Combined Formal Modeling and Model Transformation Based on AADL and Object-Z

Zhengling Guo[1], Zining Cao[1, 2, 3, 4*]

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China.
[2] Science and Technology on Electro-optic Control Laboratory, Luoyang, China.
[3] Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210023, China.
[4] MIIT Key Laboratory of Pattern Analysis and Machine Intelligence, Nanjing, China.

* Corresponding author. Email: caozn@nuaa.edu.cn (Z.C.)
Manuscript submitted February 13, 2023; revised May 20, 2023; accepted July 20, 2023.
doi: 10.17706/jsw.18.4.185-199

Abstract: Formal methods have become more and more widely used in safety-critical software engineering. A system should be specified with a formal model such as automata, Petri nets, and process algebras to be formally verified. We investigated the AADL combined with Object-Z modeling approach for subsequent formal verification work. The advantage of this is that object-oriented ideas can be used for the AADL modeling process. The space-saving effect is achieved by using class inheritance and polymorphism to extract commonalities. In this paper, we present a new formal model with a more powerful ability——OZIA expressed in the language Object-Z. The transformation rules from the AADL-Object-Z model to the OZIA model are defined to support formal verification. Finally, an example illustrates our results with the Aircraft Landing Process case study.

Key words: Object-Z, AADL behavior annex, OZIA, model transformation

1. Introduction

Complex embedded real-time systems are widely used in the fields of avionics, spacecraft, automotive control, etc. These systems are resource-constrained, real-time response, fault-tolerant, dedicated hardware, and have high requirements for real-time, reliability, and other properties. So how to properly model them to make them safe and reliable is a common problem faced in academia and industry [1]. In response to this increasingly urgent problem, the Society of Automotive Engineers (SAE) proposed the embedded real-time system architecture analysis and design standard—AADL(Architecture Analysis and Design Language, AADL) in 2004, a very powerful analysis and design language, which is increasingly used in the avionics, flight control, and other aerospace fields of embedded real-time system architecture design and analysis [2], [3].

AADL is a model-based engineering language. Although it is good at describing the architectural features of the system hardware and software, it is helpless in constraining the variables of the system state and the constraint relations of variable changes. To solve this problem, we use Object-Z [4] to extend the BA(AADL Behavior Annex, BA) [5], which is an object-oriented extension of Z. The advantage of Object-Z is that it is first specified by a class. The benefit is that the system is regulated by first specifying the behavior of its constituent objects through classes, utilizing inheritance and polymorphism. Z, however, does not provide

such an approach. Since AADL is a semi-formal model, it cannot be directly formalized for verification. For this reason, we use the hybrid interface automaton (OZIA) as a formal model.

This paper is organized as follows. Section 2 presents an overview of AADL and its formalization. Section 3 presents BA and Object-Z respectively by providing simple examples of their use. And then the specification of Object-Z expansion for BA is presented. Since AADL is semi-formal, we transform it into an interface automaton to facilitate the subsequent formal verification work in section 4. We give the correctness of the transformation by using bisimulation techniques in the end. The Aircraft Landing Process case study is presented as a validation of the usefulness of this work in Section 5. We give a general overview of formal verification in Section 6. Finally, Section 7 gives the conclusions and future directions. The structure of this paper is shown in Fig. 1.

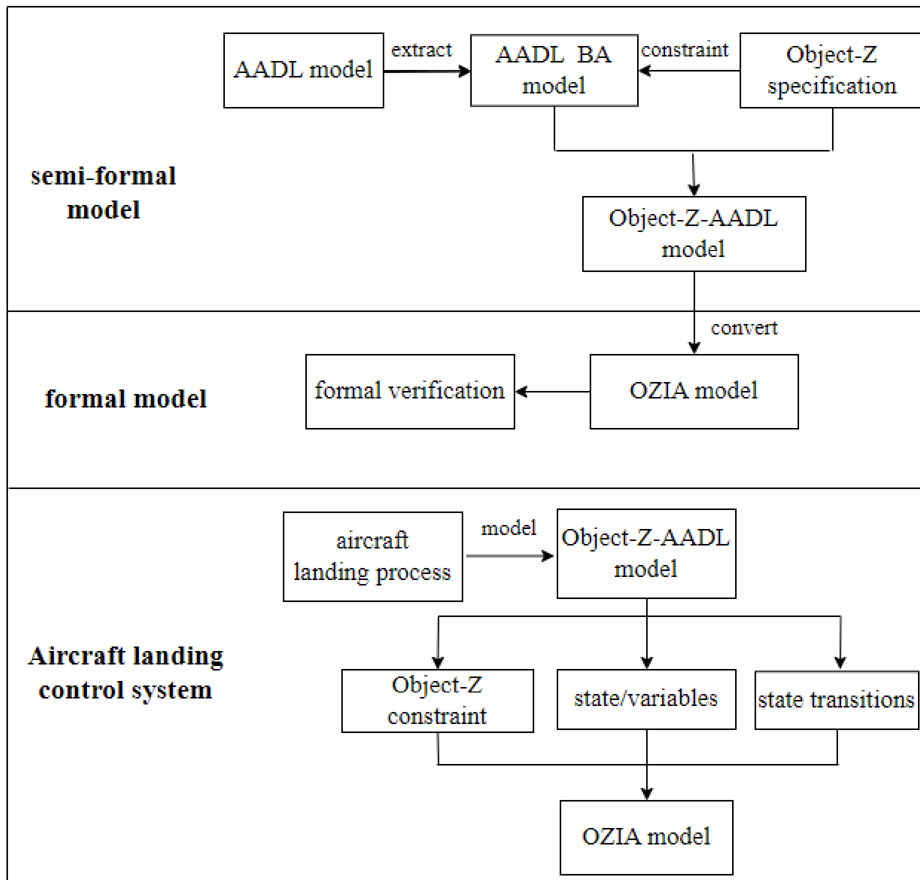


Fig. 1. The structure of this article.

2. Related Work

Due to its lack of formal specification and semantics, AADL cannot be used directly for formal verification. It is often transformed into several formal models that are employed with existing formal analysis tools such as UPPAL, Tina, and Polychrony. We have studied several formal approaches to AADL, classifying past work according to whether they support software, hardware, support for data constraint, support for object-oriented, support for model transformation, and support for verification. As shown in Table 1. This paper supports all the above five indicators.

Table 1. Comparison of related AADL formal approaches

Modeling Tools	Support for software,hardware	Support for data constraint	Support for object-oriented	Support for model transformation	Support for verification
HBA	Yes	No	No	Yes	Yes
AADL-DEVS	Yes	Yes	No	Yes	No
HAADL	Yes	Yes	No	Yes	No
BA and EA	Yes	No	No	Yes	No
WPBA	No	No	No	Yes	Yes
EMA	No	Yes	No	Yes	Yes
AADL	No	Yes	No	Yes	Yes
BA	Yes	Yes	No	Yes	Yes
AADL-BA	Yes	No	No	Yes	Yes
HA	Yes	Yes	No	Yes	Yes
This paper	Yes	Yes	Yes	Yes	Yes

3. AADL and Object-Z

3.1. AADL Modeling Specification

The main modeling notion of AADL is a component. Components can represent a software application or an execution platform [6]. The syntax of AADL is simple and powerful. It supports the hardware and software of embedded real-time system architecture design, analysis, and non-functional properties to add the function of the system description, design, analysis, and validation. It mainly includes three categories of components as shown in Fig. 2.

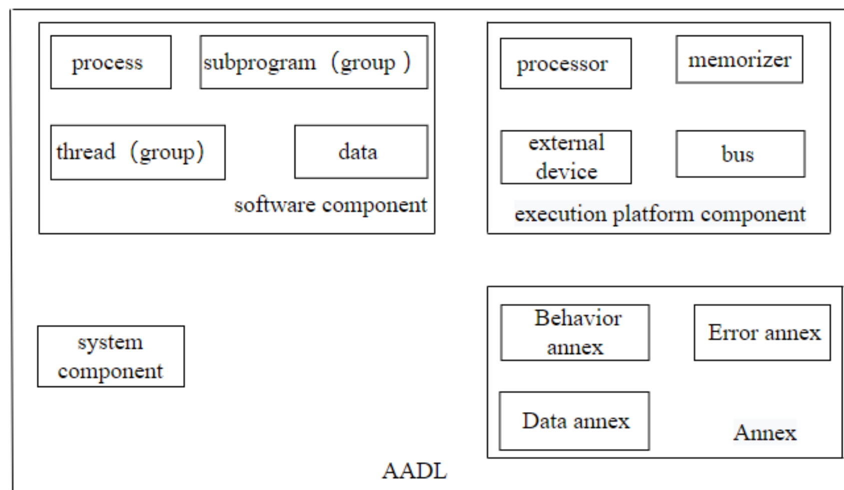


Fig. 2. The organization structure of AADL.

The formal study of AADL semantics is an extremely huge project. It brings the capability to enrich the model with additional information through a set of standard properties and annexes. Big and specific additions are specified by separate annexes such as the BA (AADL Behavior Annex, BA) to specify the architectural behavior [7]. To use formal methods for better modeling and analysis, we select a subset of AADL as the research object——BA. It was proposed in 2006 by the Laboratory of the Institute of Computer and Information Studies in Toulouse, France. BA is a simple system of state transition relationships, consisting of three main parts: *Variables*, *States,* and *Transitions*[8]. BA is defined as follows:

Definition 1 (AADL Behavior Annex, BA). BA is a four tuple $\mathrm{BA} = (Var, S, S_0, T)$, where

(1) Var is a finite set of local variables used in the automaton. where the local variables can be either the AADL base type (Base_Types) datatypes defined by the attribute set, or they can be user-defined datatypes. Local variables can be assigned values and can be compared;

(2) S is a non-empty finite set of states, including three types: initial state, complete state, and final state;

(3) $S_0 \subseteq S$ is a finite set of initial states. In a behavioral attachment, there must be only one initial state;

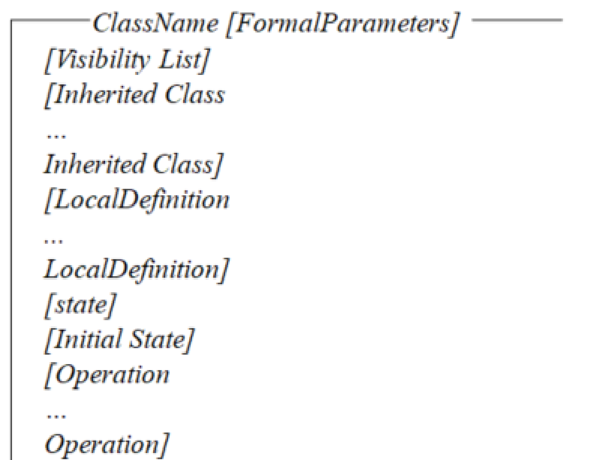
(4) $T \subseteq s \times (Guard \times Action) \times s$ is the transition function. A transition connects two states S and S'. A transition has a condition guard or a Boolean value. BA uses $S \overset{g,a}{\rightarrow}_1 S'$ represents a transition. When g is defaulted, can be omitted. When the transition is complete, action a is executed.

3.2. Object-Z

Object-Z is an object-oriented extension of the specification language Z, developed by researchers at the Center for Software Verification Research at the University of Queensland, Australia, which has been developed over several years. Object-Z offers a more powerful mechanism by splitting the specification into several interacting classes and objects. In Object-Z, a specification consists of a collection of classes. Within a class, the state and operations are written using schemas. Each class consists of a state space and an initialization together with a collection of operations that change the state[9]. It has complete axiomatic and denotational semantics. In terms of expressiveness and language readability, Object-Z is stronger than Z language. Currently, the Object-Z specification language has been widely used in academia and industry.

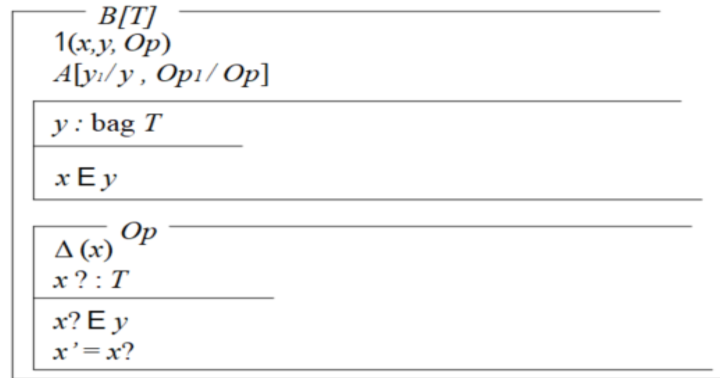
One of the most important concepts of object orientation is inheritance. The purpose of inheritance is to achieve incremental specification by building complex classes from simple components. Inheritance works as follows. All content (properties, operations, etc.) is inherited except for the visibility list. The type and constant definitions of the inherited class are merged with those explicitly declared in the derived class. The state patterns and operation schemas of inherited and derived classes are also merged. Inherited operations with names different from those in the new class are implicitly included in the inheritance[10].

The classes in a specification can be related in several ways. For example, an Object-Z class may inherit another class. Such inheritance allows complex classes and specifications to be built from simpler components iteratively in a similar way to the use of schema inclusion at an operational level. Instantiation is also supported because a class is viewed as a template for objects of that class, enabling classes to refer to objects of other classes as state variables. For example, the template of schema is as follows:



An Object-Z specification of a system typically includes several class definitions related to inheritance and instantiation[11]. Declared in terms of a generic parameter can only be used in expressions and predicates which are not type-specific. For example, the template of schema class *A[T]* is as follows:

The only construct of a class that is not inherited is the visibility list. The visibility list of the inheriting class is, therefore, totally independent of that of the inherited class. Hence, inherited features can be effectively canceled. For example, the following class *B[T]* inherits the class *A[T]* and redefines the variable y to be a bag, rather than a set of elements of type *T*.



Object-Z not only extends the syntax of Z but also the semantic universe in which specifications are given meaning. In particular, it allows variables to be declared which, rather than directly representing a value, refer to a value in much the same way as pointers in a programming language. A semantics supporting such variables is called reference semantics.

3.3. Combination of AADL and Object-Z

This section expands the AADL modeling specification based on the Object-Z pattern. We select a subset of AADL as the object of study. AADL comes with a behavior annex that includes state and transition, the lack of ability to describe the nature of the data constraints between components. So we introduce the Object-Z language to expand the Behavior Annex of AADL and add formal descriptions to its variables. Since the formalization of the complete semantics of AADL is very large, a subset of AADL is considered, including Thread, connection, behavior annex, SOM, etc. We describe the real-time system with data constraints in the AADL-Object-Z specification as a six-tuple defined as follows:

Definition 2 T is a six tuple $T = (S_T^Z, S_T^{init}, D_T, P_T, A_T^Z, TR_T^Z)$, where

(1)$S_T^Z = \{s : i \in \mathrm{N}\}$ is the set of states in the system, i.e., the state *States* in *Behavior Annex*, which is an expanded set of state patterns;

(2)S_T^{init} is a finite set of system states;

(3)D_T is a collection of data properties described in the system. The properties include not only the constraint properties such as *Guard in BA_Transition* but also the data properties defined in threads and processes, etc;

(4)P_T is the set of interfaces in the system;

(5)A_T^Z is the set of actions in a system,i.e.*Computation in BA_Transition*;

(6)TR_T^Z denotes the set of state transitions in the system, i.e. the collection of state transition *Transition Behavior Annex*.

Which, to distinguish it from the previous one, we make the mark with "*Z*" in the upper right corner of the element. To better illustrate the specification, we give an example of the diagnosis and treatment process of the diabetes system (DDS), including four parts: diagnosis of diabetes, diagnosis of diabetes type (I and II), diagnosis of complications, and treatment plan. The system model is shown in Fig. 3.

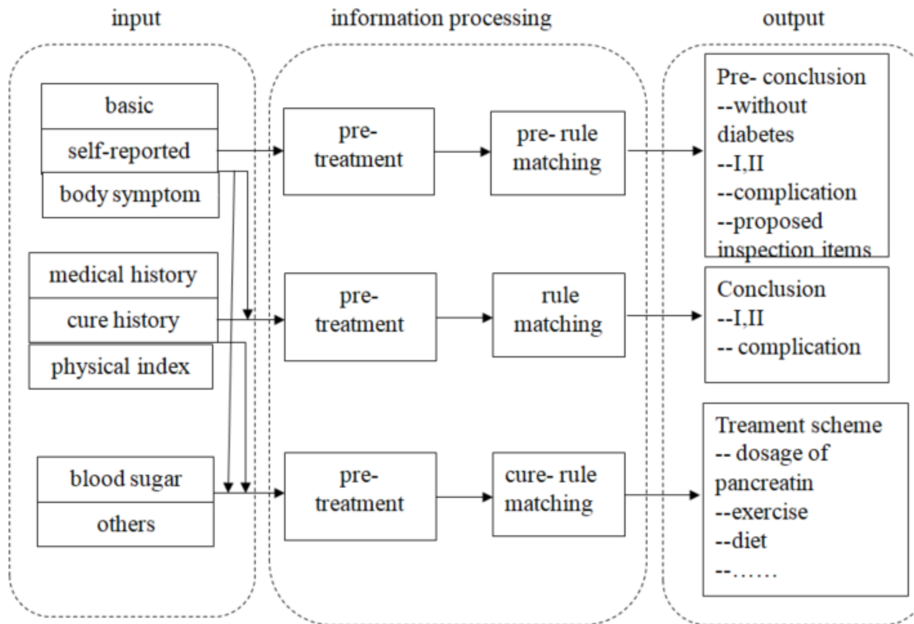


Fig. 3. The model of diagnosis and treatment process.

The system is modeled as a partial function from input symptoms to diagnostic results. The pattern of the DDS_0 system is as follows：

DDS_0

basic informaton : $\mathbb{P}$
body symptom: $\mathbb{P}$
physical index: $\mathbb{N}$
blood sugar: $\mathbb{R}$
trans: $\mathbb{B}$

$0 \notin$ *physical index*

Init

basic informaton$=\emptyset \wedge$ *body symptom*$=\emptyset \wedge \neg$*trans*

Rule-matching

Δ (*rules*, *reasoning rules*)

trans'=*trans* $\cup$ {(*physical index*!,0)}

Pre-treament

Δ (*basic informaton*, *body symptom*, *blood sugar*)

trans'=*trans* $\cup$ {(*rules*!,0)}

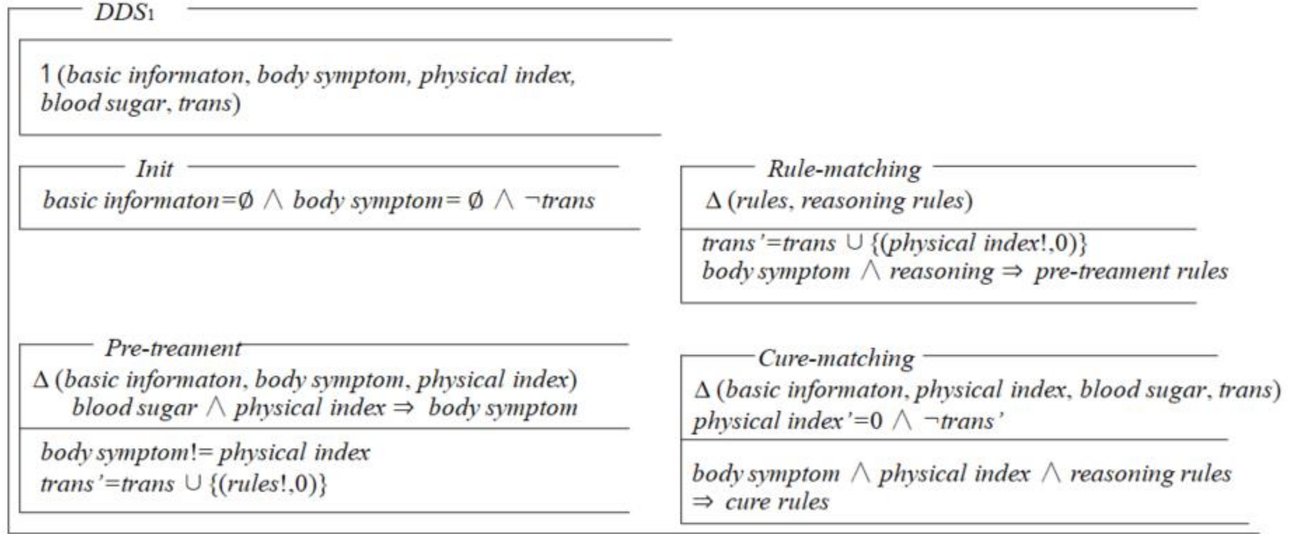
Cure-matching

Δ (*basic informaton*, *body symptom*, *blood sugar*, *trans*)

physical index'=0 $\wedge$ $\neg$*trans'*

In DDS_1, five variables express the state: *basic information*, a *body symptom*, *physical index*, *blood sugar,* and *trans*. Then transitions include four patterns, where *INIT* represents the state of not seeking medical treatment, *Pre-treatment* represents the state of pre-diagnosis, and *Rule-matching* represents the matching of the patient's physical examination parameters with the diagnosis rules. *Cure-matching* stands for

prescribing the right medicine. When the patient passes further diagnosis, the system transitions to *DDS₁*.



4. Model Transformation

4.1. OZIA Specification

The interface automaton with Object-Z, also known as the OZIA combined specification, is a combination of the Object-Z language and the interface automaton specification to form a capability that combines the nature of the interface model and the nature of the data constraints.

Interface Automata (IA) is a lightweight component specification language based on automata that treat each activity in a component as an input, output, or internal activity[12]. The definition is as follows:

Definition 3 (Interface Automata, IA). A interface automata is a six tuple $\text{IA} = \left(V_P, V_P^{init}, A_P^I, A_P^O, A_P^H, \mathcal{T}_P\right)$, where

(1) V_P is a set of states ;

(2) $V_P^{init} \subseteq V_P$ is an initial set of states, V_P^{init} contains at most one state; if $V_P^{init} = \emptyset$, then IA is null ;

(3) A_P^I, A_P^O, A_P^H is a disjoint set of inputs,outputs and internal actions, using $A_P = A_P^I \cup A_P^O \cup A_P^H$ to represent the set consisting of all actions ;

(4) $\mathcal{T}_\mathrm{P} \subseteq \mathrm{V_P} \times \mathrm{A_P} \times \mathrm{V_P}$ is a set of transition actions.

In which, if there are only certain internal actions in the interface automaton P, the symbols are denoted as $A_P^I = A_P^O = \emptyset$, then we consider the interface automaton P closed; otherwise open. When interface automata provide a specification approach for interface behavioral properties, they cannot describe the state specification of data structures. The Object-Z language specification, on the other hand, can specify the state of the system, but it is not suitable for the description of behavioral properties. Therefore, combining the two to form the interface automaton OZIA with Z allows us to describe both static and dynamic properties of the system, which we also call the interface automaton with data constraints. The definition is as follows:

Definition 4(OZIA) OZIA is a eleven tuple $\text{OZIA} = (S_P, S_P^{init}, A_P^I, A_P^O, A_P^H, V_P^I, V_P^O, V_P^H, F_P^V, F_P^A, \mathcal{T}_P)$, where

(1) S_P is the set of system states.

(2) $S_P^{init} \subseteq S_P$ is a finite set of system states. if $V_P^{init} = \emptyset$, then P is null;

(3) A_P^I, A_P^O, A_P^H is a disjoint set of inputs, outputs and internal actions, using $A_P = A_P^I \cup A_P^O \cup A_P^H$ to represent the set consisting of all actions;

(4) V_P^I, V_P^O, V_P^H is a disjoint set of inputs, outputs and internal variables, using $V_P = V_P^I \cup V_P^O \cup V_P^H$ to

represent the set consisting of all variables;

(5)F_P^V represents the mapping functions, it can map any state in $S_P$to a state pattern in some Object-Z;

(6) F_P^A represents the mapping functions, it can map any state in A_P to an operation pattern in some Object-Z, among the species can be divided into sets F_P^I, F_P^O, F_P^H, respectively mapped to input operation mode, output operation mode, and internal operation mode, etc.,

(7) $\mathcal{T}_P \subseteq V_P \times A_P \times V_p$ is a set of transition actions.

4.2. Rules of Model Transformation

The rules for conversion from AADL-Object-Z to OZIA models are as follows:

(1)S_T is the module of AADL-Object-Z corresponding to the variable set S_P of OZIA. *State guard* module corresponds to the class *Guard* of OZIA;

(2)*initial_state* in the *State* module of AADL-Object-Z corresponding to the initial state set S_P^{init} of OZIA;

(3 D_T is the module of AADL-Object-Z corresponding to the set V_P of OZIA. The Guard in AADL-Object-Zmodule corresponds to φ in OZIA. F_P^V represents the mapping functions, it can map the description $d_i \in D_T, i \in N$ in AADL-Object-Zconcerning constraints on data properties into a set of patterns in OZIA;

(4)P_T is the module of AADL-Object-Z corresponding to the interface of OZIA;

(5) A_T is the module of AADL-Object-Z corresponding to the action set A_P of OZIA. F_P^A represents the mapping functions, it can map the relevant action set $A_P = A_P^I \cup A_P^O \cup A_P^H$ in AADL-Object-Z to the operation mode set, and map it to the input operation mode A_P^I, output operation mode set A_P^O and the internal actions A_P^H in more detail according to the different actions;

(6) TR_T is a module of AADL-Object-Z corresponding to the transition action set $\mathcal{T}_P$ of OZIA.

Table 2. Rules of model transformation

AADL-Object-Zspecification	OZIA specification
all states in the mode	S_P
initial states in the mode	S_P^{init}
inputs, outputs, and internal actions	A_P^I, A_P^O, A_P^H
inputs, outputs, and internal variables	V_P^I, V_P^O, V_P^H
mapping of variables	F_P^V
mapping of actions	F_P^A
state transition relations	$\mathcal{T}_P$

Bismulation [13] is commonly used in modal logic to describe the equivalence between the behavior of two systems. Since future formal verification work is based on formal models, in this paper we transform the semi-formal model of the AADL extension language into the formal automaton OZIA. Bisimulation is defined as follows:

Definition 5 (Bisimulation): Let $M_1 = (S_1^Z, \rightarrow_1)$ and $M_2 = (F_P^A(S_2), \rightarrow_2)$ be two transition systems, $\alpha^z, F_P^A(\beta) \in A_P$. Asymmetric binary relation $R \in S_1^Z \times F_P^A(S_2)$ is called a strong conditional bisimulation relation if ${s_1}^z R f_P^A(s_2)$, implies:

(1)for each ${s_1}^{z\prime} \in S_1^Z$, if ${s_1}^z \xrightarrow{\alpha^z}_1 {s_1}^{z\prime}$ then there exist $s_2' \in F_P^A(S_2)$ such that $f_P^A(s_2) \xrightarrow{F_P^A(\beta)}_2 f_P^A({s_2}')$, and ${s_1}^{z\prime}\, R\, f_P^A({s_2}')$, where $F_P^A(\beta) = \alpha^z \in A_P$; (backward)

(2)for each $f_P^A({s_2}') \in F_P^A(S_2)$, if $f_P^A(s_2) \xrightarrow{\alpha^z}_2 f_P^A({s_2}')$, then there exists ${s_1}^{z\prime} \in S_1^Z$ such that ${s_1}^z \xrightarrow{F_P^A(\beta)}_1 {s_1}^{z\prime}$ and ${s_1}^{z\prime} R\, f_P^A({s_2}')$, where $\alpha^z = F_P^A(\beta) \in A_P$. (forward)

M_1 and M_2 are called strong bisimilar, if there exists a symmetric binary relation R between M_1 and

M_2 such that for all state $s_1{}^z \in S_1^Z$, there exists a state $f_P^A(s_2) \in F_P^A(S_2)$ that satisfies $(s_1{}^{z\prime}, f_P^A(s_2{}')) \in R$ and vice versa.

$s_1{}^z$ denotes the state in the behavioral attachment under the Object-Z mode, $f_P^A(s_2)$ denotes the state in the behavioral attachment under the automaton OZIA model and they satisfy the R relationship, i.e.,$s_1{}^z R f_P^A(s_2)$. $s_1{}^{z\prime}$denotes the state $s_1{}^z$ arrived under the action α^z, $f_P^A(s_2{}')$ denotes the state $F_P^A(\beta)$ arrived under the action $f_P^A(s_2)$, and they still satisfy the R relationship after the transition, i.e., $s_1{}^{z\prime}$ R $f_P^A(s_2{}')$.

Therefore, we can use the automaton OZIA obtained by model transformation to do further work on model checking.

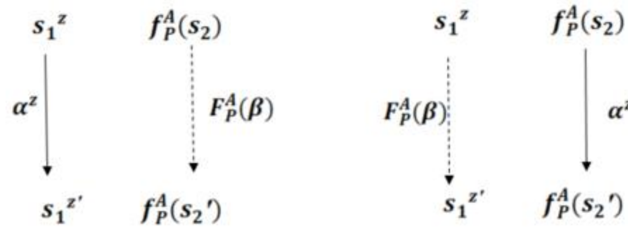


Fig. 3. The equivalence to the strong bisimulation.

5. Aircraft Landing Control System Modeling

In this section, an example of an airport aircraft landing control is used to illustrate the specific process of modeling a real-time system model with data constraints and the formalized interface automaton model after the transformation.

The process of movement in which an aircraft slips from an altitude of 50 feet and lands on the ground until it stops skidding is called landing. Landing is divided into five stages: *drop height*, *flare-out*, *flat floating*, *grounding,* and *taxiing*.

We define that the distance from the airport is more than 400ft to start the decline, and the distance from the ground is less than or equal to 50ft to level off, assuming that the speed of the aircraft is not greater than 700km/h during the decline, not greater than 400km/h during the level off and not greater than 200km/h during the level off. The landing process is schematically shown in Figure 4 below.

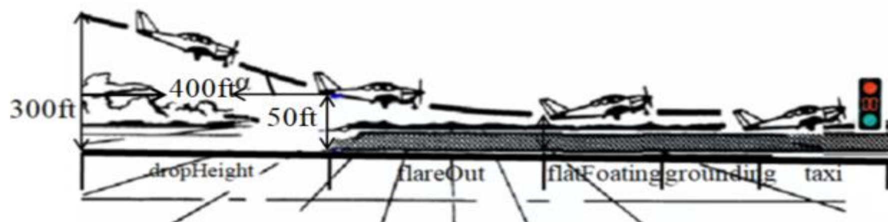


Fig. 4. The aircraft landing control system.

5.1. The AADL-Object-Z Model of System

The main function of the Aircraft Landing Process (ALP) is to obtain real-time altitude and velocity information of the aircraft and to compute the next flight state. The ALP is described by an AADL modeling specification with data constraints and is described by a threaded building block alp. The runway state is determined by the flight state whether it is idle or not: when the runway beacon receives the signal from the tower, the beacon is red and the aircraft flies off the runway, the beacon is in the green state. The

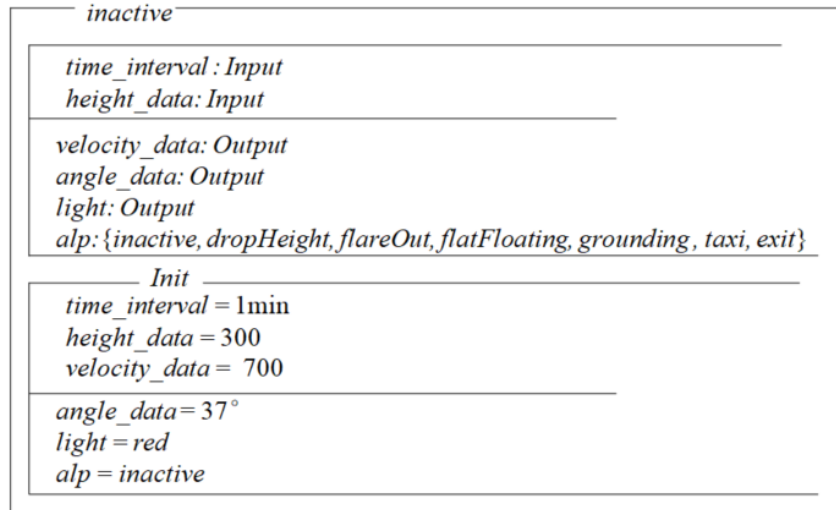
modeling process for it is shown in Table 3.

Table 3. The aircraft landing process modeled by AADL-Object-Z

```
package example
public
thread alp
    features
    height_data: in data port;
    velocity_data: in data port;
    angle_data: out data port;
    time_interval: in event port;
    light: out event port;
    properties
Dispatch_Protocol ⇒ Periodic;
Compute_exection_time ⇒ 5 ms ... 15ms;
Period ⇒ 20 ms;
end alp;
thread implementation alp. impl
  annex behavior _specification {**
    Variable:
      v:Base_Types::Unsigned_32;
      h:Base_Types::Unsigned_32;
      α:Base_Types::Unsigned_32;
      l:red or green;
    State:
      inactivity: initial_state;
      dropHeight, flareOut, flatFloating, grounding, taxiing, exit: states;
State guard:
      dropHeight: v<=700&&h>20&&l=red;
      flareOut : v<=400&&h<=20&&l=green;
      flatFloating:v<=200&&h=0&&l=green;
      grounding:v<=200&&h=0&&l=green;
      taxi:v<=100&&h=0&&l=green;
      exit :v<=50&& h=0&&l=red;
    transitions:
  inactivity -[ time_interval<=1min] → dropHeight{  v<=700&&h>20&&l=red}
  dropHeight -[ time_interval>1min&& time_interval<=2min] → flareOut { v<=400&& h<=20&&l=green}
  flareOut -[time_interval<=1min] → flatFloating { v<=200&&  h=0&&l=green};
  flatFloating -[time_interval<=1min] → grounding {v<=200&& h=0&&l=green};
  grounding -[time_interval<=1min] →exit {v<=100&&  h=0&&l=green};
  exit -[light=green] → inacitvity;
  **}
  end alp. impl;
end example;
```

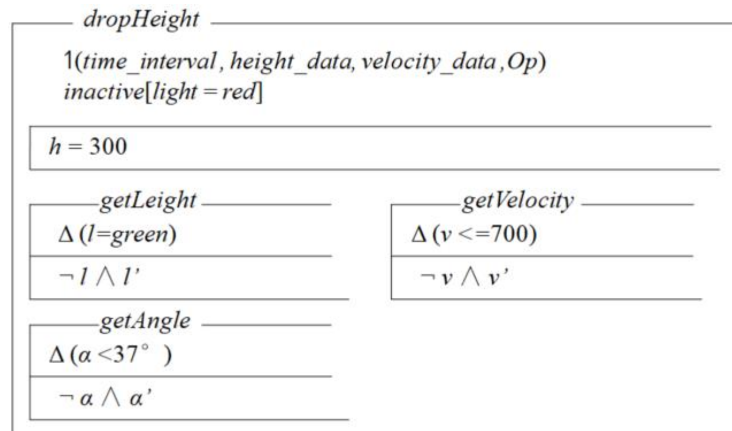
The above alp thread artifacts are described formally in the Object-Z language for their variable constraints and variants as follows:

state *inactive*:



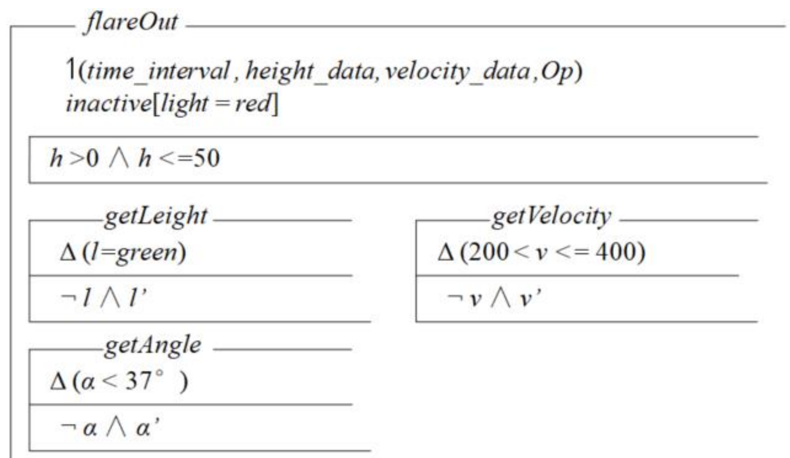
The main purpose of *inactive* is to create the conditions for landing. The control tower sends out the signal and the runway signal light receive the signal from the control tower, then the signal light changes from the original green state (indicating the runway is in a free state) to the red state (indicating the runway is in a busy state).

state *dropHeight*:



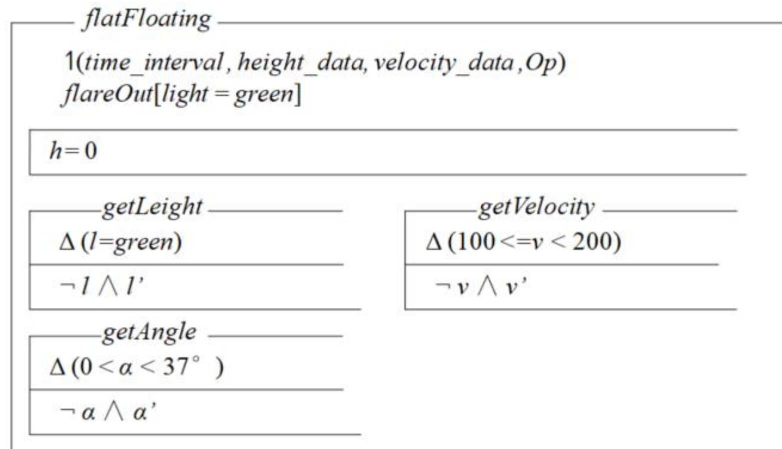
When descending to the specified altitude, the pull rod increases the angle of approach, increases the lift, and decreases the angle of descent.

state *flareOut*:



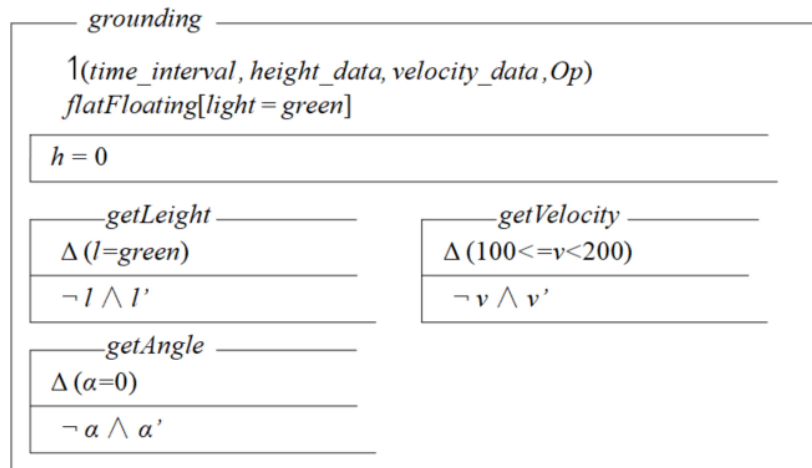
As the angle increases, the drag of the aircraft increases, the aircraft deceleration, and the altitude decrease.

state *flatFloating*:



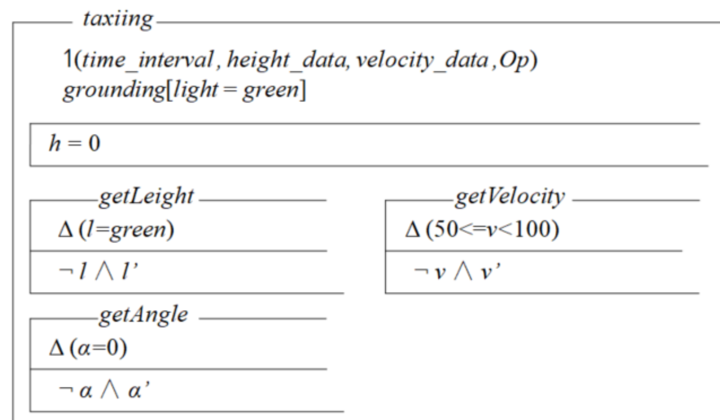
In the *flatFloating* stage, the aircraft speed decreases gradually due to the larger angle of welcome and the larger drag.

state *grounding*:



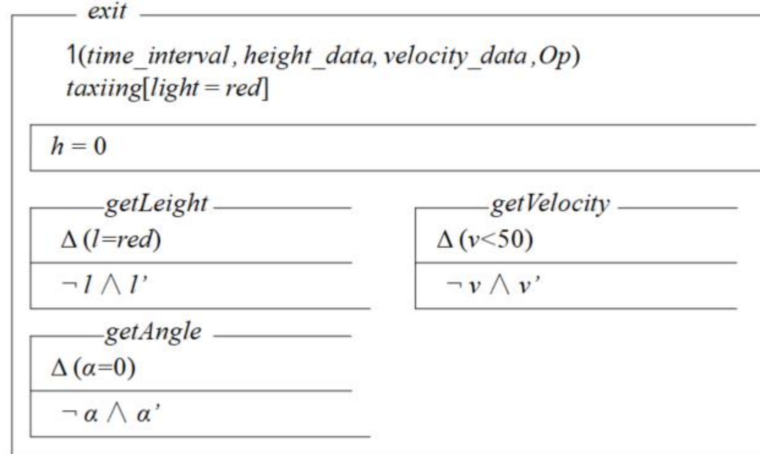
In the *grounding* stage,the lift is slightly less than the gravity, and the aircraft sinks slowly. At this time,due to the increase of aircraft angle and ground effect, the nose of the aircraft automatically drops.

state *taxiing*:



At this time, due to the increase of aircraft angle and ground effect, the nose of the aircraft automatically drops. Therefore, it is necessary to bring the rod backward appropriately with the sinking of the aircraft to maintain the grounding attitude and lift.

state *exit*:



The runway signal receives this exit signal and changes from red (indicating that the runway is busy) to green (indicating that the runway is idle).

5.2. Transformation to OZIA

Denote by *i*, *d*, *f*, *o*, *ff*, *t*, *e* to describe the state of aircraft *inactive*, *dropHeight*, *flareOut*, *flatFloating*, *grounding*, *taxiing,* and *exit*. Through the rules of model transformation in section 3.1 of this paper, we can obtain the OZIA automaton of the entire aircraft landing airport as shown in Figure 5.

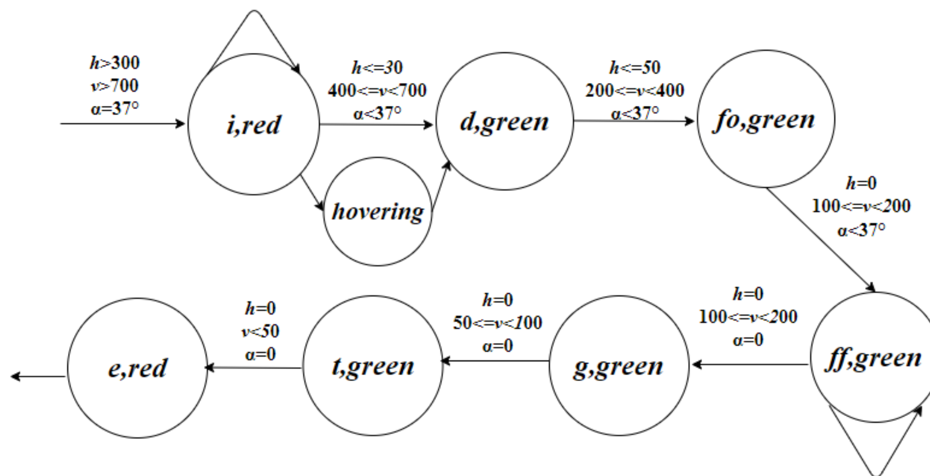


Fig. 5. The OZIA model of the aircraft landing system.

6. Formal Verification

Formal verification methods are mainly divided into two categories: one is deductive verification based on logical reasoning; the other is model detection based on exhaustive state space search. The shortcoming of the former is that it cannot be fully automated verification, and for slightly complex systems, automated reasoning is difficult, so it is only applicable to smaller systems and difficult to be accepted by the industry. Model checking is an automatic verification technique that detects whether the system behavior has the expected properties by

searching the yet poor state space of the system model. It has the advantage of being able to automate validation, thanks in large part to the support of automated validation tools, such as SPIN, SMV, PAT, etc. These tools accept the LTL formula, the CTL formula, and the process algebra, respectively. In the next step, we expect to achieve formal verification with these tools.

7. Conclusion

The model is essentially a collection of variables and transfer functions, so its expressive ability is naturally reflected in these two aspects. Why do we take compositional modeling? The reason is that the description of data constraints is not sufficient in AADL, especially in the object-oriented domain. Object-Z is specially used to describe object-oriented data constraints and state transition, so we put forward a composite system modeling based on Object-Z technology to strengthen this aspect.

To summarize, this paper proposes the formal specification language AADL-Object-Z by combining the Object-Z specification language and AADL Behavior Annex. The paper proposes an OZIA automaton as a formal model and then gives the rules of transformation between the two models. And we give a practical example of the Aircraft Landing Process case study to describe the whole process from modeling to Object-Z extension to OZIA. There are many more mature model detection tools, such as NuSMV, UPPLA, and PAT, which support real-time hybrid systems. The next work will be to do specific formal verification and analysis based on the existing formal model in this paper.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Zining Cao conducted the research; Zhengling Guo proposed the method and wrote the paper; all authors approved the final version.

Funding

This work was supported in part by the Aviation Science Foundation of China under Grant 20185152035.

References

[1] Chen, X., Zhu, Y., Zhao, Y., Wang, J., & Altynbek, A. (2021). Hybrid modeling and model transformation of AADL for verifying the properties of CPS space-time compositions. *IEEE Access*, *9*, 99539–99551.

[2] Xu, J., Yang, Z., Huang, Z., Zhou, Y., Liu, C., Xue, L., & Filali, M. (2018, October). Hierarchical behavior annex: Towards an AADL functional specification extension. *Proceedings of the 2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design* (pp. 1–11). IEEE.

[3] Lee, J., Bae, K., Ölveczky, P. C., Kim, S., & Kang, M. (2022). Modeling and formal analysis of virtually synchronous cyber-physical systems in AADL. *International Journal on Software Tools for Technology Transfer*, 1–38.

[4] Smith, G. (2012). *The Object-Z Specification Language* (Vol. 1). Springer Science and Business Media.

[5] Zhu, Y., Cao, Z., Wang, F., & Lu, W. (2020, November). AADL and modelica model combination and model conversion based on CPS. *Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering* (pp. 1136–1140).

[6] Sokolsky, O., Lee, I., & Clarke, D. (2009). Process-algebraic interpretation of AADL models. In 2009: *Proceedings of the 14th Ada-Europe International Conference Reliable Software Technologies–Ada-Europe* (pp. 222–236). Springer Berlin Heidelberg.

[7] Mkaouar, H., Zalila, B., Hugues, J., & Jmaiel, M. (2020). A formal approach to AADL model-based

software engineering. *International Journal on Software Tools for Technology Transfer*, *22*, 219–247.

[8] Aerospace, S A E. (2011). SAE architecture analysis and design language (AADL) annex volume 2. *Annex B: Data Modeling Annex Annex D: Behavior Model Annex Annex F: ARINC653 Annex*.

[9] Stepney, S., Barden, R., & Cooper, D. (2013). *Object orientation in Z.* Springer Science and Business Media.

[10] Derrick, J., Boiten, E. A., Derrick, J., & Boiten, E. A. (2014). Combining CSP and object-Z. refinement in z and object-Z. *Foundations and Advanced Applications*, 431–455.

[11] Smith, G., & Duke, D. J. (2020). Specification with class: A brief history of object-Z. In formal methods. *Proceedings of the 2019 International Workshops*.

[12] Zhang, Y., Shi, J., Zhang, T., Liu, X., & Qian, Z. (2015). Modeling and checking for cyber–physical system based on hybrid interface automata. *Pervasive and Mobile Computing*, *24*, 179–193.

[13] Milner, R., & Sangiorgi, D. (1992). Barbed bisimulation. In Automata, Languages and Programming: *Proceedings of the 19th International Colloquium Wien*.

Copyright © 2023 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0)