach was applied to the compositional verification of a UAV model developed on the DARPA CASE program.

In the proposed model, the queue associated with an AADL event or event data port is limited to size of one. This limitation is due to our domain of

interest. One interesting future task is to extend the modeling framework to allow a larger queue size. Given a *balanced* schedule, the maximum size of each queue is a constant that can be calculated from the schedule.

ACKNOWLEDGMENT

This work was funded by DARPA contract HR00111890001. The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

We would like to thank John Hatcliff and Robby for many helpful discussions to clarify our understanding of the AADL semantics. We also want to thank David Hardin and anonymous reviewers for their comments that greatly improved the paper.

References

1. Backes, J.D., Whalen, M.W., Gacek, A., Komp, J.: On implementing real-time specification patterns using observers. In: International Symposium on NASA Formal Methods. pp. 19–33. Springer (2016)
2. Baker, T., Shaw, A.: The cyclic executive model and Ada. In: Real-Time Systems Symposium. pp. 120–129. IEEE (1988)
3. Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming **16**(2), 103–149 (1991)
4. Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: CoCoSpec: A mode-aware contract language for reactive systems. In: International Conference on Software Engineering and Formal Methods. pp. 347–366. Springer (2016)
5. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Meyer, B., Nordio, M. (eds.) Tools for Practical Software Verification: LASER 2011, pp. 1–30. Springer (2012)
6. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Fourth Annual Symposium on Logic in Computer Science. pp. 353–362. IEEE (1989)
7. Cofer, D., Amundson, I., Babar, J., Hardin, D., Slind, K., Alexander, P., Hatcliff, J., Robby, R., Klein, G., Lewis, C., Mercer, E., Shackleton, J.: Cyberassured systems engineering at scale. IEEE Security & Privacy (01), 2–14 (March 2022)
8. Cofer, D., Gacek, A., Backes, J., Whalen, M.W., Pike, L., Foltzer, A., Podhradsky, M., Klein, G., Kuz, I., Andronick, J., Heiser, G., Stuart, D.: A formal approach to constructing secure air vehicle software. Computer **51**(11), 14–23 (2018)
9. Colaço, J.L., Pagano, B., Pouzet, M.: SCADE 6: A formal language for embedded critical software development. In: International Symposium on Theoretical Aspects of Software Engineering. pp. 1–11. IEEE (2017)
10. Feiler, P., Gluch, D.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional (2012)
11. Frana, R., Bodeveix, J.p., Filali, M., Rolland, J.F.: The AADL behaviour annex - experiments and roadmap. In: International Conference on Engineering of Complex Computer Systems. pp. 377 – 382. IEEE (2007)

12. Gacek, A., Backes, J., Whalen, M., Wagner, L.G., Ghassabani, E.: The JKind model checker. In: International Conference on Computer Aided Verification. pp. 20–27. Springer (2018)
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE **79**(9), 1305–1320 (1991)
14. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: International Conference on Embedded Software. pp. 134–143. ACM (2007)
15. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing, Proceedings of the 6th IFIP Congress 1974. pp. 471–475. North-Holland (1974)
16. Kamp, J.A.W.: Tense Logic and the Theory of Linear Order. Ph.D. thesis, UCLA (1968)
17. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: ACM Symposium on Operating Systems Principles. pp. 207–220. ACM (2009)
18. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)
19. Mercer, E., Slind, K., Amundson, I., Cofer, D., Babar, J., Hardin, D.: Synthesizing verified components for cyber assured systems engineering. In: 24th International Conference on Model Driven Engineering Languages and Systems. pp. 205–215. IEEE (2021)
20. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theor. Comput. Sci. **451**, 1–37 (2012)
21. Metzler, P., Suri, N., Weissenbacher, G.: Extracting safe thread schedules from incomplete model checking results. International Journal on Software Tools for Technology Transfer **22**(5), 565–581 (2020)
22. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral AADL models in real-time Maude. In: Hatcliff, J., Zucca, E. (eds.) Formal Techniques for Distributed Systems. pp. 47–62. Springer (2010)
23. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems, sub-series F: Computer and System Science. pp. 123–144. Springer-Verlag (1985)
24. Rasmussen, S., Kingston, D., Humphrey, L.: A brief introduction to unmanned systems autonomy services (UxAS). In: International Conference on Unmanned Aircraft Systems. pp. 257–268. IEEE (2018)
25. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and memory requirements analysis with AADL. In: Annual ACM SIGAda International Conference on Ada. pp. 1–10. ACM (2005)
26. Whalen, M.W., Gacek, A., Cofer, D., Murugesan, A., Heimdahl, M.P., Rayadurgam, S.: Your "what" is my "how": Iteration and hierarchy in system design. IEEE Software **30**(2), 54–60 (2013)