An Extension of HybridSynchAADL and Its Application to Collaborating Autonomous UAVs

Jaehun Lee[1], Kyungmin Bae[1], and Peter Csaba Ölveczky[2]

[1] Pohang University of Science and Technology, Korea
[2] University of Oslo, Norway

Abstract. Many collective adaptive systems consist of distributed nodes that communicate with each other and with their physical environments, but that logically should operate in a synchronous way. HybridSynchAADL is a recent modeling language and formal analysis tool for such virtually synchronous cyber-physical systems (CPSs). HybridSynchAADL leverages the Hybrid PALS equivalence to reduce the hard problem of designing and verifying virtually synchronous CPSs—with asynchronous communication, network delays, imprecise local clocks, continuous dynamics, etc.—to the much easier tasks of designing and verifying their underlying synchronous designs. The HybridSynchAADL modeling language is an annotated subset of the industrial modeling standard AADL, and Maude-with-SMT-based formal analysis of HybridSynchAADL models has been integrated into the OSATE tool environment for AADL. Up to now HybridSynchAADL has lacked important programming language features such as compound data types and user-defined functions. This makes it difficult to model advanced control logics of collective adaptive systems. In this paper, we extend the HybridSynchAADL language, its formal semantics, and its analysis tool to support array and record data types and user-defined functions. We apply our extension of HybridSynchAADL to design and analyze a collection of collaborating autonomous drones that adapt to their environments.

1 Introduction

Many distributed cyber-physical systems (CPSs)—including avionics [4, 29] and automotive systems [26, 33], networked medical devices [3, 21], and collaborating drones [25]—are *virtually synchronous*: They should logically behave as if they were synchronous—in each iteration of the system, all components, in lockstep, read inputs and perform transitions which generate outputs for the next iteration—but have to be realized in a distributed setting where the infrastructure guarantees bounds Γ on the clock skews, networks delays, and execution times. The design and model checking of such virtually synchronous CPSs is hard, due to communication delays, race conditions, execution times, and imprecise local clocks, and due to the state space explosion caused by interleavings.

The PALS ("physically asynchronous, logically synchronous") design and analysis pattern [2, 29] greatly simplifies the design and verification of virtually synchronous distributed real-time system *without* continuous behaviors: It is sufficient to design and verify the much simpler synchronous design *SD*—without asynchrony, network delays, clock skews, etc.—since the corresponding distributed real-time system $PALS(SD, \Gamma)$ satisfies the same properties as *SD*.

Many virtually synchronous CPSs are distributed *hybrid* systems where local controllers have continuous environments. *Hybrid PALS* [8] extends PALS to such distributed hybrid systems. In such systems we can no longer abstract from the time when a continuous value is sampled or when a control command is sent, both of which depend on the local controller's imprecise clock.

To make the Hybrid PALS design and verification methodology available to CPS designers, we have developed the HybridSynchAADL modeling language and analysis tool [24, 25]. Its modeling language (for modeling the synchronous designs) is an annotated sublanguage of AADL [18], an industrial modeling standard used in avionics, aerospace, automotive systems, and robotics. We have also integrated the modeling and formal model checking analysis of HybridSynchAADL models into the OSATE modeling environment for AADL.

In HybridSynchAADL, controller behaviors are defined using a subset of AADL's *Behavior Annex* [19], with behaviors defined by transitions with Boolean guards, variable assignments, conditionals, and so on. Mode-dependent continuous behaviors are specified using differential equations. In [24] we use the rewriting logic language Maude [16] to formalize complex discrete control programs, and use Maude combined with SMT solving [11, 31] (in particular, the SMT solver Yices2 [17]) to *symbolically* encode continuous behaviors—with all possible sampling and actuating times depending on imprecise clocks—and provide a Maude-with-SMT semantics, as well as symbolic reachability analysis, randomized simulation, and multithreaded portfolio analysis, for HybridSynchAADL. In [24, 25] we use HybridSynchAADL to model and analyze a collection of autonomous drones, and show that HybridSynchAADL in most cases outperforms the state-of-the-art hybrid systems tools HyComp [15], SpaceEx [20], Flow* [14], and dReach [22] on a simplified version of the case study.

Up to now HybridSynchAADL has lacked some language features that would make it more convenient and less error-prone to model sophisticated CPSs. First of all, we need user-definable data types instead of just basic types. In an adaptive system, a component may communicate with a significant number of neighboring components. We therefore extend HybridSynchAADL with *arrays*, e.g., to conveniently store information about many other nodes. Finally, we extend HybridSynchAADL with the ability to specify user-defined functions as AADL *subprograms*. We also extend the HybridSynchAADL property specification language accordingly (Section 6). In this paper, we introduce these new features of HybridSynchAADL (Section 4), and explain how its Maude-with-SMT semantics has been extended to include these features (Section 5). We demonstrate the modeling and analysis convenience of this new version of our tool with a system of collaborating drones for packet delivery, where each drone adapts to the motions of the other drones for collision avoidance (Section 6). Our tool and the model of the case study are available at `https://hybridsynchaadl.github.io/artifact/isola2022`.

2 Preliminaries

PALS. The PALS pattern [2,29] reduces the problem of designing and verifying a distributed real-time system to the much easier problems of designing and verifying its synchronous design, provided that the underlying infrastructure guarantees bounds Γ on execution times, clock skews, and network delays. For a synchronous design SD, bounds Γ, and a period p, PALS provides the distributed real-time system $PALS(SD, \Gamma, p)$, which is stuttering bisimilar to SD.

The synchronous design SD is formalized as the synchronous composition of an *ensemble* of communicating state machines [29]. At the beginning of each iteration, each state machine performs a transition based on its current state and its inputs, proceeds to the next state, and generates outputs. All machines perform their transitions at the same time, and the outputs to other machines become inputs at the *next* iteration.

Hybrid PALS. Hybrid PALS [8] extends PALS to virtually synchronous CPSs with environments that exhibit continuous behaviors. The *physical environment* E_M of a machine M has real-valued parameters $\vec{x} = (x_1, \ldots, x_l)$. The continuous behaviors of $\vec{x}$ are modeled by ordinary differential equations (ODEs) that specify different *trajectories* on $\vec{x}$. E_M also defines *which* trajectory the environment follows, as a function of the last *control command* received by E_M.

The local clock of a machine M can be seen as a function $c_M : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, where $c_M(t)$ is the value of the local clock at time t, satisfying $|c_M(t) - t| < \epsilon$ for the maximal clock skew $\epsilon > 0$ [29]. In its ith iteration, a controller M samples the values of its environment at time $c_M(i \cdot p) + t_s$, where t_s is the *sampling time*, and then executes a transition. As a result, the new control command is received by the environment at time $c_M(i \cdot p) + t_a$, where t_a is the *actuating time*.

AADL. The *Architecture Analysis & Design Language* (AADL) is an industrial modeling standard used in avionics, automotive, medical devices, and robotics to describe an embedded real-time system [18]. AADL models describe a system of hardware and software components. Software components include: *threads* modeling the application software; *data* representing data types; *subprograms* representing subroutines; and *systems* defining top-level components.

In AADL, a component *type* specifies the component's *interface* (e.g., ports) and *properties* (e.g., periods), and a component *implementation* specifies its internal structure as *subcomponents* and *connections* linking their ports. AADL constructs may have *properties* describing their parameters, declared in *property sets*. Thread and subprogram behavior is modeled as a guarded transition system with local variables using AADL's *Behavior Annex* [19]. When a thread is activated, enabled transitions are applied until a complete state is reached.

Maude with SMT. Maude [16] is a language and tool for formally specifying and analyzing distributed systems in rewriting logic. A *rewrite theory* [28] is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory—specifying system states as an algebraic data type—with Σ a signature (declaring sorts, subsorts, and function symbols) and E a set of equations; and R is a set of rewrite rules—specifying system transitions—of the form $l : \ t \longrightarrow t'$ **if** $cond$, where l is a label, t and t' are terms, and $cond$ is a conjunction of equations and rewrites.

A declaration `class C |` $att_1 : s_1, \dots, att_n : s_n$ declares a class C with attributes $att_1, \dots, att_n$ of sorts $s_1, \dots, s_n$. An *object* o of class C is represented as a term `<` $o : C$ `|` $att_1 : v_1, \dots, att_n : v_n$ `>` of sort `Object`, where v_i is the value of att_i. A `subclass` inherits the attributes and rewrite rules of its superclasses. A *configuration* is a multiset of objects and messages, and has sort `Configuration`, with multiset union denoted by juxtaposition.

In addition to its explicit-state analysis methods for concrete states (ground terms), Maude also provides SMT solving and *symbolic reachability analysis* for *constrained terms*, using connections to Yices2 [17] and CVC4 [13]. A constrained term is a pair $\phi \parallel t$ that symbolically represents all instances of the term t satisfying the SMT constraint ϕ. A *symbolic rewrite* on constrained terms can symbolically represent a (possibly infinite) set of system transitions [11, 31].

3 Overview of HybridSynchAADL

This section gives a brief overview of the HybridSynchAADL language and its formal semantics as defined in [24, 25].

3.1 The HybridSynchAADL Modeling Language

The HybridSynchAADL language is a subset of AADL extended with the property set `Hybrid_SynchAADL`. HybridSynchAADL can specify synchronous designs of distributed controller components, (local) environment components with continuous dynamics, and interactions between controllers and environments based on imprecise local clocks and sampling and actuation times.

Discrete controllers are standard software components in the Synchronous AADL subset of AADL [7, 9]. This subset includes system, process, and thread components; data components for base types; ports and connections; and thread behaviors defined in the Behavior Annex [19]. However, the subset does not include composite data types, subprograms, and arrays of components and ports.

Environments specify real-valued state variables that change continuously over time. The continuous dynamics of state variables can be declared using either ODEs or continuous real functions. An environment can have multiple *modes* for different continuous dynamics. A controller command may change the mode of the environment or the value of a state variable.

3.2 Symbolic Semantics of HybridSynchAADL

This section briefly summarizes the Maude-with-SMT semantics of the original HybridSynchAADL language; see [24] for details.

```
thread CtrlThread
 features
   c: out event port;
   i: in data port Base_Types::Float;
   o: out data port Base_Types::Float;
end CtrlThread;

thread implementation CtrlThread.impl
 subcomponents
   v: data Base_Types::Float;
 annex behavior_specification {**
  states
    s0: initial complete state;  s1: state;
  transitions
    s0 -[on dispatch]-> s1 {v := (i + v) / 2};
    s1 -[v > 25]-> s0 {c!};
    s1 -[otherwise]-> s0 {o := v};    **};
end CtrlThread.impl;
```

```
not $x_b$ or ($y_b$ and $x_v$ > 0)  ||
< ctrlThread : Thread |
  features :
    < c : DataOutPort |
      content : * # false, ... >
    < i  : DataInPort |
      content : 0 # $y_b$, ... >
    < o : DataOutPort |
      content : 0 # true,  ... >,
  subcomponents :
    < v : Data | value : $x_v$ # $x_b$, ... >,
  transitions :
    s0 -[on dispatch]-> s1
    { v := (i + v) / 2 } ;
    s1 -[v > 25]-> s0 { c ! } ;
    s1 -[otherwise]-> s0 { o := v },
  currState : s0,
  completeStates : s0, ... >
```

Fig. 1. A thread component and a constrained term representation.

Representing HybridSynchAADL Models. Each component is represented as an object instance of a subclass of the base class `Component`. The attribute `features` denotes a set of `Port` objects, `subcomponents` denotes a set of `Component` objects. `connections` denotes its connections, and `properties` denotes its properties.

```
class Component | features : Configuration, properties : PropertyAssociation,
                  subcomponents : Configuration, connections : Set{Connection} .
```

The type of each AADL component corresponds to a subclass of `Component`. The class `Thread` has attributes for thread behaviors, such as transitions, states, and local variables. The class `Env` for environments has attributes for continuous dynamics, sampling and actuating times, and mode transitions.

Ports and data components are also modeled as objects. Data components are represented as instances of the class `Data`, where `value` denotes the current value. A data content is represented as a pair e `#` b of an expression e and a Boolean condition b. If b is *false*, then there is no content (i.e., some "don't care" value $\bot$) in the data/port; otherwise, the value of the content is e.

```
class Data | value : DataContent .            subclass Data < Component .
op _#_ : Exp BoolExp -> DataContent [ctor] .
```

We use a *constrained object* of the form ϕ `||` *obj* to symbolically represent a (possibly infinite) set of object instances of *obj*, where $\phi(x_1, \ldots, x_n)$ is an SMT constraint and $obj(x_1, \ldots, x_n)$ is a "pattern" over SMT variables $x_1, \ldots, x_n$. Figure 1 shows an example of a thread component and its representation.

Specifying the Behavior. We define various *semantic operations* on constrained terms to specify the behavior of components, threads, environments, etc. In particular, the operation `executeStep` defines a symbolic rewrite relation for a "big-step" synchronous iteration of a single AADL component.

```
crl executeStep(
    PHI  || < C : Thread | features : PORTS,      variables : VIS,  completeStates : LS,
        transitions : TRS, subcomponents : COMPS, properties : PRS, currState : L >)
 => PHI' || < C : Thread | features : writePort(FM',PORTS'),
                           subcomponents : writeData(DATA',COMPS),  currState : L' >
if {PORTS',FM} := readPort(PORTS)  /\  DATA := readData(COMPS)
/\ execTrans(PHI || {emptyVal(VIS), FM, DATA, PRS}, TRS, L, LS) => L' | FM' | DATA' | PHI' .

crl execTrans(PHI || BCF, TRS, L, LS)
 => if L' in LS then  L' | FM' | DATA' | PHI'
                else  execTrans(PHI' || {local(BCF), FM', DATA', PRS}, TRS, L', LS)  fi
if (L -[GC]-> L' ACT) ; TRS' := TRS  /\  B := guardConst(GC, L, TRS', BCF)
/\ execAction(ACT, (PHI and B) || BCF) => PHI' || {VAL', FM', DATA', PRS} .

crl execAction(ID := EXP, PHI || BCF) => PHI' || update(ID, D, BCF)
if eval(EXP, PHI || BCF) => PHI' || D .

eq update(VI, D, VAL | FM | DATA | PRS) = insert(VI,D,VAL) | FM | DATA | PRS .  --- local
eq update(PI, D, VAL | FM | DATA | PRS) = VAL | insert(PI,D,FM) | DATA | PRS .  --- port
eq update(CI, D, VAL | FM | DATA | PRS) = VAL | FM | insert(CI,D,DATA) | PRS .  --- data
```

Fig. 2. Some semantic operations for thread components.

A symbolic synchronous step of the entire system is then formalized by the following rule `step`. A symbolic rewrite from $\{\phi$ `||` $obj\}$ to $\{\phi'$ `||` $obj'\}$ holds if there is a symbolic rewrite from `executeStep(`ϕ `||` obj`)` to ϕ' `||` obj', provided that obj has no ports and the constraint ϕ' is satisfiable.

```
crl [step]: {PHI || < C : System | features : none >} => {PHI' || OBJ'}
if executeStep(PHI || < C : System | >) => PHI' || OBJ'  /\ check-sat(PHI') .
```

Figure 2 shows the definition of `executeStep` and auxiliary operations for threads. In the first rule, `readPort` returns a map from each input port to its content; `readData` returns a map from each data subcomponent to its value; `execTrans` executes the transition system, given a *behavior configuration* `BCF` of local variables, port contents, data component values, and properties; `writePort` updates the output ports; and `writeData` updates the data subcomponents. In the second rule, a transition `L -[GC]-> L' ACT` from the current state `L` is chosen nondeterministically, and `execAction` executes the actions `ACT` with the guard condition `GC`; if the next state `L'` is a complete state (`L' in LS`), the operation ends; otherwise, `execTrans` is applied again. The operation `execAction` computes a behavior action; e.g., the third rule defines the semantics of an assignment action *id* `:=` *exp*, where `eval` evaluates the data content `D` of an expression.

4 An Extension of HybridSynchAADL

In this section we extend the HybridSynchAADL modeling language in [24,25] with the following AADL constructs: struct and array data types, arrays of components and ports, and subprograms.

```
data Vector
 properties
   Data_Model::Data_Representation => Struct;
   Data_Model::Base_Type => (
     classifier (Base_Types::Float),
     classifier (Base_Types::Float));
   Data_Model::Element_Names => ("x", "y");
end Vector;

data VectorArray
 properties
   Data_Model::Data_Representation => Array;
   Data_Model::Base_Type =>
     (classifier (Vector));
   Data_Model::Dimension => (5);
end VectorArray;

data TwoDimIntArray
 properties
   Data_Model::Data_Representation => Array;
   Data_Model::Base_Type => (
     classifier (Base_Types::Integer));
   Data_Model::Dimension => (10, 20);
end TwoDimIntArray;
```

Fig. 3. Examples of composite data types in AADL.

Composite Data Types. We now support (nested) *struct* and *array* types defined in the Data Modeling Annex [32] of AADL. They are declared as user-defined data components annotated with `Data_Model` properties representing the details of the data types. In particular, the property `Data_Model::Data_Representation` denotes the representation of a data type, such as `Struct` and `Array`.

Figure 3 shows examples of struct and array data types. `Vector` is a struct type with two floating-point elements `x` and `y`, declared using `Data_Model` properties. `VectorArray` is a one-dimensional array of `Vector`, and `TwoDimIntArray` is a two-dimensional array of integers, where the sizes of array dimensions are declared using `Data_Model::Dimension`. In AADL, array indices begin with 1.

Arrays of Components and Ports. In AADL, multiple instances of the same component type can be declared as an array of the component type. For example, the system component `Top.impl` in Fig. 4, contains an array of `Agent` and an array of `Tower`, where both `Agent` and `Tower` have array `ip` of input ports.

The connections between arrays of components/ports are declared with the properties `Connection_Set` and `Connection_Pattern`. A connection set specifies a list of individual connections using array indices. A connection pattern uses a predefined list of frequently used connection sets. For example, the connection `C1` in Fig. 4 uses a connection set: each pair `[src=>(`i, j`); dst=>(`k, l`);]` specifies a connection from output port `ot[`i`]` of subcomponent `agent[`j`]` to input port `ip[`k`]` of subcomponent `tower[`l`]`. The connection `C2` uses a connection pattern: output port `sd` of `agent[`i`]` connects to input port `ip[`j`]` of `agent[`k`]`, where $i = j$ (`One_To_One`) and each j is related to all k's (`One_To_All`).

Subprograms. In AADL, subprograms represent sequentially executable code. Subprogram components can have data parameters to pass and return values. Parameters can be input, output, or both input and output, where input parameters are readable and output parameters are writable. Subprogram components can also have data subcomponents to indicate local temporary variables.

In HybridSynchAADL, subprogram behavior is modeled using guarded transitions written in the Behavior Annex in a way similar to thread behavior. The execution of a subprogram starts in its initial state and ends in a final state.

```
system Agent
 features
   ip: in data port Vector [3];
   sd: out data port Vector;
   ot: out data port Vector [2];
end Agent;

system Tower
 features
   ip: in data port Vector [3];
end Tower;

system Top
end Top;
```

```
system implementation Top.impl
 subcomponents
   agent: system Agent [3];    tower: system Tower [2];
 connections
   C1: port agent.ot -> tower.ip;
   C2: port agent.sd -> agent.ip {Connection_Pattern
         => ((One_To_One, One_To_All));};
 properties
   Connection_Set =>
    ([src=>(1,1);dst=>(1,1);], [src=>(2,1);dst=>(1,2);],
     [src=>(1,2);dst=>(2,1);], [src=>(2,2);dst=>(2,2);],
     [src=>(1,3);dst=>(3,1);], [src=>(2,3);dst=>(3,2);])
   applies to C1;
end Top.impl;
```

Fig. 4. Arrays of subcomponents and features.

```
subprogram getDist
 features
   p: in parameter Vector;
   q: in parameter Vector;
   d: out parameter
        Base_Types::Float;
end getDist;
```

```
subprogram implementation getDist.l1
 annex behavior_specification {**
  states
    s0: initial state;    s1: final state;
  transitions
    s0 -[]-> s1 {d := abs(p.x - q.x) + abs(p.y - q.y)};  **};
end getDist.l1;
```

Fig. 5. A subprogram `getDist`.

A subprogram has one initial state and one or more final states. Subprograms can be called within threads and subprograms (including recursively).

Fig. 5 shows a subprogram `getDist` which computes the distance between two vectors, given by the input parameters `p` and `q`, and returns the value to the caller using the output parameter `d`. The implementation `getDist.l1` returns the rectilinear distance between input parameters `p` and `q` using a single transition from the initial state `s0` to the final state `s1`.

5 Extending the Semantics of HybridSynchAADL

This section presents the Maude-with-SMT semantics for the new features introduced in Sec. 4 by extending the original semantics of HybridSynchAADL. As those features extend the discrete subset of HybridSynchAADL, we have only changed the part for discrete controllers in the original semantics. In particular, the definition of `execAction` is significantly changed to support subprograms and assignment actions with nested struct/array targets.

5.1 Representation of the Additional Features

An array content is represented as a term $array(1 \mapsto d_1; 2 \mapsto d_2; \ldots; n \mapsto d_n)$ of sort `ArrayContent`, where the i-th element is data content d_i. Likewise, a struct content is represented as $struct(c_1 \mapsto d_1; c_2 \mapsto d_2; \ldots; c_n \mapsto d_n)$ of sort

```
< TopInstance : System |
  features : none,
  subcomponents :
    < agent[1] : System | ... >
    < agent[2] : System | ... >
    < agent[3] : System | ... >
    < tower[1] : System | ... >
    < tower[2] : System | features :
        < ip[1] : DataInPort | content :
            struct(x |==> 0 # true ;
                   y |==> 0 # true), ... >
        < ip[2] : DataInPort | ... >
        < ip[3] : DataInPort | ... >, ... >,
```

```
  connections :
    agent[1] .. ot[1] --> tower[1] .. ip[1];
    agent[2] .. ot[1] --> tower[1] .. ip[2];
    agent[3] .. ot[1] --> tower[1] .. ip[3];
    agent[1] .. ot[2] --> tower[2] .. ip[1];
    agent[2] .. ot[2] --> tower[2] .. ip[2];
    agent[3] .. ot[2] --> tower[2] .. ip[3];
    agent[1] .. sd --> agent[1]. ip[1];
    agent[1] .. sd --> agent[2]. ip[1];
    agent[1] .. sd --> agent[3]. ip[1];
    ...
    agent[3] .. sd --> agent[3]. ip[3];
  properties : none >
```

Fig. 6. A Maude representation of `Top`.

`StructContent`, where the element c_i is d_i. Array and struct contents can be nested, since `ArrayContent` and `StructContent` are subsorts of `DataContent`.

Arrays and array connections of components and ports are *fully instantiated* in our representation. Figure 6 shows an example of a Maude representation of `Top.impl` in Fig. 4. Component arrays `agent` and `tower`, and port array `ip` are instantiated as concrete objects. Array connections, declared using a connection set and a connection pattern, are also instantiated as concrete connections.

Subprograms are represented as instances of the class `Subprogram`, similar to `Thread`. The attribute `args` denotes the list of parameters; `transitions` denotes the set of transitions; `currState` denotes the current state; `finalStates` denotes the final states; and `variables` denotes the local variables and their types.

```
class Subprogram | args : List{FeatureId},  transitions : Set{Transition},
                   currState : Location,    finalStates : Set{Location},
                   variables : Map{VarId,DataType},
subclass Subprogram < Component .
```

A parameter of a subprogram is represented as an instance of a subclass of the class `Param`. The `features` attribute of the class `Subprogram` includes a set of `Param` objects instead of `Port` objects. Notice that `features` includes an unordered set of parameters, and `args` defines the argument order of them.

```
class Param | type : DataType,  properties : PropertyAssociation .
class InParam .    class OutParam .    class InOutParam .
subclass InOutParam < InParam OutParam < Param .
```

We define the function `subprogram` that returns a subprogram instance from its fully qualified name (automatically synthesized by code generation).

5.2 Semantic of Composite Data Types

We extend the definitions of the two operations `eval`—evaluating expressions—and `executeAction`—executing actions—for struct and array data types. For `eval`, we define the cases for struct expressions *exp.id* and array expressions

$exp'[exp]$. For `executeAction`, we define the case of an assignment action that includes (nested) struct and array expressions on the left-hand side.

The following rule defines the case of struct expressions $exp.id$ for `eval`. Given a constrained behavior configuration `PHI || BCF` (including local variables, port contents, data component values, and properties), we first evaluate the struct data content of exp and then choose the element id from the content:

```
crl eval(EXP . CI, PHI || BCF) => PHI’ || D
if eval(EXP, PHI || BCF) => PHI’ || struct(CI |==> D ; STR) .
```

Similarly, for array expressions $exp'[exp]$, we first evaluate the index data content e `#` b of exp and the array data content of exp'. Because e may be a symbolic expression (not an integer constant), we nondeterministically choose the i-th element from the array data content with the constraint $i = e$.

```
crl eval(EXP’[EXP], PHI || BCF) => (PHI’’ and B and I === E) || D
if eval(EXP,  PHI || BCF)  => PHI’ || E # B
/\ eval(EXP’, PHI’ || BCF) => PHI’’ || array(I |==> D ; ARR) .
```

Consider an assignment action $a.x[1].y := e$ with a nested struct/array target. The intuitive behavior is as follows. We first evaluate the "top" data content of a, e.g., $td = struct(x \mapsto array(1 \mapsto struct(y \mapsto \ldots ; \ldots); \ldots); \ldots)$. We then update the sub-content of td at the "position" indicated by "$.x[1].y$" with e.

The following rules specify the above behavior. The function `evalPos` returns the top identifier and a position; e.g., `evalPos`$(a.x[1].y, nil)$ returns the pair $\{a, (.x)[1](.y)\}$. The substitution operation $td[pos \leftarrow d](\phi \parallel bcf)$ computes a new data content obtained by replacing the content of position pos with d.

```
crl execAction(TARGET := EXP, PHI || BCF) => assign(TARGET, D, PHI’ || BCF)
if eval(EXP, PHI || BCF) => PHI’ || D .

crl assign(TARGET, D, PHI || BCF) => PHI’ || update(ID, TD’, BCF)
if {ID, POS} := evalPos(TARGET, nil) /\ eval(ID, PHI || BCF) => PHI’’ || TD
/\ TD [POS <- D] (PHI’’ || BCF) => PHI’ || TD’ .
```

5.3 Semantics of Subprogram Calls

We define `executeAction` for subprogram calls $f!(exp_1, \ldots, exp_n)$ as follows. We first obtain the subprogram instance for f, and evaluate the parameters based on the caller's behavior configuration. We then use `execTrans` to execute the subprogram's transition system with a new behavior configuration. Finally, we update the caller's behavior configuration based on the output parameters.

```
crl executeAction(F!(EXPS), PHI || BCF) => retOutParams(OM, FM’, PHI’ || BCF)
if < O : Subprogram | features : PARAMS, properties : PRS,  args : PIS,
                      transitions : TRS, finalStates : LS,
                      variables : VIS,   currState : L > := subprogram(F)
/\ {OM, FM} := outParams(EXPS, PIS, PARAMS, none, empty)
/\ {FM'', PHI''} := inParams(EXPS, PIS, PARAMS, BCF, {FM, PHI})
/\ execTrans(PHI'' || {emptyVal(VIS), FM'', empty, PRS}, TRS, L, LS)
   => L' | FM' | empty | PHI' .
```

The operation outParams returns the output targets OM in the argument list. After the call ends, these targets are updated with the values assigned to the output parameters during subprogram execution. The operation outParams also returns initial contents FM (with no value bot) for the output parameters.

```
eq outParams((EXP,EXPS), PI PIS, < PI : OutParam | type : TY > PARAMS, OM, FM)
 = outParams(EXPS, PIS, PARAMS, insert(PI,EXP,OM), insert(PI,bot(TY),FM)) .
eq outParams(EXPS, PIS, PARAMS, OM, FM) = {OM, FM} [owise] .
```

The operation inParams evaluates the values of the input expressions in the argument list using eval. Notice that the initial contents generated by outParams are updated with the evaluated values for input-output parameters.

```
eq inParams((EXP, EXPS), PI PIS, < PI : InParam | > PARAMS, BCF, {FM, PHI})
 = inParams(EXPS, PIS, PARAMS, BCF, evalInParam(EXP, BCF, {FM, PHI})) .
eq inParams(EXPS, PIS, PARAMS, BCF, {FM, PHI}) = {FM, PHI} [owise] .
crl evalInParam(EXP, BCF, {FM, PHI}) => {insert(PI,D,FM), PHI'}
if eval(EXP, PHI || BCF) => PHI' || D .
```

Finally, the operation retOutParams updates the output parameter targets (generated by outParams) with the values assigned to the output parameters. If no value is assigned to an output parameter, the target is not updated.

```
ceq retOutParams((PI |-> TARGET, OM), FM, PHI || BCF) = retOutParams(OM, FM,
      assign(TARGET, D, PHI || BCF))  if D := data(FM[PI]) /\ hasValue(D) .
eq retOutParams(empty, FM, PHI || BCF) = PHI || BCF .
```

6 Case Study: A Packet Delivery System

This section shows how HybridSynchAADL can be used to design and analyze a collection of collaborating autonomous drones, taking into account network delays, clock skews, execution times, continuous dynamics, etc. The new features supported by HybridSynchAADL make it easy to specify and analyze multiple instances of components with complex control programs.

6.1 System Description

We consider a packet delivery system adapted from [35]. As illustrated in Fig. 7, there are drones, packets, and charging stations. A drone picks up a packet and transports it to its destination. Drones use energy when moving and can recharge at charging stations. Each drone exchanges its position with other drones to adapt its movement to the motions of the other drones.

The continuous dynamics of the i-th drone is specified by the ODEs $\dot{\vec{x}}_i = \vec{v}_i$ and $\dot{e}_i = -h \cdot |\vec{v}_i|$, where $\vec{x}_i$, $\vec{v}_i$, and e_i denote its position, velocity, and energy,

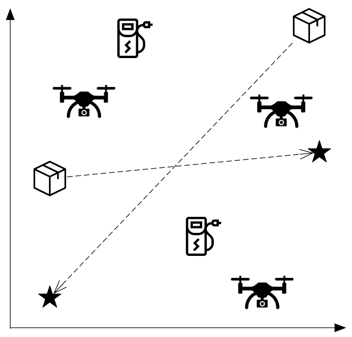

Fig. 7. A packet delivery system.

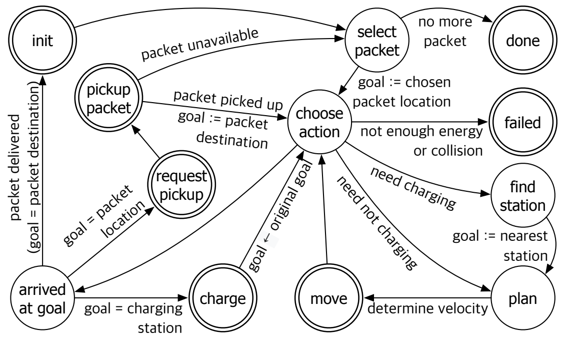

Fig. 8. The control logic of drones.

respectively, and h denotes the energy consumption rate. The controller samples the drone's position, velocity, and energy at its sampling time, and gives a new velocity value to the environment at its actuating time.

Figure 8 illustrates the control logic of drones, where double circles indicate complete states, and `init` denotes the initial state. The controller uses a state variable `goal` to indicate the drone's target, such as a packet location, a packet destination, or a charging station location. The drone's behavior is determined in state `choose_action`, based on the current values of state variables (including `goal`) and the sampled position and energy from the environment.

A new velocity is calculated in state `plan` to move towards the current `goal` while adapting to the motions of the other drones. In this paper, we use this adaptation framework to implement a simple collision avoidance technique: each drone has a priority, and when a potential collision is detected (e.g., the distance between two drones is below a certain threshold), a drone with a lower priority must yield to a drone with a higher priority.

6.2 The HybridSynchAADL Model

Figure 9 shows the top-level system component that contains `Drone` and `Packet` component arrays. We model the locations of charging stations as a constant array. The period, maximal clock skew, and sampling and actuating times are declared using `Hybrid_SynchAADL` properties. A drone can send a request to a packet (connection `C1`) and its position to the other drones (connection `C3`), and a packet can reply its destination to a drone (connection `C2`). In this section, we consider a packet delivery system with three drones and two packets.

A `Packet` component chooses one of the drones that have sent the request, and sends its destination to the selected drone. A `Drone` contains a controller and an environment connected to each other using ports. The environment declares the continuous dynamics of the drone's position, velocity, and energy mentioned above. We assume that a drone moves in a two-dimensional space. The controller also communicates with the outside components using `Drone`'s ports.

```
system PacketDelivery end PacketDelivery;

system implementation PacketDelivery.impl
 subcomponents
  drone: system Drone[3];
  packet: system Packet[2];
 connections
  C1: port drone.req -> packet.req;
  C2: port packet.dest -> drone.dest;
  C3: port drone.oPos -> drone.iPos
    {Connection_Pattern =>
      ((One_To_One, One_To_All));};
 properties
  Hybrid_SynchAADL::Synchronous => true;
  Period => 100ms;
  Timing => Delayed applies to C1,C2,C3;
```

```
  Hybrid_SynchAADL::Max_Clock_Deviation => 5ms;
  Hybrid_SynchAADL::Sampling_Time => 20ms..25ms;
  Hybrid_SynchAADL::Response_Time => 40ms..45ms;
  Connection_Set => ([src=>(1,1); dst=>(1,1);],
    [src=>(1,2); dst=>(2,1);],
    [src=>(1,3); dst=>(3,1);],
    [src=>(2,1); dst=>(1,2);],
    [src=>(2,2); dst=>(2,2);],
    [src=>(2,3); dst=>(3,2);]) applies to C1;
  Connection_Set => ([src=>(1,1); dst=>(1,1);],
    [src=>(2,1); dst=>(1,2);],
    [src=>(3,1); dst=>(1,3);],
    [src=>(1,2); dst=>(2,1);],
    [src=>(2,2); dst=>(2,2);],
    [src=>(3,2); dst=>(2,3);]) applies to C2;
end PacketDelivery.impl;
```

Fig. 9. A top level component in HybridSynchAADL.

Figure 10 shows part of the HybridSynchAADL specification for a controller thread that implements the control logic of Fig. 8. It contains several state variables, such as `goal` for the current target and `chargeStation` for the charging station locations. It also uses several subprograms, such as `chkClose` to check whether the drone is too close to other drones with higher priority. The entire HybridSynchAADL specification of our model is given in the appendix and is available at `https://hybridsynchaadl.github.io/artifact/isola2022`.

6.3 Formal Analysis

We are interested in analyzing whether all drones complete their tasks (i.e., going to state `done`) within a certain time, e.g., 10 seconds. This can be expressed as the following invariant property using HybridSynchAADL's property specification language. We analyze this property up to bound 10,100 ms.

```
invariant [complete] :
  ?initial ==> (not clock.time >= 10000) or ?allDone in time 10100 ms;
```

The above propositions `allDone` and `initial` are declared as follows. The declaration of `allDone` includes universal quantification over array index `i`, which is a new feature of HybridSynchAADL proposed in this work. Notice that there are infinitely many initial states satisfying `initial`.

```
proposition [allDone]: forall i in {1,2,3}. drone[i].ctrl.proc.thrd @ done;
proposition [initial]:
  abs(drone[1].env.x - 2.3) < 0.3 and abs(drone[1].env.y - 0.7) < 0.3 and
  abs(drone[2].env.x - 0.7) < 0.3 and abs(drone[2].env.y - 2.3) < 0.3 and
  abs(drone[3].env.x - 12.0) < 0.3 and abs(drone[3].env.y - 12.0) < 0.3;
```

We find a counterexample by randomized simulation in one minute. The counterexample shows that a collision occurs after five iterations (500 ms), be-

```
thread DroneThread
 features
   req: out data port Base_Types::Boolean[];
   dest: in data port Vector[];
   oPos: out data port Vector;
   iPos: in data port Vector[];
   ...
end DroneThread;

thread implementation DroneThread.impl
 subcomponents
   goal: data Vector;
   chargeStation: data VectorArray;  ...
 annex behavior_specification {**
  variables
    cur: Vector;  ...
  states
    init: initial complete state;  ...
  transitions
    ...
    plan -[]-> move {
      chkClose.impl! (cur, iPos,..., close);
      if (close) set_hover! elsif  ...
    };
    ...
 **};
end DroneThread.impl;
```

```
subprogram chkClose
 features
   p: in parameter Vector;
   pos: in parameter VectorArray;
   cand: in parameter BooleanArray;
   size: in parameter Base_Types::Integer;
   thld: in parameter Base_Types::Float;
   output: out parameter Base_Types::Boolean;
end detectCollision;

subprogram implementation chkClose.impl
 annex behavior_specification {**
  variables
    d: Base_Types::Float;
    i: Base_Types::Integer;
    r : Base_Types::Boolean;
  states
    s0: initial state;  sf: final state;
    sl: state;
  transitions
    s0 -[]-> sl { r := false; i := 1 };
    sl -[r = false and i <= size]-> sl {
      getDist.l1! (p, pos[i], d);
      r := d < thld and cand[i]; i := i + 1};
    sl -[otherwise]-> sf { output := r };
 **};
end detectCollision.impl;
```

Fig. 10. A controller thread in HybridSynchAADL ('...' indicates omitted parts).

cause the subprogram chkClose does not consider clock skews and sampling/actuating times. Each drone's position is sampled from the environment at some time in the sampling time interval, also perturbed by a clock skew. The calculation of chkClose is not precise enough without considering these values.

We therefore modify the implementation DroneThread.impl to mitigate this problem as follows. When invoking chkClose, we use an extra padding value depending on the maximal clock skew and sampling/actuating time intervals. With this change, no counterexample of success is found for 3 hours using randomized simulation. Furthermore, we verify that no such counterexample exists up to bound 500 ms using symbolic reachability analysis for the following invariant property safety, which takes about 126 minutes.

```
invariant [safety]: ?initial ==> not ?failure in time 500 ms;
proposition [failure]: exists i in {1,2,3}. drone[i].ctrl.proc.thrd @ fail;
```

7 Related Work

PALS is a synchronizer for CPSs without continuous behaviors, and is therefore related to time-triggered architectures (TTA) [23], but typically allows shorter periods, etc. See [6,34] for comparisons between PALS and TTA. MSYNC [6] generalizes both TTA and PALS (and its multirate extension Multirate PALS [5]). Unlike Hybrid PALS, neither of these take continuous behaviors into account.

Synchronous AADL [7, 10] and Multirate Synchronous AADL [9] also use AADL to define synchronous PALS designs, but do not consider continuous behaviors. As mentioned above, this work extends HybridSynchAADL in [24, 25] with features making it easy to specify complex systems, and demonstrate the extended version of the language and analysis tool on a new case study.

Unlike other hybrid extensions of AADL, e.g., [1, 12, 27, 30], HybridSynch-AADL supports the specification of complex controllers using (a subset of) AADL's expressive Behavior Annex, and we also consider (virtually synchronous) CPSs—with clock skews, network delays, etc. (using the Hybrid PALS equivalence). See [24, Section 10] for a more detailed discussion of related work.

8 Concluding Remarks

HybridSynchAADL is an AADL-based modeling language and formal analysis tool for sophisticated virtually synchronous (distributed) CPSs—with complex controllers, imprecise local clocks, and continuous behaviors—that is fully integrated into the OSATE tool environment for AADL. Control programs are defined using (a significant subset of) AADL's intuitive and expressive Behavior Annex, and continuous behaviors are given by differential equations. Furthermore, the performance of our tool compares favorably with (less expressive) state-of-the-art hybrid systems analysis tools.

In this paper we have extended HybridSynchAADL with (AADL) features for data types, arrays, and user-defined functions/subprograms. This should make the modeling of complex CPSs—including adaptive CPSs—significantly more convenient and less error-prone. We have introduced the language extensions (including to the property specification language), have extended the Maude-with-SMT formal semantics of HybridSynchAADL with the new features, and have illustrated the convenience of the extended language by modeling and analyzing a complex collection of packet-delivery drones that adapt to the movements of other drones to avoid collision.

References

1. Ahmad, E., Larson, B.R., Barrett, S.C., Zhan, N., Dong, Y.: Hybrid Annex: an AADL extension for continuous behavior and cyber-physical interaction modeling. In: Proc. ACM SIGAda annual conference on High integrity language technology (HILT'14). ACM, New York, NY, USA (2014)
2. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. RTSS. pp. 161–170. IEEE, USA (2009)
3. Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O.: Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. In: HCMDSS-MDPnP. pp. 23–33. IEEE (2007)

4. Bae, K., Krisiloff, J., Meseguer, J., Ölveczky, P.C.: Designing and verifying distributed cyber-physical systems using Multirate PALS: An airplane turning control system case study. Science of Computer Programming **103**, 13–50 (2015)
5. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming **91**, 3–44 (2014)
6. Bae, K., Ölveczky, P.C.: MSYNC: A generalized formal design pattern for virtually synchronous multirate cyber-physical systems. ACM Trans. Embedd. Comput. Syst. (Proc. EMSOFT'21) **20**(5s,Article 105) (2021)
7. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM'11. LNCS, vol. 6991. Springer, Berlin, Heidelberg (2011)
8. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC. pp. 145–154. ACM, New York, NY, USA (2016)
9. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer (2014)
10. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer, Berlin, Heidelberg (2012)
11. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (2019)
12. Bao, Y., Chen, M., Zhu, Q., Wei, T., Mallet, F., Zhou, T.: Quantitative performance evaluation of uncertainty-aware Hybrid AADL designs using statistical model checking. IEEE Transactions on CAD of Integrated Circuits and Systems **36**(12), 1989–2002 (2017)
13. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177. Springer (2011)
14. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. CAV. pp. 258–263. Springer (2013)
15. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. LNCS, vol. 9035. Springer, Berlin, Heidelberg (2015)
16. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer, Berlin, Heidelberg (2007)
17. Dutertre, B.: Yices 2.2. In: Proc. CAV. LNCS, vol. 8559, pp. 737–744. Springer, Berlin, Heidelberg (July 2014)
18. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley, USA (2012)
19. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL Behaviour Annex - experiments and roadmap. In: Proc. ICECCS'07. IEEE, USA (2007)
20. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. CAV. LNCS, vol. 6806. Springer, Berlin, Heidelberg (2011)
21. Kim, C., Sun, M., Mohan, S., Yun, H., Sha, L., Abdelzaher, T.F.: A framework for the safe interoperability of medical devices in the presence of network failures. In: ICCPS. pp. 149–158 (2010)
22. Kong, S., Gao, S., Chen, W., Clarke, E.M.: dReach: δ-reachability analysis for hybrid systems. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7898, pp. 200–205. Springer, Berlin, Heidelberg (2015)

23. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE **91**(1), 112–126 (2003)
24. Lee, J., Bae, K., Ölveczky, P.C., Kim, S., Kang, M.: Modeling and formal analysis of virtually synchronous cyber-physical systems in AADL. Software Tools for Technology Transfer (2022), to appear (preliminary version available at `https://hybridsynchaadl.github.io/artifact/isola2022/sttt-paper.pdf`)
25. Lee, J., Kim, S., Bae, K., Ölveczky, P.C.: HybridSynchAADL: Modeling and formal analysis of virtually synchronous CPSs in AADL. In: Proc. CAV'21. LNCS, vol. 12759, pp. 491–504. Springer, Berlin, Heidelberg (2021)
26. Leen, G., Heffernan, D., Dunne, A.: Digital networks in the automotive vehicle. Computing & Control Engineering Journal **10**(6), 257–266 (1999)
27. Liu, J., Li, T., Ding, Z., Qian, Y., Sun, H., He, J.: AADL+: a simulation-based methodology for cyber-physical systems. Frontiers Comput. Sci. **13**(3), 516–538 (2019)
28. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
29. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theoretical Computer Science **451**, 1–37 (2012)
30. Qian, Y., Liu, J., Chen, X.: Hybrid AADL: a sublanguage extension to AADL. In: Proc. Internetware'13. ACM, New York, NY, USA (2013)
31. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. Journal of Logical and Algebraic Methods in Programming **86**(1), 269–297 (2017)
32. SAE International: Architecture analysis and design language (AADL) annex volume 2: Annex B: Data modeling annex (2011)
33. Steiner, W., Bauer, G., Hall, B., Paulitsch, M., Varadarajan, S.: TTEthernet dataflow concept. In: 2009 Eighth IEEE International Symposium on Network Computing and Applications. pp. 319–322. IEEE (2009)
34. Steiner, W., Rushby, J.: TTA and PALS: Formally verified design patterns for distributed cyber-physical systems. In: 2011 IEEE/AIAA 30th Digital Avionics Systems Conference. pp. 7B5–1. IEEE (2011)
35. Talcott, C., Arbab, F., Yadav, M.: Soft agents: Exploring soft constraints to model robust adaptive distributed cyber-physical agent systems. In: Software, Services, and Systems, pp. 273–290. Springer (2015)

A The Entire HybridSynchAADL Specification

```
package DataTypes
public
  with Base_Types, Data_Model;

  data Vector
    properties
      Data_Model::Data_Representation => Struct;
      Data_Model::Base_Type => (
        classifier (Base_Types::Float),
        classifier (Base_Types::Float));
      Data_Model::Element_Names => ("x", "y");
  end Vector;

  data VectorArray
    properties
      Data_Model::Data_Representation => Array;
      Data_Model::Base_Type => (classifier (Vector));
  end VectorArray;

  data BooleanArray
    properties
      Data_Model::Data_Representation => Array;
      Data_Model::Base_Type => (classifier (Base_Types::Boolean));
  end BooleanArray;

  subprogram getDist
    features
      p: in parameter Vector;
      q: in parameter Vector;
      d: out parameter Base_Types::Float;
  end getDist;

  subprogram implementation getDist.l1
    annex behavior_specification {**
      states
        s0 : initial state;
        s1 : final state;
      transitions
        s0 -[]-> s1 { d := abs(p.x - q.x) + abs(p.y - q.y) };
      **};
  end getDist.l1;
end DataTypes;
```

```
package PacketDeliverySystem
public
  with Drone, Packet, Clock
  with Hybrid_SynchAADL;

  system PacketDelivery
  end PacketDelivery;

  system implementation PacketDelivery.abst
    subcomponents
      drone: system Drone::Drone[];
      packet: system Packet::Packet[];
      clock: system Clock::Clock.impl;
    connections
      C1: port drone.req -> packet.req;
      C2: port packet.dest -> drone.dest;
      C3: port drone.oPos -> drone.iPos
          {Connection_Pattern => ((One_To_One, One_To_All));};
    properties
      Hybrid_SynchAADL::Synchronous => true;
      Period => 100ms;
      Timing => Delayed applies to C1, C2, C3;
      Hybrid_SynchAADL::Max_Clock_Deviation => 5ms;
      Hybrid_SynchAADL::Sampling_Time => 20ms .. 25ms;
      Hybrid_SynchAADL::Response_Time => 40ms .. 45ms;
  end PacketDelivery.abst;
end PacketDeliverySystem;
```

```
package Clock
public
  with Base_Types, Data_Model,
  with Hybrid_SynchAADL;

  system Clock
    properties
      Hybrid_SynchAADL::isEnvironment => true;
  end Clock;

  --- only used for measuring the global time
  system implementation Clock.impl
    subcomponents
      time: data Base_Types::Float {Data_Model::Initial_Value => ("0");};
    properties
      Hybrid_SynchAADL::ContinuousDynamics => "d/dt(time) = 1;";
  end Clock.impl;
end Clock;
```

```
package Drone
public
  with Base_Types, DataTypes, DroneControl, DroneEnvironment;

  system Drone
    features
      --- ports for Packet
      req: out data port Base_Types::Boolean[];
      dest: in data port DataTypes::Vector[];

      --- ports for other drones
      oPos: out data port DataTypes::Vector;
      iPos: in data port DataTypes::Vector[];
  end Drone;
  system implementation Drone.impl
    subcomponents
      ctrl: system DroneControl::DroneControl;
      env: system DroneEnvironment::DroneEnvironment.impl;
    connections
      --- connections between controller and drone's environment
      C1: port ctrl.set_charge -> env.charge;
      C2: port ctrl.northS -> env.moveNS;
      C3: port ctrl.southS -> env.moveSS;
      C4: port ctrl.westS -> env.moveWS;
      C5: port ctrl.eastS -> env.moveES;
      C6: port ctrl.northF -> env.moveNF;
      C7: port ctrl.southF -> env.moveSF;
      C8: port ctrl.westF -> env.moveWF;
      C9: port ctrl.eastF -> env.moveEF;

      C10: port ctrl.set_hover -> env.hover;
      C11: port ctrl.turn_off -> env.off;

      C12: port env.oX -> ctrl.iX;
      C13: port env.oY -> ctrl.iY;
      C14: port env.oE -> ctrl.iE;

      --- connections between ctrl and its enclosing component
      C15: port ctrl.req -> req;
      C16: port dest -> ctrl.dest;
      C17: port ctrl.oPos -> oPos;
      C18: port iPos -> ctrl.iPos;
  end Drone.impl;
end Drone;
```

```
package Packet
public
  with Base_Types, DataTypes;

  system Packet
    features
      req: in data port Base_Types::Boolean[];
      dest: out data port DataTypes::Vector[];
  end Packet;

  system implementation Packet.impl
    subcomponents
      packetProc: process PacketProc.impl;
    connections
      C1: port req -> packetProc.req;
      C2: port packetProc.dest -> dest;
  end Packet.impl;

  process PacketProc
    features
      req: in data port Base_Types::Boolean[];
      dest: out data port DataTypes::Vector[];
  end PacketProc;
  process implementation PacketProc.impl
    subcomponents
      packetThread: thread PacketThread.impl;
    connections
      C1: port req -> packetThread.req;
      C2: port packetThread.dest -> dest;
  end PacketProc.impl;

  thread PacketThread
    features
      req: in data port Base_Types::Boolean[];
      dest: out data port DataTypes::Vector[];
  end PacketThread;

  thread implementation PacketThread.impl
    subcomponents
      destination: data DataTypes::Vector;
    annex behavior_specification {**
      variables
        droneId: Base_Types::Integer;
      states
        idle: initial complete state;   done: final state;  check: state;
      transitions
        idle -[on dispatch]-> check {
          droneId := 0;
          for(forIdx: Base_Types::Integer in 1 .. #Spec::droneNum) {
            if (req[forIdx]) droneId := forIdx end if
          }
        };
        check -[droneId <= 0]-> idle;
        check -[droneId > 0]-> done { dest[droneId] := destination };
    **};
  end PacketThread.impl;
end Packet;
```

```
package DroneEnvironment
public
  with Base_Types;
  with Hybrid_SynchAADL;

  system DroneEnvironment
    features
      oX: out data port Base_Types::Float;
      oY: out data port Base_Types::Float;
      oE: out data port Base_Types::Float;
      moveNF: in event port;   moveNS: in event port;
      moveSF: in event port;   moveSS: in event port;
      moveEF: in event port;   moveES: in event port;
      moveWF: in event port;   moveWS: in event port;
      hover: in event port;  charge: in event port;  off: in event port;
    properties
      Hybrid_SynchAADL::isEnvironment => true;
  end DroneEnvironment;
  system implementation DroneEnvironment.impl
    subcomponents
      x: data Base_Types::Float;
      y: data Base_Types::Float;
      e: data Base_Types::Float;
    connections
      C1: port x -> oX;
      C2: port y -> oY;
      C3: port e -> oE;
    modes
      hovering: initial mode;  charging: mode;  landed: mode;
      northF: mode;    northS: mode;  southF: mode;  southS: mode;
      westF: mode;      westS: mode;  eastF: mode;  eastS: mode;

      northF -[moveNF]-> northF;  northS -[moveNF]-> northF;
      northF -[moveSF]-> southF;  northS -[moveSF]-> southF;
      northF -[moveWF]-> westF;    northS -[moveWF]-> westF;
      northF -[moveEF]-> eastF;    northS -[moveEF]-> eastF;
      northF -[moveNS]-> northS;  northS -[moveNS]-> northS;
      northF -[moveSS]-> southS;  northS -[moveSS]-> southS;
      northF -[moveWS]-> westS;    northS -[moveWS]-> westS;
      northF -[moveES]-> eastS;    northS -[moveES]-> eastS;
      northF -[hover]-> hovering;  northS -[hover]-> hovering;
      northF -[charge]-> charging;  northS -[charge]-> charging;
      northF -[off]-> landed;    northS -[off]-> landed;
;
      southF -[moveNF]-> northF;  southS -[moveNF]-> northF;
      southF -[moveSF]-> southF;  southS -[moveSF]-> southF;
      southF -[moveWF]-> westF;    southS -[moveWF]-> westF;
      southF -[moveEF]-> eastF;    southS -[moveEF]-> eastF;
      southF -[moveNS]-> northS;  southS -[moveNS]-> northS;
      southF -[moveSS]-> southS;  southS -[moveSS]-> southS;
      southF -[moveWS]-> westS;    southS -[moveWS]-> westS;
      southF -[moveES]-> eastS;    southS -[moveES]-> eastS;
      southF -[hover]-> hovering;  southS -[hover]-> hovering;
      southF -[charge]-> charging;  southS -[charge]-> charging;
      southF -[off]-> landed;    southS -[off]-> landed;

      eastF -[moveNF]-> northF;    eastS -[moveNF]-> northF;
      eastF -[moveSF]-> southF;    eastS -[moveSF]-> southF;
      eastF -[moveWF]-> westF;    eastS -[moveWF]-> westF;
      eastF -[moveEF]-> eastF;    eastS -[moveEF]-> eastF;
      eastF -[moveNS]-> northS;    eastS -[moveNS]-> northS;
      eastF -[moveSS]-> southS;    eastS -[moveSS]-> southS;
      eastF -[moveWS]-> westS;    eastS -[moveWS]-> westS;
      eastF -[moveES]-> eastS;    eastS -[moveES]-> eastS;
      eastF -[hover]-> hovering;  eastS -[hover]-> hovering;
      eastF -[charge]-> charging;  eastS -[charge]-> charging;
      eastF -[off]-> landed;     eastS -[off]-> landed;

      westF -[moveNF]-> northF;    westS -[moveNF]-> northF;
      westF -[moveSF]-> southF;    westS -[moveSF]-> southF;
      westF -[moveWF]-> westF;    westS -[moveWF]-> westF;
      westF -[moveEF]-> eastF;    westS -[moveEF]-> eastF;
      westF -[moveNS]-> northS;    westS -[moveNS]-> northS;
      westF -[moveSS]-> southS;    westS -[moveSS]-> southS;
      westF -[moveWS]-> westS;    westS -[moveWS]-> westS;
      westF -[moveES]-> eastS;    westS -[moveES]-> eastS;
      westF -[hover]-> hovering;  westS -[hover]-> hovering;
      westF -[charge]-> charging;  westS -[charge]-> charging;
      westF -[off]-> landed;     westS -[off]-> landed;

      charging -[moveNF]-> northF;  hovering -[moveNF]-> northF;
      charging -[moveSF]-> southF;  hovering -[moveSF]-> southF;
      charging -[moveWF]-> westF;  hovering -[moveWF]-> westF;
      charging -[moveEF]-> eastF;  hovering -[moveEF]-> eastF;
      charging -[moveNS]-> northS;  hovering -[moveNS]-> northS;
      charging -[moveSS]-> southS;  hovering -[moveSS]-> southS;
      charging -[moveWS]-> westS;  hovering -[moveWS]-> westS;
      charging -[moveES]-> eastS;        hovering -[moveES]-> eastS;
      charging -[hover]-> hovering;  hovering -[hover]-> hovering;
      charging -[charge]-> charging;  hovering -[charge]-> charging;
      charging -[off]-> landed;    hovering -[off]-> landed;
    properties
      Hybrid_SynchAADL::ContinuousDynamics =>
        "d/dt(y) =  #Spec::fastVel;
         d/dt(e) = -#Spec::fastVel * #Spec::h;" in modes (northF),
        "d/dt(y) = -#Spec::fastVel;
         d/dt(e) = -#Spec::fastVel * #Spec::h;" in modes (southF),
        "d/dt(x) = -#Spec::fastVel;
         d/dt(e) = -#Spec::fastVel * #Spec::h;" in modes (westF),
        "d/dt(x) =  #Spec::fastVel;
         d/dt(e) = -#Spec::fastVel * #Spec::h;" in modes (eastF),
        "d/dt(y) =  #Spec::slowVel;
         d/dt(e) = -#Spec::slowVel * #Spec::h;" in modes (northS),
        "d/dt(y) = -#Spec::slowVel;
         d/dt(e) = -#Spec::slowVel * #Spec::h;" in modes (southS),
        "d/dt(x) = -#Spec::slowVel;
         d/dt(e) = -#Spec::slowVel * #Spec::h;" in modes (westS),
        "d/dt(x) =  #Spec::slowVel;
         d/dt(e) = -#Spec::slowVel * #Spec::h;" in modes (eastS),
        "d/dt(e) = -#Spec::k;" in modes (hovering),
        "d/dt(e) =  #Spec::c ;" in modes (charging),
        "d/dt(e) = 0" in modes (landed);
  end DroneEnvironment.impl;
end DroneEnvironment;
```

```
package DroneControl
public
  with Base_Types, Data_Model, DataTypes;

  system DroneControl
    features
      req: out data port Base_Types::Boolean[];
      dest: in data port DataTypes::Vector[];
      oPos: out data port DataTypes::Vector;
      iPos: in data port DataTypes::Vector[];
      iX: in data port Base_Types::Float;  set_hover: out event port;
      iY: in data port Base_Types::Float;  set_charge: out event port;
      iE: in data port Base_Types::Float;  turn_off: out event port;
      northF: out event port;      northS: out event port;
      southF: out event port;      southS: out event port;
      westF: out event port;      westS: out event port;
      eastF: out event port;      eastS: out event port;
  end DroneControl;

  system implementation DroneControl.impl
    subcomponents
      proc: process DroneProcess.impl;
    connections
      C1: port proc.req -> req;    C3: port proc.oPos -> oPos;
      C2: port dest -> proc.dest;  C4: port iPos -> proc.iPos;
      C5: port iX -> proc.iX;    C17: port proc.set_hover -> set_hover;
      C6: port iY -> proc.iY;    C18: port proc.set_charge -> set_charge;
      C7: port iE -> proc.iE;    C19: port proc.turn_off -> turn_off;
      C9: port proc.northF -> northF;  C13: port proc.northS -> northS;
      C10: port proc.southF -> southF;  C14: port proc.southS -> southS;
      C11: port proc.westF -> westF;  C15: port proc.westS -> westS;
      C12: port proc.eastF -> eastF;  C16: port proc.eastS -> eastS;
  end DroneControl.impl;

  process DroneProcess
    features
      req: out data port Base_Types::Boolean[];
      dest: in data port DataTypes::Vector[];
      oPos: out data port DataTypes::Vector;
      iPos: in data port DataTypes::Vector[];
      iX: in data port Base_Types::Float;  set_hover: out event port;
      iY: in data port Base_Types::Float;  set_charge: out event port;
      iE: in data port Base_Types::Float;  turn_off: out event port;
      northF: out event port;      northS: out event port;
      southF: out event port;      southS: out event port;
      westF: out event port;      westS: out event port;
      eastF: out event port;      eastS: out event port;
  end DroneProcess;

  process implementation DroneProcess.impl
    subcomponents
      thrd: thread DroneThread.impl;
    connections
      C1: port thrd.req -> req;    C3: port thrd.oPos -> oPos;
      C2: port dest -> thrd.dest;  C4: port iPos -> thrd.iPos;
      C5: port iX -> thrd.iX;    C17: port thrd.set_hover -> set_hover;
      C6: port iY -> thrd.iY;    C18: port thrd.set_charge -> set_charge;
      C7: port iE -> thrd.iE;    C19: port thrd.turn_off -> turn_off;
      C9: port thrd.northF -> northF;  C13: port thrd.northS -> northS;
      C10: port thrd.southF -> southF;  C14: port thrd.southS -> southS;
      C11: port thrd.westF -> westF;  C15: port thrd.westS -> westS;
      C12: port thrd.eastF -> eastF;  C16: port thrd.eastS -> eastS;
  end DroneProcess.impl;
```

```
thread DroneThread
  features
    req: out data port Base_Types::Boolean[];
    dest: in data port DataTypes::Vector[];
    oPos: out data port DataTypes::Vector;
    iPos: in data port DataTypes::Vector[];
    iX: in data port Base_Types::Float;  set_hover: out event port;
    iY: in data port Base_Types::Float;  set_charge: out event port;
    iE: in data port Base_Types::Float;  turn_off: out event port;
    northF: out event port;      northS: out event port;
    southF: out event port;      southS: out event port;
    westF: out event port;      westS: out event port;
    eastF: out event port;      eastS: out event port;
end DroneThread;

--- to be refined; see DroneThread_D3P2.impl
thread implementation DroneThread.impl
  subcomponents
    goal: data DataTypes::Vector;
    chargeStation: data DataTypes::VectorArray;
    packetPos: data DataTypes::VectorArray;
    packetId: data Base_Types::Integer;
    packetCtr: data Base_Types::Integer {Data_Model::Initial_Value => ("0");};
    toChargingStation: data Base_Types::Boolean;
    delivering: data Base_Types::Boolean;
end DroneThread.impl;

subprogram next
  features
    id: in out parameter Base_Types::Integer;
    counter: in out parameter Base_Types::Integer;
    size: in parameter Base_Types::Integer;
end next;

subprogram implementation next.packet
  annex behavior_specification {**
    variables
      i: Base_Types::Integer;
    states
      init: initial state;    done: final state;
    transitions
      init -[counter = 0]-> done { counter := 1 };
      init -[0 < counter and counter < size]-> done {
        id := id + 1; counter := counter + 1; if (id > size) id := 1 end if
      };
      init -[counter >= size]-> done { counter := counter + 1 };
    **};
end next.packet;

subprogram chkClose
  features
    p: in parameter DataTypes::Vector;
    pos: in parameter DataTypes::VectorArray;
    cand: in parameter DataTypes::BooleanArray;
```

```
      size: in parameter Base_Types::Integer;
      thld: in parameter Base_Types::Float;
      output: out parameter Base_Types::Boolean;
  end chkClose;

  subprogram implementation chkClose.impl
    annex behavior_specification {**
      variables
        d: Base_Types::Float;
        i: Base_Types::Integer;
        r : Base_Types::Boolean;
      states
        s0: initial state;  sl: state;  sf: final state;
      transitions
        s0 -[]-> sl { r := false; i := 1 };
        sl -[r = false and i <= size]-> sl {
          DataTypes::getDist.l1 ! (p, pos[i], d);
          r := d < thld and cand[i]; i := i + 1
        };
        sl -[otherwise]-> sf{ output := r };
    **};
  end chkClose.impl;

  subprogram closeTo
    features
      currPos: in parameter DataTypes::Vector;
      candidates: in parameter DataTypes::VectorArray;
      size: in parameter Base_Types::Integer;
      target: out parameter DataTypes::Vector;
  end closeTo;

  subprogram implementation closeTo.fuel
    annex behavior_specification {**
      variables
        dist: Base_Types::Float;
        minDist: Base_Types::Integer;
        i: Base_Types::Integer;
      states
        init: initial state;  loop: state;  done: final state;
      transitions
        init -[size > 0]-> loop {
          DataTypes::getDist.l1 ! (currPos, candidates[1], minDist);
          target := candidates[1];
          i:= 2
        };
        loop -[i <= size]-> loop {
          DataTypes::getDist.l1 ! (currPos, candidates[i], dist);
          if (dist < minDist) minDist := dist; target := candidates[i] end if;
          i := i + 1
        };
        loop -[otherwise]-> done;
    **};
  end closeTo.fuel;
end DroneControl;
```

```
property set Spec is
  droneNum: inherit aadlinteger applies to (system, process, thread);
  packetNum: inherit aadlinteger applies to (system, process, thread);
  stationNum: inherit aadlinteger applies to (system, process, thread);
  droneId: inherit aadlInteger applies to (system, process, thread);
  packetId: inherit aadlInteger applies to (system, process, thread);

  close: constant aadlreal => 0.3;
  near: constant aadlreal => 0.8;
  maxFuel: constant aadlinteger => 200;
  minFuel: constant aadlinteger => 5;
  h: constant aadlinteger => 5;
  c: constant aadlinteger => 1000;
  k: constant aadlinteger => 2;
  fastVel: constant aadlreal => 3.5;
  slowVel: constant aadlreal => 2.0;
  padding: inherit aadlreal applies to (system, process, thread);
end Spec;
```

```
package PacketDeliverySystem_D3P2
public
  with PacketDeliverySystem, Packet_D3P2, Drone_D3P2;
  with Data_Model, Spec;

  system PacketDeliverySystem_D3P2 extends PacketDeliverySystem::PacketDelivery
    properties
      Classifier_Substitution_Rule => Type_Extension;
  end PacketDeliverySystem_D3P2;

  system implementation PacketDeliverySystem_D3P2.impl
    extends PacketDeliverySystem::PacketDelivery.abst
    subcomponents
      drone: refined to system Drone_D3P2::Drone_D3P2.impl[3];
      packet: refined to system Packet_D3P2::Packet_D3P2.impl[2];
    connections
      C1: refined to port {Connection_Set => (
          [src => (1, 1); dst => (1, 1);], [src => (2, 1); dst => (1, 2);],
          [src => (1, 2); dst => (2, 1);], [src => (2, 2); dst => (2, 2);],
          [src => (1, 3); dst => (3, 1);], [src => (2, 3); dst => (3, 2);]);};
      C2: refined to port {Connection_Set => (
          [src => (1, 1); dst => (1, 1);], [src => (1, 2); dst => (2, 1);],
          [src => (2, 1); dst => (1, 2);], [src => (2, 2); dst => (2, 2);],
          [src => (3, 1); dst => (1, 3);], [src => (3, 2); dst => (2, 3);]);};
    properties
      Spec::droneNum => 3 applies to drone, packet;
      Spec::packetNum => 2 applies to drone, packet;
      Spec::stationNum => 1 applies to drone;
      Spec::droneId => 1 applies to drone[1];
      Spec::droneId => 2 applies to drone[2];
      Spec::droneId => 3 applies to drone[3];
      Data_Model::Dimension => (1) applies to
        drone[1].ctrl.proc.thrd.packetPos, drone[2].ctrl.proc.thrd.packetPos,
        drone[3].ctrl.proc.thrd.packetPos;
```

```
--- Drone Output Port Initialization
Data_Model::Initial_Value => ("false") applies to
  drone[1].req[1], drone[1].req[2],
  drone[2].req[1], drone[2].req[2],
  drone[3].req[1], drone[3].req[2];
Data_Model::Initial_Value => ("{x : 2.2,  y : 0.8}")
  applies to drone[1].oPos;
Data_Model::Initial_Value => ("{x : 0.8,  y : 2.2}")
  applies to drone[2].oPos;
Data_Model::Initial_Value => ("{x : 12.0, y : 12.0}")
  applies to drone[3].oPos;

--- Drone Controller Output Ports Initialization
Data_Model::Initial_Value => ("false") applies to
  drone[1].ctrl.req[1], drone[1].ctrl.req[2],
  drone[2].ctrl.req[1], drone[2].ctrl.req[2],
  drone[3].ctrl.req[1], drone[3].ctrl.req[2];
Data_Model::Initial_Value => ("{x : 2.2,  y : 0.8}")
  applies to drone[1].ctrl.oPos;
Data_Model::Initial_Value => ("{x : 0.8,  y : 2.2}")
  applies to drone[2].ctrl.oPos;
Data_Model::Initial_Value => ("{x : 12.0, y : 12.0}")
  applies to drone[3].ctrl.oPos;

--- Drone Thread Subcomponents Initialization
Data_Model::Initial_Value => (
  "[{x : 5.0, y : 2.2}, {x : 2.2, y : 5.0}]") applies to
  drone[1].ctrl.proc.thrd.packetPos,
  drone[2].ctrl.proc.thrd.packetPos,
  drone[3].ctrl.proc.thrd.packetPos;
Data_Model::Initial_Value => (
  "[{x : 5.0, y : 5.0}]") applies to
  drone[1].ctrl.proc.thrd.chargeStation,
  drone[2].ctrl.proc.thrd.chargeStation,
  drone[3].ctrl.proc.thrd.chargeStation;
Data_Model::Initial_Value => (
  "{x : 0.0, y : 0.0}") applies to
  drone[1].ctrl.proc.thrd.goal,
  drone[2].ctrl.proc.thrd.goal,
  drone[3].ctrl.proc.thrd.goal;
Data_Model::Initial_Value => ("2") applies to
  drone[1].ctrl.proc.thrd.packetId,
  drone[3].ctrl.proc.thrd.packetId;
Data_Model::Initial_Value => ("1") applies to
  drone[2].ctrl.proc.thrd.packetId;

--- Drone Environment Subcomponents Initialization
Data_Model::Initial_Value => ("any") applies to
  drone[1].env.x, drone[1].env.y,
  drone[2].env.x, drone[2].env.y,
  drone[3].env.x, drone[3].env.y;
Data_Model::Initial_Value => ("200.0") applies to
  drone[1].env.e, drone[2].env.e, drone[3].env.e;
```

```
      --- Packet Output Port Initialize
      Data_Model::Initial_Value => ("{x : 0.0, y : 0.0}") applies to
      packet[1].dest[1], packet[1].dest[2], packet.dest[3],
      packet[2].dest[1], packet[2].dest[2], packet.dest[3];

      --- Packet Subcomponent Initialize
      Data_Model::Initial_Value => ("{x : 5.0, y : 0.0}") applies to
        packet[1].packetProc.packetThread.destination;
      Data_Model::Initial_Value => ("{x : 0.0, y : 5.0}") applies to
        packet[2].packetProc.packetThread.destination;
  end PacketDeliverySystem_D3P2.impl;
end PacketDeliverySystem_D3P2;
```

```
package Packet_D3P2
public
  with Base_Types, DataTypes
  with Packet;

  system Packet_D3P2 extends Packet::Packet
    features
      req: refined to in data port Base_Types::Boolean[3];
      dest: refined to out data port DataTypes::Vector[3];
    properties
      Classifier_Substitution_Rule => Type_Extension;
  end Packet_D3P2;

  system implementation Packet_D3P2.impl extends Packet::Packet.impl
    subcomponents
      packetProc: refined to process PacketProc_D3P2.impl;
  end Packet_D3P2.impl;

  process PacketProc_D3P2 extends Packet::PacketProc
    features
      req: refined to in data port Base_Types::Boolean[3];
      dest: refined to out data port DataTypes::Vector[3];
    properties
      Classifier_Substitution_Rule => Type_Extension;
  end PacketProc_D3P2;

  process implementation PacketProc_D3P2.impl extends Packet::PacketProc.impl
    subcomponents
      packetThread: refined to thread PacketThread_D3P2.impl;
  end PacketProc_D3P2.impl;

  thread PacketThread_D3P2 extends Packet::PacketThread
    features
      req: refined to in data port Base_Types::Boolean[3];
      dest: refined to out data port DataTypes::Vector[3];
  end PacketThread_D3P2;

  thread implementation PacketThread_D3P2.impl extends Packet::PacketThread.impl
  end PacketThread_D3P2.impl;
end Packet_D3P2;
```

```
package Drone_D3P2
public
  with Base_Types, DataTypes;
  with Drone, DroneControl_D3P2;

  system Drone_D3P2 extends Drone::Drone
    features
      req: refined to out data port Base_Types::Boolean[2];
      dest: refined to in data port DataTypes::Vector[2];
      iPos: refined to in data port DataTypes::Vector[3];
  end Drone_D3P2;

  system implementation Drone_D3P2.impl extends Drone::Drone.impl
    subcomponents
      ctrl: refined to system DroneControl_D3P2::DroneControl_D3P2.impl;
  end Drone_D3P2.impl;

end Drone_D3P2;
```

```
package DroneControl_D3P2
public
  with Base_Types, DataTypes;
  with DroneControl;

  system DroneControl_D3P2 extends DroneControl::DroneControl
    features
      req: refined to out data port Base_Types::Boolean[2];
      dest: refined to in data port DataTypes::Vector[2];
      iPos: refined to in data port DataTypes::Vector[3];
    properties
      Classifier_Substitution_Rule => Type_Extension;
  end DroneControl_D3P2;

  system implementation DroneControl_D3P2.impl
      extends DroneControl::DroneControl.impl
    subcomponents
      proc: refined to process DroneProcess_D3P2.impl;
  end DroneControl_D3P2.impl;

  process DroneProcess_D3P2 extends DroneControl::DroneProcess
    features
      req: refined to out data port Base_Types::Boolean[2];
      dest: refined to in data port DataTypes::Vector[2];
      iPos: refined to in data port DataTypes::Vector[3];
    properties
      Classifier_Substitution_Rule => Type_Extension;
  end DroneProcess_D3P2;

  process implementation DroneProcess_D3P2.impl
      extends DroneControl::DroneProcess.impl
    subcomponents
      thrd: refined to thread DroneThread_D3P2.impl;
  end DroneProcess_D3P2.impl;
```

```
thread DroneThread_D3P2 extends DroneControl::DroneThread
  features
    req: refined to out data port Base_Types::Boolean[2];
    dest: refined to in data port DataTypes::Vector[2];
    iPos: refined to in data port DataTypes::Vector[3];
end DroneThread_D3P2;

thread implementation DroneThread_D3P2.impl
    extends DroneControl::DroneThread.impl
annex behavior_specification {**
    variables
      cur: DataTypes::Vector;
      d: Base_Types::Float;
      error: Base_Types::Boolean;
      collide: Base_Types::Boolean;
      pos: DataTypes::VectorArray;
      rec: DataTypes::BooleanArray;
      extra_padding: Base_Types::Integer;
    states
      init: initial complete state;
      select_packet, select_packet_aux: state;
      move: complete state;
      choose_action, choose_action_aux: state;
      arrived: state;
      plan: state;
      find_station: state;
      charge: complete state;
      wait_packet: complete state;
      pickup_packet: complete state; pickup_packet_aux: state;
      done: final state;
      fail: final state;
    transitions
      init -[on dispatch]-> select_packet {
        cur.x := iX; cur.y := iY;
        toChargingStation := false;
        delivering := false
      };

      select_packet -[]-> select_packet_aux {
        DroneControl::next.packet!(packetId, packetCtr, #Spec::packetNum)
      };
      select_packet_aux -[packetCtr > #Spec::packetNum]-> done { turn_off! };
      select_packet_aux -[otherwise]-> choose_action {
        goal := packetPos[packetId] };

      move -[on dispatch]-> choose_action { cur.x := iX; cur.y := iY };

      choose_action -[]-> choose_action_aux {
        for(forIdx : Base_Types::Integer in 1 .. #Spec::droneNum){
          pos[forIdx] := iPos[forIdx];
          rec[forIdx] := forIdx != #Spec::droneId and iPos[forIdx]'fresh
        };
        DroneControl::chkClose.impl ! (pos[#Spec::droneId], pos, rec,
                                      #Spec::droneNum, #Spec::close, error);
        DataTypes::getDist.l1 ! (cur, goal, d)
      };
      choose_action_aux -[error]-> fail { turn_off! };
      choose_action_aux -[not error and d < #Spec::near]-> arrived;
      choose_action_aux -[not error and d >= #Spec::near and
                          iE < #Spec::minFuel]-> fail { turn_off! };
      choose_action_aux -[not error and d >= #Spec::near and
                          iE >= d * #Spec::maxFuel * 0.1]-> plan;
      choose_action_aux -[not error and d >= #Spec::near and
                          iE <  d * #Spec::maxFuel * 0.1 and
                          (not toChargingStation)]-> find_station;

      find_station -[]-> plan {
        DroneControl::closeTo.fuel ! (cur, chargeStation,
                                     #Spec::stationNum, goal);
        toChargingStation := true
      };

      arrived -[not toChargingStation and not delivering]-> wait_packet {
        req[packetId] := true;
        oPos := cur; set_hover!
      };
      arrived -[not toChargingStation and delivering]-> init {
        oPos := cur; set_hover!
      };
      arrived -[toChargingStation]-> charge {
        set_charge!
      };

      plan -[]-> move {
        extra_padding := 0.15;
        DroneControl::chkClose.impl ! (cur, pos, rec, #Spec::droneId - 1,
                                      #Spec::close + extra_padding, collide);
        if (collide) set_hover!
        elsif (goal.x - cur.x > 1.5) eastF!
        elsif (goal.x - cur.x > 0.3) eastS!
        elsif (cur.x - goal.x > 1.5) westF!
        elsif (cur.x - goal.x > 0.3) westS!
        elsif (goal.y - cur.y > 1.5) northF!
        elsif (goal.y - cur.y > 0.3) northS!
        elsif (cur.y - goal.y > 1.5) southF!
        elsif (cur.y - goal.y > 0.3) southS!
        else set_hover!
        end if;
        oPos := cur
      };

      charge -[on dispatch]-> choose_action {
        cur.x := iX; cur.y := iY ;
        toChargingStation := false;
        if (not delivering)
          goal := packetPos[packetId]
        else
```

```
          goal := dest[packetId]
        end if
      };

      wait_packet -[on dispatch]-> pickup_packet {
        cur.x := iX; cur.y := iY ;
        oPos := cur; set_hover!
      };

      pickup_packet -[on dispatch]-> pickup_packet_aux {
        cur.x := iX; cur.y := iY };
      pickup_packet_aux -[dest[packetId]'fresh]-> choose_action {
        delivering := true;
        goal := dest[packetId]
      };
      pickup_packet_aux -[not dest[packetId]'fresh]-> select_packet;
    **};
  end DroneThread_D3P2.impl;

end DroneControl_D3P2;
```

```
package PacketDeliverySystem_D3P2_Patch
public
  with Drone_D3P2_Patch, PacketDeliverySystem_D3P2;

  system PacketDeliverySystem_D3P2_Patch
      extends PacketDeliverySystem_D3P2::PacketDeliverySystem_D3P2
  end PacketDeliverySystem_D3P2_Patch;

  system implementation PacketDeliverySystem_D3P2_Patch.impl
      extends PacketDeliverySystem_D3P2::PacketDeliverySystem_D3P2.impl
    subcomponents
      drone: refined to system Drone_D3P2_Patch::Drone_D3P2_Patch.impl[3];
  end PacketDeliverySystem_D3P2_Patch.impl;
end PacketDeliverySystem_D3P2_Patch;
```

```
package Drone_D3P2_Patch
public
  with DroneControl_D3P2_Patch, Drone_D3P2;

  system Drone_D3P2_Patch extends Drone_D3P2::Drone_D3P2
  end Drone_D3P2_Patch;

  system implementation Drone_D3P2_Patch.impl extends Drone_D3P2::Drone_D3P2.impl
    subcomponents
      ctrl: refined to system
        DroneControl_D3P2_Patch::DroneControl_D3P2_Patch.impl;
  end Drone_D3P2_Patch.impl;
end Drone_D3P2_Patch;
```

```
package DroneControl_D3P2_Patch
public
  with Base_Types, DataTypes, DroneControl, DroneControl_D3P2;
```

```
system DroneControl_D3P2_Patch extends DroneControl_D3P2::DroneControl_D3P2
end DroneControl_D3P2_Patch;

system implementation DroneControl_D3P2_Patch.impl
    extends DroneControl_D3P2::DroneControl_D3P2.impl
  subcomponents
    proc: refined to process DroneProcess_D3P2_Patch.impl;
end DroneControl_D3P2_Patch.impl;

process DroneProcess_D3P2_Patch extends DroneControl_D3P2::DroneProcess_D3P2
end DroneProcess_D3P2_Patch;

process implementation DroneProcess_D3P2_Patch.impl
    extends DroneControl_D3P2::DroneProcess_D3P2.impl
  subcomponents
    thrd: refined to thread DroneThread_D3P2_Patch.impl;
end DroneProcess_D3P2_Patch.impl;

thread DroneThread_D3P2_Patch extends DroneControl_D3P2::DroneThread_D3P2
end DroneThread_D3P2_Patch;

thread implementation DroneThread_D3P2_Patch.impl
    extends DroneControl_D3P2::DroneThread_D3P2.impl
annex behavior_specification {**
    variables
      cur: DataTypes::Vector;
      d: Base_Types::Float;
      error: Base_Types::Boolean;
      collide: Base_Types::Boolean;
      pos: DataTypes::VectorArray;
      rec: DataTypes::BooleanArray;
      extra_padding: Base_Types::Integer;
      sample_upper: Base_Types::Float;
      sample_lower: Base_Types::Float;
      response_upper : Base_Types::Float;
      sample_padding: Base_Types::Float;
      response_padding: Base_Types::Float;
    states
      init: initial complete state;
      select_packet, select_packet_aux: state;
      move: complete state;
      choose_action, choose_action_aux: state;
      arrived: state;
      plan: state;
      find_station: state;
      charge: complete state;
      wait_packet: complete state;
      pickup_packet: complete state; pickup_packet_aux: state;
      done: final state;
      fail: final state;
    transitions
      init -[on dispatch]-> select_packet {
        cur.x := iX; cur.y := iY;
        toChargingStation := false; delivering := false
      };

      select_packet -[]-> select_packet_aux {
        DroneControl::next.packet!(packetId, packetCtr, #Spec::packetNum)
      };
      select_packet_aux -[packetCtr > #Spec::packetNum]-> done { turn_off! };
      select_packet_aux -[otherwise]-> choose_action {
        goal := packetPos[packetId]
      };

      move -[on dispatch]-> choose_action { cur.x := iX; cur.y := iY };

      choose_action -[]-> choose_action_aux {
        for(forIdx : Base_Types::Integer in 1 .. #Spec::droneNum){
          pos[forIdx] := iPos[forIdx];
          rec[forIdx] := forIdx != #Spec::droneId and iPos[forIdx]'fresh
        };
        DroneControl::chkClose.impl ! (pos[#Spec::droneId], pos, rec,
                                       #Spec::droneNum, #Spec::close, error);
        DataTypes::getDist.l1 ! (cur, goal, d)
      };
      choose_action_aux -[error]-> fail { turn_off! };
      choose_action_aux -[not error and d < #Spec::near]-> arrived;
      choose_action_aux -[not error and d >= #Spec::near and
                          iE < #Spec::minFuel]-> fail { turn_off! };
      choose_action_aux -[not error and d >= #Spec::near and
                          iE >= d * #Spec::maxFuel * 0.1]-> plan;
      choose_action_aux -[not error and d >= #Spec::near and
                          iE <  d * #Spec::maxFuel * 0.1 and
                          (not toChargingStation)]-> find_station;

      find_station -[]-> plan {
        DroneControl::closeTo.fuel ! (cur, chargeStation,
                                      #Spec::stationNum, goal);
        toChargingStation := true
      };

      arrived -[not toChargingStation and not delivering]-> wait_packet {
        req[packetId] := true; oPos := cur; set_hover!
      };
      arrived -[not toChargingStation and delivering]-> init {
        oPos := cur; set_hover!
      };
      arrived -[toChargingStation]-> charge { set_charge! };

      charge -[on dispatch]-> choose_action {
        cur.x := iX; cur.y := iY ; toChargingStation := false;
        if (not delivering) goal := packetPos[packetId]
        else goal := dest[packetId] end if
      };

      plan -[]-> move {
        sample_upper := #Hybrid_SynchAADL::Sampling_Time.upper_bound;
```

```
          sample_lower := #Hybrid_SynchAADL::Sampling_Time.lower_bound;
          response_upper := #Hybrid_SynchAADL::Response_Time.upper_bound;
          sample_padding := (sample_upper - sample_lower +
            #Hybrid_SynchAADL::Max_Clock_Deviation * 2
            + #Hybrid_SynchAADL::Period) * #Spec::fastVel;
          response_padding := (response_upper - sample_lower +
            #Hybrid_SynchAADL::Max_Clock_Deviation * 2) * #Spec::fastVel * 2;
          extra_padding := 0.1;
          DroneControl::chkClose.impl ! (cur, pos, rec, #Spec::droneId - 1,
                                         #Spec::close + sample_padding +
                                         response_padding + extra_padding,
                                         collide);
          if (collide) set_hover!
          elsif (goal.x - cur.x > 1.5) eastF!
          elsif (goal.x - cur.x > 0.3) eastS!
          elsif (cur.x - goal.x > 1.5) westF!
          elsif (cur.x - goal.x > 0.3) westS!
          elsif (goal.y - cur.y > 1.5) northF!
          elsif (goal.y - cur.y > 0.3) northS!
          elsif (cur.y - goal.y > 1.5) southF!
          elsif (cur.y - goal.y > 0.3) southS!
          else set_hover! end if;
          oPos := cur
        };

        wait_packet -[on dispatch]-> pickup_packet {
          cur.x := iX; cur.y := iY ;
          oPos := cur; set_hover!
        };

        pickup_packet -[on dispatch]-> pickup_packet_aux {
          cur.x := iX; cur.y := iY };
        pickup_packet_aux -[dest[packetId]'fresh]-> choose_action {
          delivering := true;
          goal := dest[packetId]
        };
        pickup_packet_aux -[not dest[packetId]'fresh]-> select_packet;
    **};
  end DroneThread_D3P2_Patch.impl;

end DroneControl_D3P2_Patch;
```