Received September 13, 2017, accepted October 30, 2017, date of publication November 8, 2017,
date of current version December 22, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2770323

Formal Verification of Behavioral AADL Models by Stateful Timed CSP

FENG ZHANG[1], YONGWANG ZHAO[1,2], DIANFU MA[1], AND WENSHENG NIU[3]
[1]State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang Univerisity, Beijing 100191, China
[2]Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing 100191, China
[3]Aeronautical Computing Technique Research Institute, Xi'an 71000, China

Corresponding author: Yongwang Zhao (zhaoyw@buaa.edu.cn)

This work was supported by the Project of State Key Laboratory of Software Development Environment under Grant SKLSDE- 2017ZX-13.

ABSTRACT AADL along with its Behavior Annex is an architecture and behavior description language for safety-critical domains, e.g. avionics, aerospace, and defence. In order to formally analyze behavior properties of AADL models, it is necessary to transform the AADL language into formal languages supported by formal verification tools. Moreover, comprehensive formal verification of AADL models highly requires that the transformation supports larger subset of the AADL language, as well as verification tools are able to capture various behavior of AADL, such as concurrency and timing. As an extended communicating sequential process (CSP), stateful timed CSP with its model checker-PAT provide an strongly expressive language and verification tool for real-time systems, distributed systems, and concurrent systems. This paper introduces a model transformation approach from a comparatively complete subset of AADL to stateful timed CSP, in particular supporting major components of AADL Behavior Annex. We propose a comprehensive set of transformation rules for AADL to stateful timed CSP. Then, we perform formal verification in PAT to analyze concurrent behavior properties of AADL models, such as safety, liveness, and trace refinement with various fairness assumptions, in which we consider time capacities, deadlines, periods of AADL threads and durations of AADL processes. As a study case, we develop an AADL model of F-16 "Auto Pilot Controller" and transform the model into Stateful Timed CSP. We specify a set of critical properties of the model and perform formal verification in PAT.

INDEX TERMS Model transformation, AADL, behavior annex, stateful timed CSP, PAT, formal model-checking.

I. INTRODUCTION

AADL (SAE Architecture Analysis & Design Language [1]) is a textual and graphical Modelling language used to design and analyze the software and hardware architecture of safety-critical real-time systems. AADL along with its Behavior Annex [2] describes not only the architectural aspects of models, such as the dataflow and control flow, but also a presentation for behavioral characteristics.

Formal verification can ensure safety, liveness and other properties of AADL models. Formal methods enable complete exploration of state space of a formal model with respect to certain properties. Communicating Sequential Processes [3] (CSP) as a formal theory of process algebra can formally specify concurrent behaviors of communication systems. Stateful Timed CSP [4] (STCSP) as an extension of Timed CSP [5] with mutual data operations and a rich set of timed process constructs is able to capture timed system behavior. Process Analysis Toolkit (PAT [6], [7]) as a model checker of STCSP provides cost-effective analysis capabilities as well as more powerful functionality and more efficient performance over other model checkers.

In order to apply formal verification on AADL models using PAT, it needs the model transformation from AADL to STCSP. There already exist studies focusing on transformation from AADL to specification languages of model checkers, such as CSP [8]–[10] and CSP-like formal languages [11]. Other studies transform AADL to Fiacre [12], [13], Maude [14], [15] and TASM [16], [17]. These works focus on a small subset of the AADL language and perform formal verification of deadlock free, liveness, etc. To formally verify realistic safety-critical systems, a larger subset, in particular covering the whole set of AADL

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Behavior Annex, is highly required. Moreover, formal verification of other critical properties, such as divergency and trace refinement, can significantly improve assurance of the systems.

In this paper, we provide an approach for formal verification of behaviorial AADL models by transformation into PAT. We consider a large subset of AADL focusing on safety-critical software. The subset covers software components, communication among components and Behavior Annex defined in the AADL language. We define a comprehensive transformation of the subset into STCSP with the transformation of Behavior Annex with complex state transitions. In detail, this paper makes the following contributions:

1) We define a large subset of AADL focusing on modeling software systems, and propose a comprehensive set of transformation rules into STCSP. Compared to existing studies, our subset and transformation rules support major components of Behavior Annex including variable definitions, all types of state definitions, state transitions with guard conditions and actions.
2) We develop an AADL model of F-16 ''Auto Pilot Controller'' (F-16 APC) [18] which is constructed according to aircraft aerodynamics data from NASA. By refering to the AADL ARINC653 Annex, the AADL model and its behavior state machine are targeted for ARINC653 operating systems.
3) We perform formal verification of the transformed model, i.e. STCSP models, in the PAT model checker. We specify the critical properties of ''F-16 APC'' including a set of safety, liveness, and an abstraction specification for refinement checking.

The rest of the paper is organized as follows. Section II-A describes the concept of AADL along with its Behavior Annex and ARINC653 Annex, Section II-B presents the strengths of PAT and its specification language STCSP to justify why we select STCSP to model AADL and PAT to perform formal analysis; Section II-C introduces the related works. Section III overviews our approach; Section IV presents model-transformation rules and their demonstration examples; Section V presents a case study, and Section VI gives the conclusions and future directions.

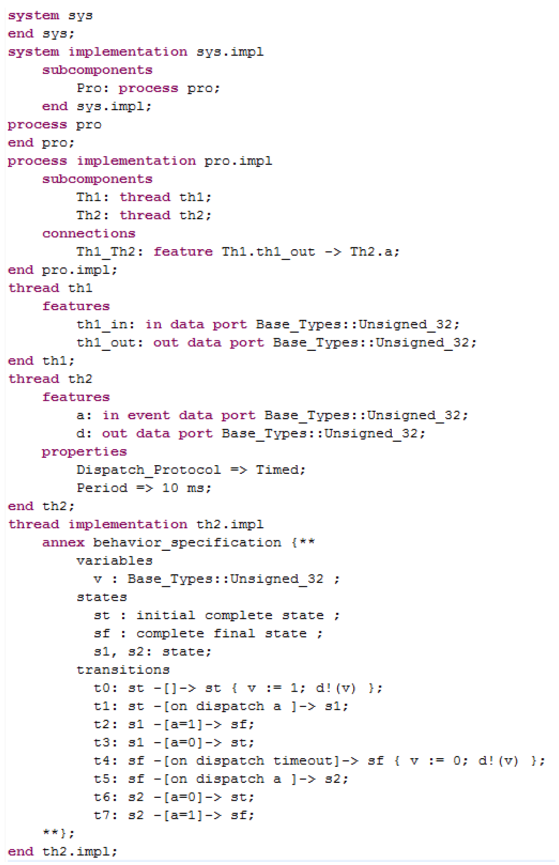
II. BACKGROUND

A. AADL, BEHAVIOR ANNEX AND ARINC653 ANNEX

SAE Architecture Analysis & Design Language is a textual and graphical language to design and analyze safety-critical real-time systems, whose operation strongly depends on meeting safety system requirements. AADL is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. The AADL Behavior Annex associates automata to define dynamic execution semantics of AADL. The evoluation of states in behavior specification is specified by state transitions. The detailed information of AADL and Behavior Annex will be presented in Subsection III-A.

AADL ARINC653 Annex [19] is defined to support the modeling, analysis and automated integration of ARINC653 [20] partitioned architectures. It provides AADL architectural style guidelines and AADL defined ARINC653 oriented properties to define a common approach to use AADL standardized components to express ARINC653 architectures.

A simple example of AADL with behavior annex is illustrated in Figure 1, in which there is a system *sys* comprising a process *pro*. Process *pro* comprise two threads *th1* and *th2* and between features of the two threads there is a connection *Th1_Th2*. Thread *th2* comprise a behavior specification which is borrowed from AADL Behavior Annex.



```
system sys
end sys;
system implementation sys.impl
    subcomponents
        Pro: process pro;
    end sys.impl;
process pro
end pro;
process implementation pro.impl
    subcomponents
        Th1: thread th1;
        Th2: thread th2;
    connections
        Th1_Th2: feature Th1.th1_out -> Th2.a;
end pro.impl;
thread th1
    features
        th1_in: in data port Base_Types::Unsigned_32;
        th1_out: out data port Base_Types::Unsigned_32;
end th1;
thread th2
    features
        a: in event data port Base_Types::Unsigned_32;
        d: out data port Base_Types::Unsigned_32;
    properties
        Dispatch_Protocol => Timed;
        Period => 10 ms;
end th2;
thread implementation th2.impl
    annex behavior_specification {**
        variables
          v : Base_Types::Unsigned_32 ;
        states
          st : initial complete state ;
          sf : complete final state ;
          s1, s2: state;
        transitions
          t0: st -[]-> st { v := 1; d!(v) };
          t1: st -[on dispatch a ]-> s1;
          t2: s1 -[a=1]-> sf;
          t3: s1 -[a=0]-> st;
          t4: sf -[on dispatch timeout]-> sf { v := 0; d!(v) };
          t5: sf -[on dispatch a ]-> s2;
          t6: s2 -[a=0]-> st;
          t7: s2 -[a=1]-> sf;
    **};
end th2.impl;
```

FIGURE 1. A simple AADL example.

A state-charts presentation of automata can be described as a automata diagram. For example in Figure 1, the behavior specification in thread th2 can be illustrated in Figure 2.

B. STATEFUL TIMED CSP AND PROCESS ANALYSIS TOOLKIT (PAT)

CSP is a formal language for describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi, based on message passing via channels. CSP was first described in a 1977 paper by Tony Hoare, but has since evolved substantially. CSP has been

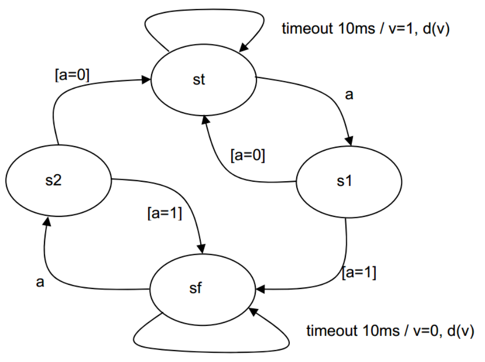


FIGURE 2. **The diagram presentation for a behavior specification.**

practically applied in industry for specifying and verifying the concurrent aspects of various different systems.

STCSP is a formal modeling language extending CSP [3]. STCSP is able to specifying real-time, concurrent systems. STCSP comprises elements of constants/variables, channels, or/and processes. A process is defined as $P(i_1, i_2, \dots, i_n) = PExp$, where P is the process name, $i_1, i_2, \dots, i_n$ is an optional list of process parameters and $PExp$ is a process expression. A Process P models the control logic of the system using a rich set of process constructs.

$P ::= Stop \mid Skip$	- primitives
$\mid e \rightarrow P$	- event prefixing
$\mid [precoditions]\ P$	- process guard
$\mid e\{program\} \rightarrow P$	- process actions
$\mid ch!x \rightarrow P \mid ch?x \rightarrow P$	- channel output/input
$\mid P\ []\ Q \mid P\ <>\ Q$	- external/internal choices
$\mid P \parallel Q \mid P \mid\mid\mid Q$	- parallel/interleaving
$\mid P; P$	- sequential composition
$\mid P \setminus events$	- hiding
$\mid if\ (b)\ \{P\}\ else\ \{Q\}$	- conditional choice
$\mid while(cond)\ \{P\}$	- while program
$\mid Wait[d];\ P$	- delay
$\mid P\ timeout[t]\ Q$	- timeout
$\mid P\ interrupt[t]\ Q$	- timed interrupt
$\mid P\ within[t]$	- within
$\mid P\ deadline[t]$	- deadline
$\mid Q$	- process referencing
$program ::= x = exp$	- assignment
$\mid program;\ program$	- composition
$\mid if\ (b)\ \{program\}\ else\ \{program\}$	- conditional
$\mid while\ (b)\ \{program\}$	- iteration

Stop is a deadlock process which does absolutely nothing and allows time elapsing. *Skip* is a process which terminates possibly after delaying for a while, and then behaves exactly the same as *Stop*. The event-prefixed process $e \rightarrow P$ engages in an event e first and then behaves as P. It has to be noted that the event e may serve as a synchronization barrier, if event-prefixed processes are combined with parallel composition. $e\{program\} \rightarrow P$ engages in event e whilst executing the sequential *program*, which may be a simple procedure updating data variables(e.g. in the form of $ax = 1;\ y = 2$) or a complicated sequential program. $P\ []\ Q$ is general choice which states that either P or Q may execute if P performs an event first, then P takes control; otherwise, Q takes control. $P <> Q$ is internal choice which states that either P or Q may execute and the choice is made internally and non-deterministically and immeidately. ***if***$(b)\{P\}$ ***else***$\{Q\}$ is a conditional choice and ***while***$(cond)\{P\}$ is a while program. $P;Q$ is a sequential composition which states that P starts first and Q starts immediately, without any delay, when P has finished. $P\backslash events$ Process is hiding operation which turns *events* to invisible ones.

Wait, timeout, interrupt, within, deadline are time related operators. *Wait[t]* delays the system execution for a period of t time units then terminates, therefore *Wait[t];P* delays the starting time of P by exactly t time units. P *timeout[t]* Q passes control to process Q if no event has occurred in process P before t time units have elapsed. P *interrupt[t]* behaves as P until t time units elapse and then switches to Q. P *deadline[t]* is constrained to terminate within t time units. P *within[t]* states that the first visible event of P must be engaged within t time units.

The choice of STCSP is justified as follows. The state of art approaches for modeling and verifying complex real-time systems are mostly based on Timed Automata which are finite state automata equipped with clock variables. Timed Automata are deficient in modeling hierarchical complex systems due to the widespread applications and increasing complexity. STCSP based on Timed CSP is capable of specifying hierarchical real-time systems. Finite-state zone graphs can be generated automatically from STCSP models by dynamic zone abstraction. Model checking with non-Zenoness in STCSP can be achieved based on the zone graphs using PAT. The model checker PAT can support system modeling and verification using STCSP and show its usability/scalability via verification of real-world systems.

PAT is a self-contained framework to support simulating and reasoning on concurrent, real-time systems and other critical domains, which can use BDD and digitization [21], implicit clocks and zone Abstraction [22], clock-symmetry reduction [23] for symbolic model-checking of STCSP models. PAT implements various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. Shi [24] presents a comprehensive comparison of CSP extensions, and their supporting tools FDR, ProB and PAT.

The choice of PAT is justified by several features as follows: (i) **Easy Extensibility**. PAT's architecture provides extensibility [25] in many possible aspects: modeling languages, model checking algorithms, and reduction techniques. Various model checkers have been integrated into PAT under this new architecture. (ii) **Multiple modeling methods**. It supports modeling traditional CSP,

TABLE 1. Comparisons of related works.

Works	Specification Languages	Verification Tools	AADL Elements Considered			Safety		LTL/CTL	Trace Refinement
			Connections	Behavior Annex	Components	Deadlock	Divergency		
Yang C et al. [8]	CSP	FDR	++++	-	+++	✓	✓	-	-
Ahmad E et al. [9] [10]	Hybrid CSP	FDR&BLESS	+++	++	++++	✓	-	-	-
Mkaouar H et al. [11]	LNT	CADP	++++	-	++++	✓	-	✓	-
Berthomieu B et al. [12]	Fiacre	Tina	+++	+	++++	✓	-	✓	-
Olveczky P C et al. [14] [15]	Maude	Maude	+++	+++	++++	✓	-	✓	-
Yang Z et al. [16], Hu K et al. [17]	Timed Abstract State Machines	UPPAAL	+++	++	++++	✓	-	✓	-
Bao Y et al. [32]	Networks of Priced Timed Automata	UPPAAL-SMC	+++	++	++++	✓	-	✓	-
Johnsen A et al. [33], Kim J H et al. [34]	UPPAAL Timed Automata	UPPAAL-SMC	++++	++	++++	✓	-	✓	-
Our Work	**Stateful Timed CSP**	**PAT**	++++	+++	++++	✓	✓	✓	✓

probability CSP, real-time CSP, probabilistic real-time CSP, labeled transition and timed automata in various domains. (iii) **More properties** able to be analysed, such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic properties. (iv) **State of art model-checking algorithms**: linearizability [26] via optimized refinement checking, explicit-state non-zenoness checking [27], partial order reduction with abstractions [28], improved BDD-based discrete analysis [29] and so on. (v) It provides **user friendly editing environment** (multi-document, multi-language, and advanced syntax editing features) for introducing models and **user friendly simulator** for interactively and visually simulating system behaviors.

C. RELATED WORK

We survey the state of art studies on model transformation of AADL for formal verification.

1) CSP

Yang et al. [8] propose an approach that uses machine-readable CSP to build formal semantics of AADL models, and then performs formal verification using FDR to analysis deadlock and livelock properties. Ahmad et al. [9], [10] use a Hybrid CSP to present formal semantics of a synchronous subset of AADL models.

2) LNT

Mkaouar et al. [11] transform AADL models to LNT [30] specification language which is similar to CSP. LNT is supported by the CADP toolbox which offers a rich formal verification like simulation and model checking.

3) FIACRE

Berthomieu et al. [12] introduce a high-level view of transformation principles and a implementation tool for behavioral verification of AADL models with behavior specifications that take the realtime semantics into account . Abid et al. [13] propose a set of specification patterns which can be used to express real-time requirements commonly found in the AADL design of reactive systems, and provide an integrated model checking tool chain in which Fiacre is used to specify behaviors of AADL systems while verification activities ultimately relies on Tina.

4) MAUDE

Ölveczky et al. [14], [15] provide a formal object-based real-time concurrent semantics using rewriting logic for AADL behavioral models, and the semantics model can be directly executable in Real-Time Maude. They develop a plug-in component integrated into OSATE, which can automatically transform AADL models into corresponding Real-Time Maude specifications.

5) UPPAAL

Bao et al. [31] extend the syntax and semantics of Hybrid AADL, propose a set of mapping rules transforming uncertain-aware Hybrid AADL designs into NPTA, and implement a tool chain that integrates both UPPAAL-SMC and OSATE. Johnsen et al. [32] use the methodology of semantic anchoring, which is through a set of transformation rules to timed automata constructs, to contribute with a formal and implemented semantics of a subset of AADL. Kim et al. [33] present a method for formal analysis of AADL models using UPPAAL. Yang et al. [16] present a machine checked, semantics-preserving transformation rules from AADL to Timed Abstract State Machines (TASM), and verify transformation rules in the theorem prover Coq. Hu et al. [17] present a methodology for translating AADL to TASM and implement a tool using Atlas Transformation Language (ATL).

The comparisons of the above related works are listed in Table 1. These works focus on a subset of the AADL language. For connections in AADL, our work considers four types of connections, i.e. three types of port connection (data/event/event data) and parameter connection, each of which is represented by a "+" in the table. Some related works do not consider parameter connection. For AADL Behavior Annex, our work supports variable definitions, all types of state definitions (*initial* | *complete* | *return* | *final*), and state transitions with guard conditions and actions, in which *variables*, *states*, *state transitions* are represent by a "+" respectively. Most of related works only consider a small subset of Behavior Annex. For aspect of verified properties, our work considers more types of properties by STCSP and PAT, e.g. deadlock free, divergence, LTL/CTL formula, and trace refinement.

III. APPROACH OVERVIEW

The general methodology of our work is: first, we select a comparatively complete subset of AADL, determine the policy of behavioral description of AADL models; Second, we present the methodology of behavior-based model transformation from the subset of AADL to STCSP; Third, we perform model-checking of the transformed STCSP models to analysis the properties in AADL models using PAT. The main idea of our methodology is illustrated in Figure 3.

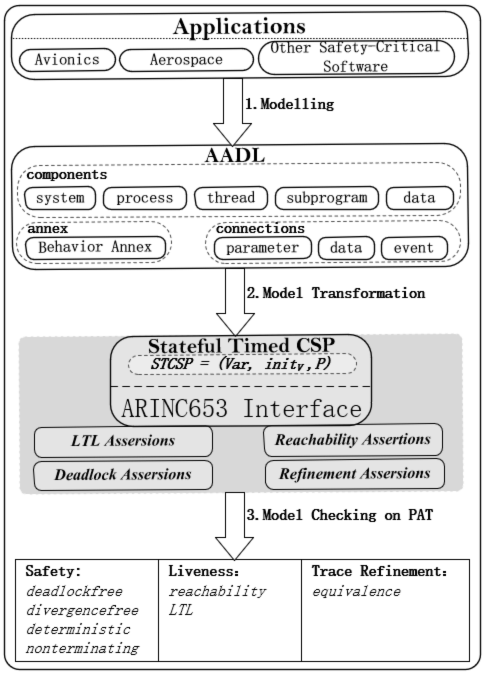


FIGURE 3. Approach overview.

A. SELECTION OF AADL SUBSET

We only focuses on the description and analysis of the software portion of embedded systems, so we do not consider the execution platform elements of AADL: *processors*, *virtual processors*, *memory*, *buses*, *virtual buses* and *devices*. For the software elements of AADL, the *group*, *prototype* and *refinement* are mainly for the reusability of AADL code, we can modeling software systems even without these three elements, therefore we do not consider them in our first version of works. Thus, this paper focuses on software components and its behavior specification. This subset of AADL elements is enough to model embedded software in most application cases.

The software components comprise *system, process, thread, subprogram* and *data*. A *subprogram* component represents sequentially executed source text that is called by threads with parameter transmission, a *thread* representing a sequential flow of executed instructions models a concurrent schedulable unit that can execute concurrently with other threads in a process, a *process* represents a virtual address space which is a space partition unit whose boundaries are enforced at runtime, and a software *system* represents an assembly of interacting application software.

Features specify how a component interfaces with other components. Features comprise *port* features and *parameter* features. A connection is a linkage between features of two components representing communication of data and control between components. In addition, features can be combined with *properties*.

A behavior specification of a component described using Behavior Annex contains *state variables*, *states*, *state transitions*. *States* may be *initial* | *complete* | *return* | *final*. State transitions can be guarded by a binary expression and linked with a action sequence.

B. SUMMARY OF MODEL TRANSFORMATION RULES

The summary of model transformation rules from AADL to STCSP is described as follows.

Transform connections and features. Transform AADL connections to STCSP communication events; Map AADL features as connection ends, i.e. ports and parameters, to STCSP variables representing data transferred during communication actions (see Section IV-A);

Transform behavior specifications. The transformation of an AADL behavior specification comprises some sophisticated procedures (see Section IV-B).

- Map variables in a behavior specification to STCSP variables (see Section IV-B - Tr_BS1);
- Define STCSP variables corresponding to *initial*, *complete*, and *return* states to denote the current state a transition automaton currently runs at(see Section IV-B - Tr_BS2);
- Transform guards and actions in a behavior specification to guards and actions in STCSP processes (see Section IV-B.1 and IV-B.2);
- Transform transitions in a behavior specification to STCSP processes representing behaviors of these transitions (see Section IV-B.3);
- Assemble STCSP processes representing AADL transitions to one compositional STCSP processes which comprise all the state transitions of a behavior specification (see Section IV-B.3).

Transform components. Transform AADL components to STCSP processes, moreover the transformed target STCSP processes maintain the hierarchy of the source AADL models (see Section IV-C).

IV. TRANSFORMATION OF AADL INTO STCSP

The basic transformation rules are listed in Figure 4. We describe our transformation rules using tuples which abstract the concrete AADL model code. We present transformation rules for AADL features, connections, behavior specifications and software components (i.e. subprograms, threads, processes and systems) in detail as follows.

AADL			STCSP	Rules	Comments
features	ports	data/data event ports	**var portId;**	**tr_feature(4.1)**	map an AADL data/event data port name to a STCSP variable name transmitted on these port connections.
		event ports	--		map an AADL event connection name to a STCSP event name.
	cut parameters		**var parameterId;**		map a out parameter of a subprogram to a STCSP variable name representing parameters transmitted out of this subprogram when calling this subprogram.
connections			**eventId <- connectionId;**	**tr_connection (4.1)**	map AADL port/parameter connections to STCSP communication events.
Behavioral Specification	variables		**var variableId;**	**tr_variable (4.2-Tr_BS1)**	transform state variables in a Behavior Annex to STCSP variables.
	initial, complete, return states		**define stateId;**	**define variables (4.2-Tr_BS2)**	define STCSP variables corresponding to initial, complete and return states to denote the current state a transition automaton run at.
	guards		guards in STCSP events	**tr_guard(4.2.1)**	the transformed STCSP processes should present the semantics that: a transition to a return state ends a subprogram and a transition to complete state completes a thread
	actions		action sequences	**tr_action(4.2.2)**	
	transition	destination state is intermediate	A STCSP processe ended with a STCSP processe representing the transition started with the current destination state.	**tr_transitions (4.2.3)**	
		destination state is *complete/return*	A STCSP processe suspends with a **Skip**; Set the destinational *complete* state.		
		destination state is *final*	A STCSP process stop with a **Skip**; Reset all the *complete* states.		
subprogram **subp**			**subp**(paraIn1, paraIn2, ···, paraInn) **= BehavioralAnnex();**	**tr_subp(4.3.1)**	
thread **thr**			**thr**() = InputEvents -> **BehavioralAnnex**();	**tr_thread(4.3.2)**	
process **pro**			**pro**() − InputEvents -> (**periodic_thr() || thr2() ||** ···**|| thrn()**);	**tr_pro(4.3.3)**	periodic_thr is a periodic thread and other threads th2··· thn are aperiodic threads;
system **s**			**s()** = **pro1()** interrupt[dur1] **pro2()** interrupt[dur1+dur2] ··· **pro_n-1()** interrupt[dur1+dur2+ ··· +durn-1] **pron()** interrupt[dur1+dur2+ ··· +durn] **s()**;	**tr_sys(4.3.4)**	pro1···pron are processes in system s; dur1···durn are time duration of processes;

FIGURE 4. **Basic transformation rules.**

A. TRANSFORM AADL I/O CONNECTIONS TO STCSP COMMUNICATION EVENTS

An AADL connection is a linkage between features of two components which represents the communication of data and/or control between components. Features as connection ends define connection contents and connection types. An AADL connection we consider may be a port connection or a parameter connection. Port connections comprise three types: **Rewritable** data transmissions (i.e. sampling connections, blackboard connections in ARINC653), **Queued/Buffered** data transmissions (i.e. queued data/event data connections, buffered data connections in ARINC653) and **event** transmissions which present control transmissions. **Parameter** connections represent parameter transmission between a called subprogram and a calling thread occurs.

Rule tr_feature maps an AADL *data/event data port* name to the name of a STCSP variable representing data transmitted, and an *out* parameter of a subprogram to the name of a STCSP variable representing parameters transmitted out of this subprogram when this subprogram is called.

An AADL feature is defined as a tuple:
feature = (*feaType, feaId, feaDirec, feaProp*).

- *feaType* is type of transmitted elements on this feature: *data/ event/ data event port*, or *parameter*;
- *feaId* is the ID of this feature;
- *feaDirec* is transmission direction on this feature: *in*, *out* or *in out*;
- *feaProp* presents communication attributes: rewritable data transmissions, e.g. sampling or blackboards; queued/buffered data transmissions, e.g. queueing, buffers; or event communications.

Rule tr_connection transforms AADL connections to STCSP communication events.

An AADL port connection is defined as a tuple:
port_conx = (*conxId, srcPortId, conx_symbol, destPortId*).

- *ConxId* is ID of this connection;
- *SrcPortId* and *destPortId* are ID of the source port and ID of the destination port respectively;
- *Conx_symbol* is connection direction: "–>" or "<–>."

An AADL parameter connection is defined as a tuple:
para_conx = (*conxId, srcId, DirectionalSymbol, destId*).

- *ConxId* is ID of this connection;
- *SrcId* and *destId* are ID of the source and ID of the destination respectively;
- *DirectionalSymbol* is transmission direction "–>."

Rule tr_connection is listed in Algorithm 1 in which *CreateChannel(chId)* is a function creating a STCSP channel with ID *chId*.

We describe the transformation for data/event port connections and parameter connections in detail as follows.

1) Port Connections

(1) **Rewritable data transmissions, e.g. sampling/ blackboards, are transformed to STCSP read/write events.**

In Figure 5, the connection *Blackboard_CnR* between two threads presents a directional **Blackboard** data connection from port *th1.DataROut* to port *th2.DataRIn*. The port ends of

Algorithm 1 tr_connection() – Transform a Connection to a STCSP I/O Communication Event

```
Input: conx = (port_conxs, para_conxs);
        port_conx = (conxId, srcPortId, conx_symbol, destPortId);
        para_conx = (conxId, srcId, DirectionalSymbol, destId).
Output: Communication channel definition: Channel chId chBuf;
        Event name: eventId.
1:  For port_conx:
2:  ConxType is determined by portType(srcPortId & destPortId);
3:  if ConxType is queueing or buffer then
4:      if conx_symbol is "–>" then
5:          chId ← conxId+"R";
6:          CreateChannel(chId);
7:          - -When occurring data transmissions on this channel:
8:          - -For srcPort: chId!srcPortId;
9:          - -For destPort: chId?x{destPortId=x;};
10:     else if conx_symbol is "<–>" then
11:         chId ← conxId+"R";
12:         CreateChannel(chId);
13:         - -When transmitting data right-directionally:
14:         - -For srcPort: chId!srcPortId;
15:         - -For destPort: chId?xdestPortId=x;;
16:         chId ← conxId+"L";
17:         CreateChannel(chId);
18:         - -When transmitting data left-directionally:
19:         - -For srcPort: chId?srcPortId;
20:         - -For destPort: chId!xdestPortId=x;;
21:     end if
22: else if ConxType is sampling or blackboard then
23:     eventId ← conxId;
24: else if ConxType is event then
25:     if conx_symbol is "–>" then
26:         ChannelId ← conxId+"R";
27:     else if conx_symbol is "<–>" then
28:         ChannelId1Id ← conxId+"R";
29:         ChannelId2Id ← conxId+"L";
30:     end if
31: end if
32: For para_conx:
33: if directionalSymbol is "–>" then
34:     - -When ocuuring subprogram calls:
35:     value(destId) ← value(srcId);
36: end if
```

this connection are mapped to variables with the ends' names as follows:

var th1_DataROut;

var th2_DataRIn;

The connection *Blackboard_CnR* is transformed to a STCSP event with the same name *Blackboard_CnR* which represents thread *th1* writes the value of *th1.DataROut* to port *th2.DataRIn* through this connection with the following event:

Blackboard_CnR{th2_DataRIn = th1_DataROut;}.

(2) **Queued/Buffered connections are transformed to STCSP channel communications.**

For bidirectional buffer/queue data port, it should be described as two channels representing right and left directional communications respectively. In Figure 5, buffer connection *Buffer_Cn* is transformed to two channel communications with the following channel definitions:

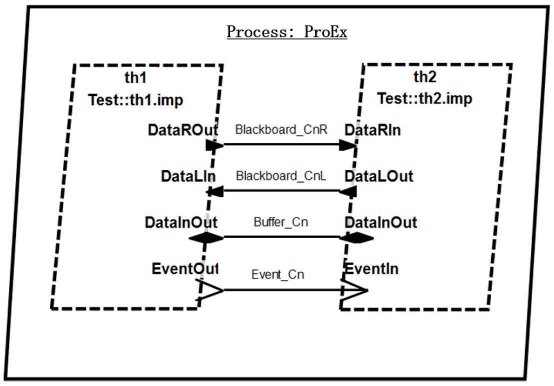


FIGURE 5. Event/Data port connections between threads.

channel Buffer_CnR BufNum;

channel Buffer_CnL BufNum;

Channel *Buffer_CnR* represents the right-direction data transmission which is from *th1* to *th2* and channel *Buffer_CnL* represents the left-direction data transmission, and *BufNum* is the buffer number of a channel.

(3) **Event connections** representing control flows are transformed to STCSP events. We use a 1-buffered channel to represent the event mechanism in event connections, and define a STCSP variable with unique value to represent the event in each event connections respectively.

In figure 5, the event connection *Event_Cn* presents a control flow from thread *th1* to *th2*. This event connection is described as a 1 buffered channel *ch_Event_Cn*, which represents an event channel, with the ID of the connection *Event_Cn* as follows:

//Define a channel representing an event connection with buffer 1

channel ch_Event_Cn 1;

//Define an variable representing the event in each event connections

//Each Event has an unique value

var Event_Cn = 0;

th1()= … → ch_Event_Cn!Event_Cn → Skip;

th2()= Event_Cn?x → …;

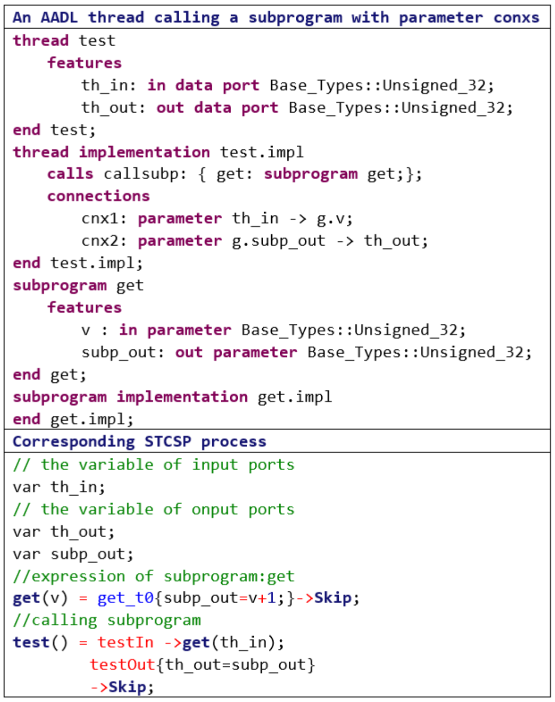
pro_Ex() = th1() || th2();

The event *Event_Cn* represents an event which means that only when thread *th1* sends event *ch_Event_Cn!Event_Cn* and thread *th2* receives this event, then will thread *th2* perform its own execution. The STCSP *pro_Ex() = th1() || th2();* described the event-scheme threads execution.

2) Parameter Connections

Parameter connections represent data flows between called subprograms and their calling threads, so parameter connections are transformed to parameter transmissions (like C programs) when a thread calls subprograms.

For example in Figure 6, the value of *th_in* is transmitted to subprogram *get* through a subprogram call *get(th_in)*.



```
An AADL thread calling a subprogram with parameter conxs
thread test
    features
        th_in: in data port Base_Types::Unsigned_32;
        th_out: out data port Base_Types::Unsigned_32;
end test;
thread implementation test.impl
    calls callsubp: { get: subprogram get;};
    connections
        cnx1: parameter th_in -> g.v;
        cnx2: parameter g.subp_out -> th_out;
end test.impl;
subprogram get
    features
        v : in parameter Base_Types::Unsigned_32;
        subp_out: out parameter Base_Types::Unsigned_32;
end get;
subprogram implementation get.impl
end get.impl;
Corresponding STCSP process
// the variable of input ports
var th_in;
// the variable of onput ports
var th_out;
var subp_out;
//expression of subprogram:get
get(v) = get_t0{subp_out=v+1;}->Skip;
//calling subprogram
test() = testIn ->get(th_in);
          testOut{th_out=subp_out}
          ->Skip;
```

FIGURE 6. Parameter connections.

B. TRANSFORM A BEHAVIOR SPECIFICATION TO A COMPOSITE STCSP PROCESS

A behavior specification conformed to the Behavior Annex with *variables*, *states*, *transitions* is transformed to a composite STCSP process which comprises multiple sub-processes that collectively represent the entire behaviors of all the AADL state transitions.

The states in a behavior specification may be *initial*, *complete*, *return*, *final* or intermediate. *Transitions* define state transitions from a source state to a destination state. Transitions can be guarded with events or boolean conditions using "[]." An action part attached to the transition can perform subprogram calls, message sending or assignments using "{}." The action is related to the transition and not to the states: if a transition is enabled, the action part is performed and then the current state becomes the destination state. Transitions in both subprograms and threads start from an *initial* state. A transition to a *return* state ends a subprogram. A transition to a *complete* state completes a thread and it will resume from this *complete* state at next dispatch.

A behavior specification is defined as a tuple:
BA = (sta_var, sta, trans).
– *sta_var* represents state variables;
– *sta* represents states;
– *trans* represents state transitions.

Rule tr_BS transforms a behavior specification to a STCSP process which presents all the behaviors described in the *transitions* part. This rule performs the following procedures:

- **Tr_BS1**: Map state variables in a behavior specification to STCSP variables;
- **Tr_BS2**: Define STCSP variables corresponding to *initial*, *complete*, and *return* states to denote the current state a transition automaton currently runs at;
- **Tr_BS3**: Transform guards in AADL using rule *tr_guard* (see Section IV-B.1) to STCSP guards; transform actions in AADL using rule *tr_action* (see Section IV-B.2) to STCSP actions;
- **Tr_BS4**: Transform transitions along with its guards and actions to STCSP processes. The transformed STCSP processes should present the semantics that: the *transitions* part starts from a *initial* state; a transition to a *return* state ends a subprogram and a transition to a *complete* state completes a thread (see Section IV-B.3-RTrans1 and RTrans2);
- **Tr_BS5**: Transformed STCSP processes representing state transitions are combined to one composite STCSP process which represents the entire behavior specification (see Section IV-B.3-RTrans3).

1) TRANSFORM GUARDS IN TRANSITIONS TO STCSP GUARDS AND/OR EVENTS

Transitions can be guarded by dispatch conditions, or execute conditions. The guard part in AADL and STCSP has the similar semantics that an AADL state transition or a STCSP process can execute only when the guard part is satisfied.

An AADL guard is a tuple:
guard = (dispatch_conditions, execute_conditions).

– *dispatch_conditions* explicitly specifies dispatch conditions out of a *complete* state. A dispatch condition is expressed as a disjunction of conjunctions of dispatch trigger conditions as follows:

– *dispatch_condition ::= conjunction1* **or** *conjunction2* ... **or** *conjunctionM*;
– *conjunctionI ::= triggerI1* **and** *trigeerI2* ... **and** *triggerIN*;
– *trigger ::= (InPortId, InPortType)*;
– *execute_conditions* represents execute conditions that specify transition conditions out of an execution state to another state.

Rule tr_guard transforms an AADL guard composed of dispatch conditions and/or execute conditions to STCSP guards of a STCSP process and/or trigger events, and is listed as Algorithm 2.

For example in Figure 7, there is an event guard *[in1?]* in transition *t0* and an expression guard *[x>1]* in transition

Algorithm 2 tr_guard() – Transform a Transition Guard to STCSP Guards and/or Event(s)

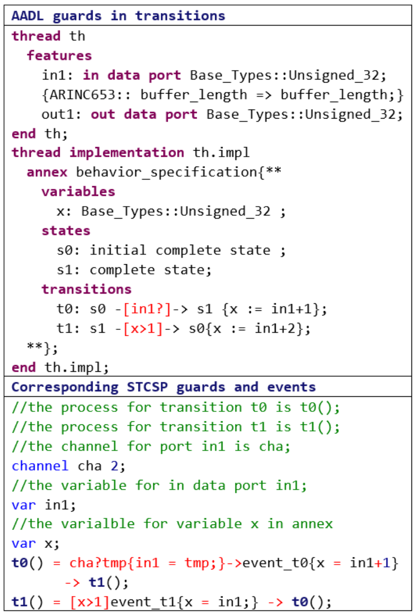
```
Input: (AADL) guard ::= (dispatch_conditions, execute_conditions);
        dispatch_condition ::= conjunction1 or conjunction2 ą or conjunctionM;
        conjunctionI ::= triggerI1 and trigeerI2ą and triggerIN;
        trigger ::= (InPortId, InPortType);
        execute_condition ::= logical_value_expression.
Output: (STCSP)DisEvents ::= (ConEvents1 [] ConEvents2 [] ConEventsM);
        ConEventsI ::= (InEventI1 "–>" InEventI2 ą "–>" InEventIN);
        Logical_expre.
1:  if execute_conditions then
2:      replace '=' with "==", "not" with '!', "or" with "||", "and" with "&&";
3:      Logical_expre ← replaced execute_conditions;
4:  else if dispatch_conditions then
5:      - -Disjunction of dispatch_trigger conjuntions;
6:      for I = 1 ··· M do
7:          for j = 1 ··· N do
8:              - -AADL dispatch_trigger → STCSP events;
9:              if triggerIJ.InPortType is in_event_port then
10:                 eventId := ID(the connection ended with this port);
11:                 InEventIJ ← eventId;
12:             else if triggerIJ.InPortType is in_event_data_port then
13:                 channelId := ID(the connection ended with this port);
14:                 InEventIJ ← channelId?tmp{trigger.InPortId := tmp};
15:             end if
16:         end for
17:     end for
18: end if
```

t1. [in1?] is transformed to an input event using a channel defined by the port connection ended with port *in1*, and *[x>1]* is transformed to a expression guard section of a transition event *event_t1*.

2) TRANSFORM ACTIONS IN TRANSITIONS TO STCSP EVENTS AND/OR SUBPROGRAM CALLS

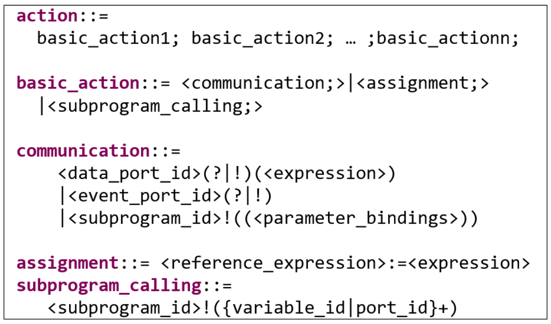
An Action in AADL transition is comprised of one or multiple basic actions representing communications, assignment operations or subprogram calls. The abstract syntax of an action is listed as following Figure 8.

When an action comprises multiple communications and assignments, this action is split into multiple sections by communications and subprogram calls: each communication or subprogram call is transformed to an event and adjacent assignment operations form an event named by the transition ID.



```
AADL guards in transitions
thread th
  features
    in1: in data port Base_Types::Unsigned_32;
    {ARINC653:: buffer_length => buffer_length;}
    out1: out data port Base_Types::Unsigned_32;
end th;
thread implementation th.impl
  annex behavior_specification{**
    variables
      x: Base_Types::Unsigned_32 ;
    states
      s0: initial complete state ;
      s1: complete state;
    transitions
      t0: s0 -[in1?]-> s1 {x := in1+1};
      t1: s1 -[x>1]-> s0{x := in1+2};
  **};
end th.impl;
Corresponding STCSP guards and events
//the process for transition t0 is t0();
//the process for transition t1 is t1();
//the channel for port in1 is cha;
channel cha 2;
//the variable for in data port in1;
var in1;
//the varialble for variable x in annex
var x;
t0() = cha?tmp{in1 = tmp;}->event_t0{x = in1+1}
       -> t1();
t1() = [x>1]event_t1{x = in1;} -> t0();
```

FIGURE 7. Transform Guards in Transitions to Guards And/Or Events in STCSP events.



```
action::=
  basic_action1; basic_action2; … ;basic_actionn;

basic_action::= <communication;>|<assignment;>
  |<subprogram_calling;>

communication::=
    <data_port_id>(?|!)(<expression>)
    |<event_port_id>(?|!)
    |<subprogram_id>!((<parameter_bindings>))

assignment::= <reference_expression>:=<expression>
subprogram_calling::=
   <subprogram_id>!({variable_id|port_id}+)
```

FIGURE 8. The syntax of actions.

Rule tr_action transforms an AADL action part of a transition to STCSP action part of an event.

An AADL action part comprises one or multiple basic actions. A basic action is a tuple:

basic_action = (communications, assignments, subprogram_callings).

- A *communication* is a tuple *communication = (portId, expression)*, *portId* is ID of communication port, *expression* is contents of this communication;
- An *assignment* is the form: *reference_expression := expression*, the value of *expression* is assigned to the *reference_expression*. *Reference_expression* may be a feature, a data subcomponent, a state variable and so on.
- *subprogram_calling* is of the form: *subprogram_id!(parameter_bingdings)* which expresses that the containing component calls the contained subprogram with parameter transmissions defined by the *parameter_bingdings*.

Rule tr_action is listed as Algorithm 3.

Algorithm 3 tr_action() – Transform an AADL Action to STCSP Action Event(s)

```
Input: (AADL)action::= basic_action1; basic_action2; ··· ;basic_actionN;
        basic_action ::= (communications, assignments, subprogram_callings);
        communication = (portId, expression);
        assignment ::= target := expression;
        subprogram_calling ::= subprogram id!(parameter_binds);
Output: (STCSP){action1} –> {action2} –> ··· –> {actionN};
        action ::= subprogram_calling/channel_communication/assignment_action;
        subprogram_callings ::= subpId!(parameter_binds);
        channel_communication ::= chId!/?(logical_expression);
        assignment_action ::= varId = logical_expression;
 1: for i = 1 ··· N do
 2:     - -subprogram calls:
 3:     if subprogram_calling || DataAccess then
 4:         subpId ← subprogram_id;
 5:         actionI ← subpId!(parameter_binds);
 6:         - -communication actions:
 7:     else if PortInOut then
 8:         channelId := ID(the connection ended with this port);
 9:         actionI ← channelId!/?{expression};
10:         - -assignment actions:
11:     else if assignment_action then
12:         varId ← ID(target);
13:         actionI ← (varId := expression;);
14:     end if
15: end for
```

For example in Figure 9, the action of transition *t0* comprises five parts: assignment operations $x := in1$ and $y := x + 1$, a port communication $out1!(y)$, assignment operation $z := y + 2$ and a port communication $out2!(z)$. The first two adjacent assignment operations $x := in1$ and $y := x + 1$ are placed into an event $event\_t0\_0$, assignment operations $z := y + 2$ is placed into another event $event\_t0\_1$. The port communication $out1!(y)$ and $out2!(y)$ among assignments are treated as channel communications. The sequence of these transformed events is the same as the sequence of the AADL action parts.

3) TRANSFORM TRANSITIONS TO STCSP PROCESSES

The transformation of the *transitions* part is comparatively sophisticated. The principles for transformations of AADL state transitions have the following aspects.

RTrans1. For a single transition: Transform a single transition to a STCSP process.

```
AADL actions in transitions
thread th
  features
    in1: in data port Base_Types::Unsigned_32;
    out1: out data port Base_Types::Unsigned_32;
     {ARINC653:: buffer_length => buffer_length;}
    out2: out data port Base_Types::Unsigned_32;
     {ARINC653:: buffer_length => buffer_length;}
end th;
thread implementation th.impl
  annex behavior_specification{**
    variables
      x,y,z: Base_Types::Unsigned_32 ;
    states
      s0: initial complete state ;
    transitions
      t0: s0 -[]-> s0{x := in1;y := x+1; out1!(y);
                      z := y+2; out2!(z)};
  **};
end th.impl;
Corresponding STCSP actions and events
//the process for transition t0 is t0(), for transition t1 is t1();
//the channel for ports out1, out2 are respective chaOut1 and chaOut2;
channel chaOut1 2;
channel chaOut2 2;
//the variable fordata ports;
var in1; var out1; var out2;
//the varialble for variable x in annex
var x; var y; var z;

t0() = event_t0_0{x = in1; y = x+1} ->
      chaOut1!(y) ->event_t0_1{z=y+2} ->
      chaOut2!(z) -> t0();
```

FIGURE 9. Transform Actions to Events.

1 The *guard* of the STCSP process comprises two parts:
   - a. the AADL transition's guard, stored in *transGuard*;
   - b. the variables for *initial/complete/return* states to denote the current enabled state, stored in *stateIdentis*;

2 The *action* of the STCSP process comprises two parts:
   - a. the AADL transition's action, stored in *transActions*;
   - b. the *Set* operation which enables the next transition's state and the *Reset* operation which disables the current executing transition's state, stored in *ResSetStates*;

3 The type of the destination state of a transition determines the next transition:
   - a. if the destination state is intermediate, i.e. it is not *initial/complete/return/final*: the end of the STCSP process representing the current AADL transition directs to the STCSP process representing the AADL transition stared with the current destination state;
   - b. if the destination state is *complete/return*: the STCSP process representing the current AADL transition finishes with a *Skip*; and set the destinational *complete/return* state so as to state that it will resume with this state when next dispatched;
   - c. if the destination state is *final*: the STCSP process representing the current AADL transition stop with a *Skip*; and reset all the *complete/return* states so as to state that it will not resume with any complete states or other states.

```
Transitions with complete and intermediate states
thread th2
  features
    th_in: in data port Base_Types::Unsigned_32;
    th_out: out data port Base_Types::Unsigned_32;
end th2;
thread implementation th2.impl
  annex behavior_specification {**
    variables
      x: Base_Types::Unsigned_32;
    states
      s0: initial complete state;
      s1: state;
     transitions
       t0:s0 -[]-> s1{x:=th_in+1};
       t1:s1 -[]-> s0{x:=th_in+2};
    **};
end th2.impl;
Corresponding STC process
//Define variables for ports and state variables
var x; var th_in;
//"init_state = 1" means that the intial state is enabled
var init_state = 1;
//define STCSP variables for every complete/return states;
var s0 = 0;
//”s0 = 0;” means that s0 is disabled;
//”inti_state =0;” means that the intial state is disabled;
t0() =
[init_state==1||s0==1]event_t0{x=th_in+1;s0=0;inti_state =0;}
     -> t1();
//”s0=1;” means that s0 is enabled;
t1() = event_t1{x=th_in+2;s0=1;} -> Skip;

th2() = th2_input -> t0();
```

FIGURE 10. Transitions start from complete and intermediate states.

For example in Figure 10, there are two transitions: transition *t0* whose destination state is a intermediate state *s1* and transition *t1* whose destination state is a complete state *s0*. This situation means that when transition *t0* goes to its destination state *s1*, the transition *t1* starting from *s1* will continue to run to state *s0* and this round of transition ends because of *s0* is complete.

RTrans2. For multiple transitions with a same source state: assemble these transformed STCSP processes with choice operator "[]", and each of them combined with their transformed guard and action parts.

For example in figure 11, transition *t1 t2* and *t3* start from a same source state s1, their transformed STCSP processes are combined by choice operator "[]" to construct a composite process *t1_t2_t3* which presents possible STCSP processes for three transitions.

RTrans3. Assemble all STCSP processes representing AADL transitions starting with *initial* or *complete* states with choice operator "[]." These composed process will determine to execute *initial* processes or which complete processes with respectively guard parts.

For example in Figure 11, the behaviors of five transitions can be expressed in *t0()[]t4()*, where *t0* is the transition start from the *initial complete* state *s0*, and *t4* is the transition start from the *complete* state *s2*.

A transition is a tuple:

transition = (labelId, srcState, guard, destState, action).

– *LabelId* is ID of the transition;
– *SrcState* and *destState* are the source state and the destination state of the transition;

```
Multi-transitions with a same source state
thread th3
  features
    th_in: in data port Base_Types::Unsigned_32;
    th_out: out data port Base_Types::Unsigned_32;
end th3;

thread implementation th3.impl
  annex behavior_specification {**
    variables
      x: Base_Types::Unsigned_32;
    states
      s0: initial complete state;
      s1: state;
      s2: complete state;
    transitions
      t0:s0 -[]-> s1{x:=th_in};
      t1:s1 -[x>1]-> s0{x:=th_in+1};
      t2:s1 -[x=1]-> s0{x:=th_in+2};
      t3:s1 -[x<1]-> s2{x:=th_in+3};
      t4:s2 -[]-> s0{x:=th_in+4};
**};
Corresponding STC process
//Define variables for ports and state variables
var x; var th_in;
//"init_state = 1" means that the intial state is enabled
var init_state = 1;
//define STCSP variables for every complete/return states;
var s0 = 0;
//”s0 = 0;” means that s0 is disabled;
//”inti_state =0;” means that the intial state is disabled;
t0() =
  [init_state == 1||s0==1]event_t0{x=th_in+1;s0=0;init_state =0;}
  -> t1_t2_t3();
//”s0=1;” means that s0 is enabled;
t1() = [x>1]event_t1{x=th_in+1;s0=1;} -> Skip;
t2() = [x==1]event_t2{x=th_in+2;s0=1;} -> Skip;
t3() = [x<1]event_t3{x=th_in+3;s2=1;} -> Skip;
t1_t2_t3() = t1()[]t2()[]t3();
t4() = event_t4{x=th_in+4;s0=1;} -> Skip;

th3() = th3_input -> (t0()[]t4());
```

FIGURE 11. Multi-Transitions start from a same source state.

– *Guard* and *action* are the guard part and action part of a transition.

A transition is of the form:

label_i: s_i-[[guard_i]] —> d_i[{action_i}];*

Rule tr_transitions transforms an entire AADL *transitions* part in a behavior specification to a composite STCSP process representing all the transition behaviors of this behavior specification.

Rule tr_transitions is listed in Algorithm 4 in which *Set(StateId)* is to set the variable representing state *StateId* to value *1* and *Reset(StateId)* is to reset the variable representing state *StateId* to value *0*.

C. TRANSFORMATION OF COMPONENTS

An AADL component is transformed to a STCSP process, while the transformed STCSP process maintains the component hierarchy of the source AADL. The component hierarchy of a system instance is determined by recursively instantiating the subcomponents of a top-level system.

With the AADL definitions, we adopt the following transformation methods for software components.

1) TRANSFORMATION OF SUBPROGRAMS

A subprogram is transformed to a STCSP process with parameter transmissions which takes *in* parameters and outputs *out* parameters.

Algorithm 4 tr_transitions() – Transform *Transitions* Part of a Behavior Specification to a STCSP Process

```
Input: AADL state_transitions
    state_transitions ::=
    transitions
    label_0: s_0-[ [guard_0] ]-> d_0[{action_0}];
    label_1: s_1-[ [guard_1] ]-> d_1[{action_1}];
    . . .
    label_N: s_N-[ [guard_N] ]-> d_N[{action_N}];
Output: STCSP process
    P_trans() = P_i() [] P_j() [] · · · [] P_k();
    P_p() = [stateIdentis_p && transGuard_p] eventId_p{transActions_p;ResSetStates_p;} -> NextTrans;
1: Define variables for state variables using tr_behavior_variable();
2: Define variables to represent initial and complete states using tr_behavior_state();
3: Set(initialState);
4: Reset(completeStates);
5: for I = 1 ... n do
6:     P_I ← label_I;
7:     eventId_I ← event+ label_I;
8:     transGuard_I ← tr_guard(guard_I);$^{RTrans1.1.a}$
9:     transActions_I ← tr_action(actions_I);$^{RTrans1.2.a}$
10:    if s_I is initial and this transition is initialization then$^{RTrans1.1.b}$
11:        stateIdentis_I += "(initial == 1)||";
12:    else if s_I is complete then
13:        stateIdentis_I += "s_I = 1";
14:    end if
15:    if s_I is initial or complete then
16:        ResSetStates_p += Reset(S_I);$^{RTrans1.2.b}$
17:        P_trans() + = "P_I() []";$^{RTrans3}$
18:    end if
19:    if d_I is intermediate then$^{RTrans1.3.a}$
20:        NextTrans ← P_I;
21:    else if d_I is complete then$^{RTrans1.3.b}$
22:        NextTrans ← Skip;
23:        ResSetStates_p ← Set(d_I);
24:    else if d_I is final then$^{RTrans1.3.c}$
25:        NextTrans ← Skip;
26:        ResSetStates_p ← Reset(all states);
27:    end if
28:    if multiple transitions started from s_I then$^{RTrans2}$
29:        P_IM ← composition of labels of transitions started from s_I;
30:        P_IM() + = "P_I() []";
31:        –I, K ... M are label number of the transitions started from a same source state
32:    end if
33: end for
```

An AADL subprogram is a tuple:

subprogram = (subpID, InPara, OutPara, connections, BS).

- *SubpID* is ID of a subprogram;
- *InPara* and *OutPara* are parameters into and out of a subprogram;
- *Connections* are between parameters of a subprogram and its containing components' features;
- *BS* is the behavior specification.

Rule tr_subprogram maps an AADL subprogram to a STCSP process expression with parameter transmissions between this called subprogram and its calling threads.

An example of transformation of subprograms is illustrated in previous Figure 6.

2) TRANSFORMATION OF THREADS

An AADL thread is a tuple:

thread = (thID, properties, features, connections, subpCalls, BS).

- *ThID* is ID of a thread;
- *Properties* defines a thread is a periodic thread or an aperiodic thread dispatched by events;
- *Features* and *connections* are between threads or a thread and its containing process;
- *SubpCall* represents subprogram callings that a thread calls a subprogram in its behavior specification;
- *BS* is the behavior specification.

Rule tr_thread maps an AADL thread to a STCSP process expression which describes input and output events, behaviors of its behavior specification which may call subprograms with corresponding parameter transmissions.

Periodic threads are periodically dispatched. The time capacity of the comprising process has integral multiple of the period of its comprised periodic threads. We will introduce the consideration of timing properties in the following section.

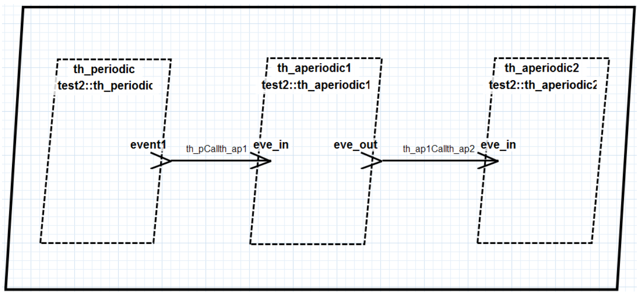


FIGURE 12. Periodic and aperiodic dispatch.

Aperiodic threads are dispatched by arrival of events input from the pre-declared event ports. For example in Figure 12, a process *pro* comprises threads *th_periodic*, *th_aperiodic1* and *th_aperiodic2*, where thread *th_aperiodic1* is dispatched by a queued event from a channel *th_pCallth_ap1* and thread *th_aperiodic2* is dispatched by a queued event from a channel *th_pCallth_ap2*. The three corresponding thread are modeled as follows:

th_periodic() = ...→ th_pCallth_ap1!event1 → Skip;
th_aperiodic1() = th_pCallth_ap1?x → ...

th_ap1Callth_ap2!event2 → Skip;
th_aperiodic2() = th_ap1Callth_ap2?y → ...;

3) TRANSFORMATION OF PROCESSES

An AADL process is a tuple:

process = (proID, timeCapa, features, connections, threads, subps, BS).

- *ProID* is ID of a process;
- *TimeCapa* is time capacity;
- *Features* and *connections* are between processes;
- *Threads* and *subprograms* are components contained in a process;
- *BS* is behavior specification.

Rule tr_process maps an AADL process component to a STCSP process expression which describes input and output events, concurrent behaviors of its contained threads.

For periodic threads, we introduce the consideration of timing properties (such as period, time capacity and deadline of threads, duration of processes) in the following section.

For a aperiodic thread dispatched by an event, its STCSP process expression is started by a concurrent event which dispatch this thread, and this thread are connected by a concurrent symbol "∥" with other threads.

For example in Figure 12, the process *pro* is described as follows:

pro() = th_periodic() ∥ th_aperiodic1() ∥ th_aperiodic2();

4) TRANSFORMATION OF AN ENTIRE SYSTEM

An AADL System contains sequentially loaded processes with respective durations. From the AADL Reference Manual, We know that the basic schedule unit is *thread*, *process* only represents a virtual address space. Therefore, a system containing multiple processes performs processes' execution with a defined sequence and every process has a pre-defined duration. So we transform an AADL system containing multiple processes as sequential executed processes with a notion of timed *interrupt* which represents the duration of a process.

An AADL system is a tuple:

system = (sysID, features, connections, processes).

- *sysID* is ID of a process;
- *Features* and *connections* are between processes;
- *processes* are AADL process components contained in a system;

Rule tr_system maps an AADL system to a STCSP process expression which describes input and output events, sequential run processes contained in this system with their respective duration.

For example, an AADL system *sys* contains three process *pro1, pro2, pro3* whose duration is *duration1, duration2, duration3* respectively. So the AADL system *sys* can be transformed as follows:

sys() = pro1() interrupt[duration1] pro2()
        interrupt[duration1+duration2] pro3()
        interrupt[duration1+duration2+duration3]
sys();

The handling of timed interrupt is described in section IV-D.4.

D. TRANSFORMATION OF TIMING PROPERTIES

The most exceptional advantages of STCSP against traditional CSP may be its timing characteristics when modeling real-time systems. We consider time capacities, deadlines, periods of AADL threads (ARINC653 processes/tasks), and durations of AADL processes (ARINC653 partitions).

1) TRANSFORM AADL THREAD TIME CAPACITY AND DEADLINE

We model time capacities and deadlines of AADL threads (ARINC653 processes/tasks) using STCSP **deadline**.

The deadline time of ARINC653 processes is to determine whether the process is satisfactorily completing its processing within the allotted time. When a process is started, its deadline is set to the value of current time plus time capacity. Therefore, the consideration of deadline time is associated with time capacity. In STCSP, "*P deadline[t]*" is constrained to terminate within *t* time units.

For example, thread *thr* with event *a* and *b* and time capacity *2* is modeled as follows:

thr() = (a → b → Skip) deadline[2];

2) TRANSFORM AADL THREAD PERIOD

We model periods of periodic AADL threads (ARINC653 processes/tasks) using STCSP timed **interrupt**.

Periodic threads (ARINC653 tasks/processes) are periodically dispatched. Typically, the comprising partition period is the greatest common factor of comprised process durations within a partition.

In STCSP, "*P interrupt[t] Q*" behaves as process *P* until *t* time units elapse and then switches to process *Q*. For instance, process "*(a → b → c → ...) interrupt[t] Q*" will engage in event *a*, *b*, *c*, ... of process *P* if *t* time units haven't elapsed. When *t* time units have elapsed, then the process engages in process *Q*. Therefore, the periodic threads can be modeled using STCSP timed *interrupt*.

For example, thread *thr* with period *4* can be modeled as follows:

thr() = (a → b → Skip) interrupt[4] thr();

3) TRANSFORM AADL PROCESS DURATION

We model durations of AADL processes (ARINC653 partitions) using STCSP timed **interrupt**.

Scheduling of AADL processes (ARINC653 partitions) is strictly deterministic round-robin over time defined by partition durations. A partition duration is the amount of execution time required by the partition. Partitions are scheduled on a fixed, cyclic basis. To assist this cyclic activation, ARINC653 OS maintains a major time frame of fixed duration, which is periodically repeated throughout the module's runtime operation.

The duration of processes can also be modeled by STCSP timed interrupt. For example, an ARINC653 module *M* comprise process *Proc1* and *Proc2* which have duration *8* and *10* respectively can be modeled as follows:

M() = (Proc1 interrupt[8] Proc2) interrupt[10+8] M();

"*Proc1 interrupt[8] Proc2*" indicates that *Proc1* has duration *8* and the duration of *Proc2* is represent by *interrupt[10+8]*.

4) A SUMMARY OF TIMING TRANSFORMATION

As a summary of this section, we use a simple but complete example to describe our timing transformation for time capacities, deadlines, periods, and durations.

Given an AADL system (an ARINC653 module) *M*, which has process (ARINC653 partition) *Proc1* and *Proc2* with duration *8* and *10* respectively. *Proc1* has a periodic thread (ARINC653 process/task), which is *Per1* with deadline *2* and periodic *4*, and an aperiodic thread *Aper* with deadline *2*. The aperiodic thread *Aper* is dispatched by an event *event1* from the periodic thread *Per1* using queued channel *ch1*. *Proc2* has only one periodic thread *Per2* with deadline *2* and period *10*. This system *M* can be modeled by PAT-readable STCSP in Table 2:

TABLE 2. An typical example of timing transformations.

```
channel ch1 1; // Define a channel with buffer size 1;
var event1 = 1; // Define an event;
// Periodic thread Per1;
Per1 = ( (Wait[1]; p –>> ch1!event1 –>>Skip) deadline[2] ; Skip )
            interrupt[4] Per1;
// Aperiodic thread Aper;
Aper = ch1?x –>> Skip; (q –>> Skip) deadline[2]; Aper;
// Process Pro1;
Proc1 = Per1 || Aper;
// Periodic thread Per2;
Per2 = ( (Wait[1]; b–>>Skip)deadline[2];Skip ) interrupt[10]Per2;
// Process Pro2;
Proc2 = Per2;
// System M;
M() = (Proc1 interrupt[8] Proc2) interrupt[10+8] M();
```

In the above STCSP model, operator "–>>" means an urgent event which takes no time when executed Concurrent threads *Per1* and *Aper* is connected by operator ||, which means that the aperiodic thread *Aper* is dispatched only it receives the dispatch event *event1* from the periodic thread *Per1* using the channel *ch1*. The usage of *deadline* is easily explicable.

We should pay attention to the operator timed *interrupt* and *Wait*. When using multiple timed *interrupt* to describe each AADL process's duration, it should be noted that the value in *interrupt* should be the duration value of the current process plus the previous processes' duration values. For example in "*M() = (Proc1 interrupt[8] Proc2) interrupt[10+8] M();*", "*Proc1 interrupt[8] Proc2*" means that *Proc1* has duration *8*; when we describe the duration of the second process *Pro2* we should plus the duration of the previous process using *interrupt[10+8]*.

The STCSP operator *Wait* has no direct relation to our transformation, but it can help operator timed *interrupt* to realize rigorous periodic executions of periodic threads. In STCSP, an event will take arbitrary amount of time if we use "–>" event prefix. Therefore there exists an undesirable situation: for example the periodic *Per1* in the above example, if we remove the *Wait[1]*, when we simulate the execution of its comprising process *Proc1*, **within** the duration of the process *Proc1*, the event *p* of *Per1* may occur three times (it should only occur two times in our intention since the duration of the comprising process *Proc1* is two times of the period of the comprised thread *Per1*), because event *p* can execute on time *8* since the event can execute taking no times. So we can use *Wait* to denote that the event *p* takes some time and guarantee *Pro1* can execute two times and only two times, although the operator *Wait* is not used to denote event execution time.

V. EXPERIMENT

Using a key portion of a complex system as a case study will be persuasive to evaluate our methodology. We select "Auto Pilot Controller" (F-16 APC), the key portion of a complex simulation system "A Non-Linear F - 16 Simulation System" built using data published by NASA. We build an AADL model for the key portion "F-16 APC" which is based on ARINC653 OS platform, transform this AADL model to a STCSP model using our model-transformation method, and perform formally analysis on the transformed STCSP model using PAT.

A. SUMMARY OF "A NON-LINEAR F - 16 SIMULATION SYSTEM"

"A Non-Linear F - 16 Simulation System" is a non-linear F-16 model that simulates the dynamics of the real aircraft. The original version was a low fidelity model built using data described by Brian L. Stevens and Frank L. Lewis in [18], and the high fidelity model built using data published by NASA in [34] which has a larger aerodynamic data range and implements the leading edge flap. Force and moment coefficients are at angle of attacks from −20 to 90 degrees and side-slip angles of −30 to 30 degrees. The leading edge flap allows the F-16 to fly at larger angle of attacks by reducing the tendency to stall [35].

Its input includes four controls, thirteen states, the leading edge flap deflection and a model flag; It outputs twelve state derivatives and six other states of flight. It allows for control over thrust, elevator, aileron and rudder. The F-16 model allows for control over thrust, elevator, aileron and rudder. All of the actuators are modeled as first-order lags with a gain(K) and limits on deflection and rates. The navigation equations are taken from page 81 of Stevens and Lewis [18] and the equations that determine the force and moment coefficients are taken from pb. 37- 40 of the NASA Report [34].

We refer the readers to the home page [35]. This Simulation System is illustrated as Figure 13.

B. THE KEY PORTION: "F-16 APC"

The "F-16 APC" comprises:

(1) Flight attitude control system.

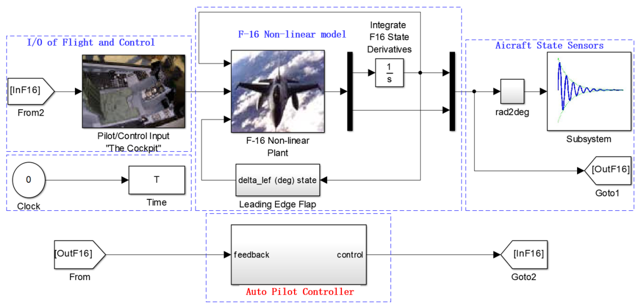


FIGURE 13. F-16 Aircraft flight simulation system.

(2) Flight path control system.

(3) Airspeed control System.

This model allows for control over thrust, elevator, aileron and rudder. The thrust is measured in pounds, acts positively along the positive body x-axis. A positive deflection gives a decrease in the body rates for the other control surfaces. A positive aileron deflection gives a decrease in the roll rate p, it requires the right aileron deflect downward and the left aileron deflect upward. A positive elevator deflection results in a decrease in pitch rate a, so elevator is deflected downward. A positive deflection of the rudder decreases the yaw rate r and can be described as a deflection to right. The high fidelity model has a additional control surface that allows for the F-16 to fly at higher angles of attack.

C. AADL MODEL OF THE KEY "F-16 APC"

We build an AADL model of the "F-16 APC" which is based on ARINC653 OS platform using AADL and its Behavior Annex . The modeling style based on ARINC653 platform is referred to AADL ARINK653 Annex.

The ARINK653 Annex is defined to support the modeling, analysis and automated integration of ARINC653 and derived or similar partitioned architectures. In ARINK653 Annex, a partition is represented by an AADL *process* component, ARINC653 processes are mapped to AADL *threads* component. Inter-partition communications represent data exchanges across partitions, which comprise queuing and sampling; processes contained within the same partition can exchange data using intra-partition communications which comprise data communications (i.e. buffer and blackboard) and control flows (i.e. event and semaphore). In AADL ARINC653 Annex, **sampling** is specified using AADL *data port* connection, **queuing** is specified using AADL *event data port* connection; **buffers** are specified using AADL *event data port* connections, **blackboards** can be specified using AADL *data port* connections.

The AADL model of the "F-16 APC" is modeled with an AADL *system*: Auto_Pilot, which comprises three processes: *SetEngine*, *SetElevator* and *SetAileronRudder*, who comprises 2 threads and 1 subcomponents, 4 threads and 2 subcomponents, 3 threads and 8 subprograms respectively. The whole AADL model comprises 16 subprogram calls, 107 features, 48 connections and 13 behavior specification items which comprise 59 variables, 44 states, 43 transitions and 114 actions. The elements of the AADL model are listed in TABLE 3.

TABLE 3. AADL model statistics.

Model Elements	Count
system	1
process	3
thread	9
data	9
subprogram	11
feature	107
connection	48
subprogram call	16
behavior annex	13
annex variable	59
annex state	44
annex transition	43
annex actions	114

Process *SetAileronsRudder* as an illustration is illustrated in Figure 14.

D. FORMAL VERIFICATION

We transform the AADL model of "F-16 APC" to a target STCSP model. Given the STCSP model, we use PAT to model check its safety, liveness and trace refinement properties.

1. Safety Verification represents "nothing bad will happen", which comprises deadlock-freeness, reachability, invariance or properties expressed in the form of finite state automata.

(1) In concurrent computing, a **deadlock** is a state in which each member of a group is waiting for some other member to take action, such as sending a message or more commonly releasing a lock. For example:

#assert Auto_Pilot() deadlockfree;

is used to verify wether the system is **deadlock-free**;

(2) **Reachability** refers to the ability to get from one state to another with one or multiple events. For example:

#define goal_setE setE == 1;

#assert SetEngine() reaches goal_setE;

are used to verify wether the system **reaches** the goal of *goal_setE*.

(3) In computer science, a computation is said to be **diverge** if it does not terminate in an unobservable exceptional state, otherwise it is said to converge. Divergent system is usually undesirable. Given a process, it may perform internal transitions forever without engaging any useful events, e.g., $P = (a \to P) \setminus a;$. In this case, P is said to be divergent. Another example in our verification:

#assert Auto_Pilot() ;

is used to verify wether the system is **divergence-free**;

(4) Given a process, if it is **deterministic**, then for any state, there is no two or more out-going transitions leading to different states but with same events. E.g, the following process is not deterministic: $P = a \to Stop\ []\ a \to Skip;$. Another example in our verification:

#assert Auto_Pilot() deterministic;

is used to verify wether the system is **deterministic**;

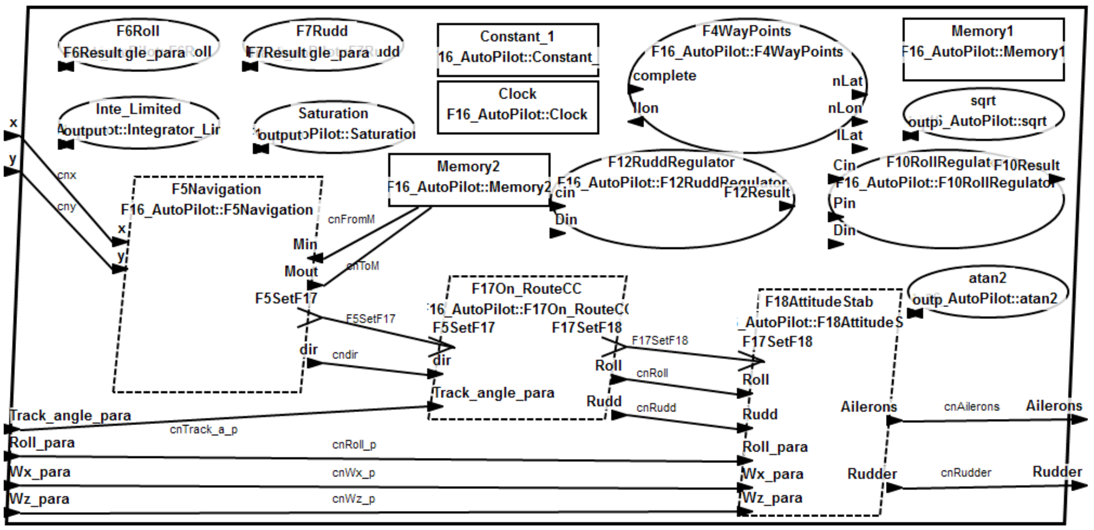


FIGURE 14. AADL Process SetAileronsRudder.

2. Liveness Verification represents "something good eventually happens", and liveness properties are always specified with Linear-time Temporal Logic (**LTL**) formulas. LTL is a temporal logic with modalities referring to time. In LTL, one can encode formulae about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc.

For example:

#assert SetEngine() ⊨ [](Auto_Pilot_cnvt→ <>SetEngine_cnEngine);
#assert SetElevator() ⊨ [](Auto_Pilot_cnalt→ <>Auto_Pilot_cnElevator);
#assert SetAileronsRudder() ⊨ [](Auto_Pilot_cnnpos→ <>Auto_Pilot_cnRudder);
#assert SetAileronsRudder() ⊨[] (Auto_Pilot_cnnpos→<>Auto_Pilot_cnAilerons);

are used to assert that when the system occurs input events, such as *Auto_Pilot_cnvt*, wether the expected output events, such as *SetEngine_cnEngine*, eventually happens, i.e. every input triggers its corresponding desirable outputs.

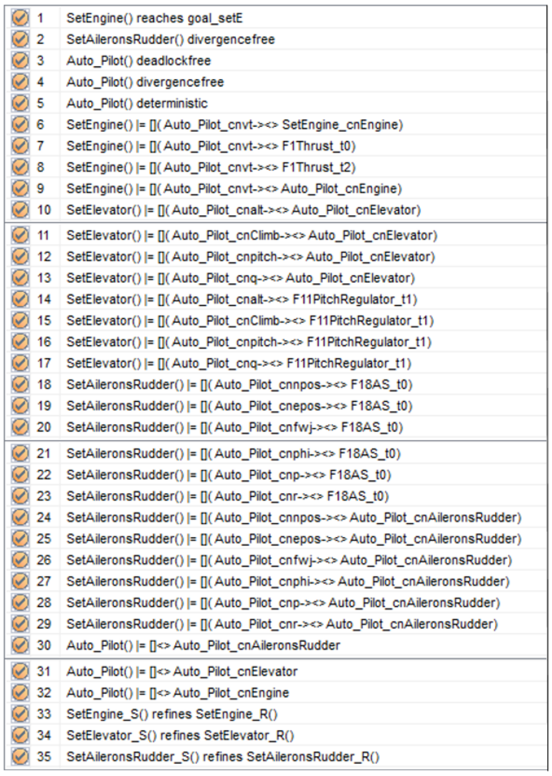
We can also use the above assertion to verify every expected events, for example:

#assert SetAileronsRudder() ⊨ [](Auto_Pilot_cnnpos →<>F18AS_t0);

is used to verify if the expected event *F18AS_t0* happens.

And there are also some other LTL formulas or liveness properties have been executed verification in our project.

3. Trace Refinement Verification checks "whether the abstract behavior trace of an implementation satisfies its abstract behavior trace of a specification." Different from LTL assertions, an assertion for trace refinement compares the whole abstract behaviors of a given process with another process, i.e. whether there is a subset relationship. For one of three process, the refinement analysis of process *SetEngine()* is executed as follows:



1	SetEngine() reaches goal_setE
2	SetAileronsRudder() divergencefree
3	Auto_Pilot() deadlockfree
4	Auto_Pilot() divergencefree
5	Auto_Pilot() deterministic
6	SetEngine() |= [](Auto_Pilot_cnvt-><> SetEngine_cnEngine)
7	SetEngine() |= [](Auto_Pilot_cnvt-><> F1Thrust_t0)
8	SetEngine() |= [](Auto_Pilot_cnvt-><> F1Thrust_t2)
9	SetEngine() |= [](Auto_Pilot_cnvt-><> Auto_Pilot_cnEngine)
10	SetElevator() |= [](Auto_Pilot_cnalt-><> Auto_Pilot_cnElevator)
11	SetElevator() |= [](Auto_Pilot_cnClimb-><> Auto_Pilot_cnElevator)
12	SetElevator() |= [](Auto_Pilot_cnpitch-><> Auto_Pilot_cnElevator)
13	SetElevator() |= [](Auto_Pilot_cnq-><> Auto_Pilot_cnElevator)
14	SetElevator() |= [](Auto_Pilot_cnalt-><> F11PitchRegulator_t1)
15	SetElevator() |= [](Auto_Pilot_cnClimb-><> F11PitchRegulator_t1)
16	SetElevator() |= [](Auto_Pilot_cnpitch-><> F11PitchRegulator_t1)
17	SetElevator() |= [](Auto_Pilot_cnq-><> F11PitchRegulator_t1)
18	SetAileronsRudder() |= [](Auto_Pilot_cnnpos-><> F18AS_t0)
19	SetAileronsRudder() |= [](Auto_Pilot_cnepos-><> F18AS_t0)
20	SetAileronsRudder() |= [](Auto_Pilot_cnfwj-><> F18AS_t0)
21	SetAileronsRudder() |= [](Auto_Pilot_cnphi-><> F18AS_t0)
22	SetAileronsRudder() |= [](Auto_Pilot_cnp-><> F18AS_t0)
23	SetAileronsRudder() |= [](Auto_Pilot_cnr-><> F18AS_t0)
24	SetAileronsRudder() |= [](Auto_Pilot_cnnpos-><> Auto_Pilot_cnAileronsRudder)
25	SetAileronsRudder() |= [](Auto_Pilot_cnepos-><> Auto_Pilot_cnAileronsRudder)
26	SetAileronsRudder() |= [](Auto_Pilot_cnfwj-><> Auto_Pilot_cnAileronsRudder)
27	SetAileronsRudder() |= [](Auto_Pilot_cnphi-><> Auto_Pilot_cnAileronsRudder)
28	SetAileronsRudder() |= [](Auto_Pilot_cnp-><> Auto_Pilot_cnAileronsRudder)
29	SetAileronsRudder() |= [](Auto_Pilot_cnr-><> Auto_Pilot_cnAileronsRudder)
30	Auto_Pilot() |= []<> Auto_Pilot_cnAileronsRudder
31	Auto_Pilot() |= []<> Auto_Pilot_cnElevator
32	Auto_Pilot() |= []<> Auto_Pilot_cnEngine
33	SetEngine_S() refines SetEngine_R()
34	SetElevator_S() refines SetElevator_R()
35	SetAileronsRudder_S() refines SetAileronsRudder_R()

FIGURE 15. All the properties verified.

SetEngine_S() =SetEngine()\ {SetEngine_cnSet_AirSp,SetEngine_cnvt,F1Thrust_t0, F1Thrust_t1,F1Thrust_t2, cnF1SetF13,SetEngine_cnThrust,F13engine_t0, IL_beha,Auto_Pilot_cnEngine};
SetEngine_R() = Auto_Pilot_cnvt→SetEngine_cnEngine →Skip;
#assert SetEngine_S() refines SetEngine_R();

The above expressions are used to verify if the all abstract behaviors of process *SetEngine()* refines the outside abstract behaviors, only with process inputs and outputs without interior actions.

And other processes have also been executed refinement analysis in our project. All 35 properties verified is listed in Figure 15.

VI. CONCLUSION AND FUTURE WORK

This paper presents a method of behavioral description of AADL and a methodology of model transformation from a subset of AADL to STCSP, and then performs model-checking against properties such as safety, liveness and trace refinement etc using PAT.

The subset of AADL we consider includes: all the software components i.e. systems, processes, periodic/aperiodic threads, data, subcomponents etc.; connections, e.g. data port connections, event port connections, parameter connections and subprograms calls etc.; and the behavior specification comprises variables, states and transitions, which is enough to model the most common software.

We apply our methodology to a key portion ''F-16 APC'' of a complex project ''A Non-Linear F-16 Simulation System'' which is built according to aircraft aerodynamics data published by NASA. We build an software AADL model for the ''F-16 APC'', transform this AADL model to a STCSP model using our methodology, and performs model-checking on the transformed STCSP model using PAT. The properties we consider include safety, liveness and trace refinement.

We have developed a prototype tool implementing our transformation rules and integrated it into OSATE2 environment. This prototype tool is developed based on Xtend [36] template and developed as a plugin into OSATE2 which is used by China Aerospace.

Our experience is encouraging, but much more works remain ahead. First, increasingly larger AADL subsets should be considered to face complex applications. For example, a larger subset including such as sporadic threads, shared variables by several threads with subprogram access, complex scheduling, etc., has been considered in our projects. We will consider the largest subset in the future works. Second, the following important work is to verify that the rules of model transformation conforms to semantics-equivalence. We plan to use Isabelle to formally verify that the AADL model along with its transformed STCSP model conforms to operational semantics equivalence. Third, we need more complex industrial applications to examine our theory and the toolset, adjust our schema, and revise the technical architecture and implementation details, so as to realize our object that increase the confidence of safety-critical software.

REFERENCES

[1] SAE. (2017). *Architecture Analysis and Design Language.* [Online]. Available: http://www.aadl.info/aadl/currentsite/

[2] *Architecture Analysis and Design Language (AADL) Annex D: Behavior Model Annex.* SAE Int., New York, NY, USA, 2011.

[3] C. A. R. Hoare, ''Communicating sequential processes,'' *Commun. ACM*, vol. 26, no. 1, pp. 100–106, 1983.

[4] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and É. André, ''Modeling and verifying hierarchical real-time systems using stateful timed CSP,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, p. 3, 2013.

[5] S. Schneider, ''An operational semantics for timed CSP,'' *Inf. Comput.*, vol. 116, no. 2, pp. 193–213, 1995.

[6] National University of Singapore. (2007). *PAT: Process Analysis Toolkit.* Accessed: Nov. 2017. [Online]. Available: http://sav.sutd.edu.sg/PAT/

[7] Y. Liu, J. Sun, and J. S. Dong, ''PAT 3: An extensible architecture for building multi-domain model checkers,'' in *Proc. IEEE 22nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov./Dec. 2011, pp. 190–199.

[8] C. Yang, Y. Dong, F. Zhang, E. Ahmad, and B. Gu, ''Formal semantics of AADL models with machine-readable CSP,'' in *Proc. IEEE/ACIS 11th Int. Conf. Comput. Inf. Sci. (ICIS)*, May/Jun. 2012, pp. 565–571.

[9] E. Ahmad, Y. Dong, S. Wang, N. Zhan, and L. Zou, ''Adding formal meanings to AADL with hybrid annex,'' in *Proc. Int. Workshop Formal Aspects Compon. Softw.*, 2014, pp. 228–247.

[10] E. Ahmad, Y. Dong, B. Larson, J. Lü, T. Tang, and N. Zhan, ''Behavior modeling and verification of movement authority scenario of Chinese Train Control System using AADL,'' *Sci. China Inf. Sci.*, vol. 58, no. 11, pp. 1–20, 2015.

[11] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, ''From AADL model to LNT specification,'' in *Proc. Ada-Eur. Int. Conf. Rel. Softw. Technol.*, 2015, pp. 146–161.

[12] B. Berthomieu *et al.*, ''Formal verification of AADL models with Fiacre and Tina,'' in *Proc. 5th Int. Congr. Exhib. Embedded Real-Time Softw. Syst.*, 2010, pp. 1–9.

[13] N. Abid, Z. Dal, and D. Le Botlan, ''Real-time specification patterns and tools,'' in *Proc. Int. Workshop Formal Methods Ind. Critical Syst.*, 2012, pp. 1–15.

[14] P. C. Ölveczky, A. Boronat, and J. Meseguer, ''Formal semantics and analysis of behavioral AADL models in real-time Maude,'' in *Formal Techniques for Distributed Systems*. Berlin, Germany: Springer, 2010, pp. 47–62.

[15] P. C. Ölveczky. (Jun. 2011). ''Formal model engineering for embedded systems using real-time Maude.'' [Online]. Available: https://arxiv.org/abs/1107.0063v1

[16] Z. Yang, K. Hu, D. Ma, J. P. Bodeveix, L. Pi, and J. P. Talpin, ''From AADL to timed abstract state machines: A verified model transformation,'' *J. Syst. Softw.*, vol. 93, pp. 42–68, Jul. 2014.

[17] K. Hu, T. Zhang, Z. Yang, and W.-T. Tsai, ''Exploring AADL verification tool through model transformation,'' *J. Syst. Archit.*, vol. 61, no. 3, pp. 141–156, 2015.

[18] B. L. Stevens and F. L. Lewis, *Aircraft Control and Simulation*. New York, NY, USA: Wiley, 1992.

[19] *AADL Annex A: ARINC653 Annex*. SAE Int., New York, NY, USA, 2015.

[20] *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1—Required Services*, Aeronautical Radio, Inc., Annapolis, MD, USA, 2010.

[21] T. K. Nguyen, J. Sun, Y. Liu, and J. S. Dong, ''Symbolic model-checking of Stateful timed CSP using BDD and digitization,'' in *Formal Methods and Software Engineering*. Berlin, Germany: Springer, 2012, pp. 398–413.

[22] J. Sun, Y. Liu, J. S. Dong, and X. Zhang, ''Verifying Stateful timed CSP using implicit clocks and zone abstraction,'' in *Formal Methods and Software Engineering*. Berlin, Germany: Springer, 2009, pp. 581–600.

[23] Y. Si, J. Sun, Y. Liu, and T. Wang, ''Improving model checking Stateful timed CSP with non-Zenoness through clock-symmetry reduction,'' in *Formal Methods and Software Engineering*. Berlin, Germany: Springer, 2013, pp. 182–198.

[24] L. Shi, Y. Liu, J. Sun, J. S. Dong, and G. Carvalho, ''An analytical and experimental comparison of CSP extensions and tools,'' in *Formal Methods and Software Engineering*. Berlin, Germany: Springer, 2012, pp. 381–397.

[25] J. Sun, Y. Liu, J. S. Dong, and J. Pang, ''PAT: Towards flexible verification under fairness,'' in *Proc. Int. Conf. Comput. Aided Verification*, 2009, pp. 709–714.

[26] Y. Liu, W. Chen, Y. A. Liu, S. J. Zhang, J. Sun, and J. S. Dong, ''Verifying linearizability via optimized refinement checking,'' *IEEE Trans. Softw. Eng.*, vol. 39, no. 7, pp. 1018–1039, Jul. 2013.

[27] T. Wang *et al.*, ''A systematic study on explicit-state non-Zenoness checking for timed automata,'' *IEEE Trans. Softw. Eng.*, vol. 41, no. 1, pp. 3–18, Jan. 2015.

[28] H. Hansen, S.-W. Lin, Y. Liu, T. K. Nguyen, and J. Sun, "Diamonds are a girl's best friend: Partial order reduction for timed automata with abstractions," in *Computer Aided Verification*. New York, NY, USA: Springer-Verlag, 2014, pp. 391–406.

[29] T. K. Nguyen, J. Sun, Y. Liu, J. S. Dong, and Y. Liu, "Improved BDD-based discrete analysis of timed systems," in *Proc. 18th Int. Symp. Formal Methods*, Paris, France, Aug. 2012, pp. 326–340.

[30] H. Garavel *et al.*, "CADP 2011: A toolbox for the construction and analysis of distributed processes," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 2, pp. 89–107, 2013.

[31] Y. Bao, M. Chen, Q. Zhu, T. Wei, F. Mallet, and T. Zhou, "Quantitative performance evaluation of uncertainty-aware hybrid AADL designs using statistical model checking," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, to be published.

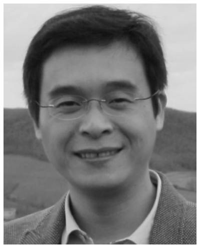
[32] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, "Automated verification of AADL-specifications using UPPAAL," in *Proc. IEEE 14th Int. Symp. High-Assurance Syst. Eng. (HASE)*, Oct. 2012, pp. 130–138.

[33] J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikučionis, and P. Olsen, "Formal analysis and testing of real-time automotive systems using UPPAAL tools," in *Proc. Int. Workshop Formal Methods Ind. Critical Syst.*, 2015, pp. 47–61.

[34] L. T. Nguyen, M. E. Ogburn, W. P. Gilbert, K. P. Kibler, P. W. Brown, and P. L. Deal, "Simulator study of stall/post-stall characteristics of a fighter airplane with relaxed longitudinal static stability," NASA, Hampton, VA, USA, Tech. Rep. NASA-TP-1538, 1979.

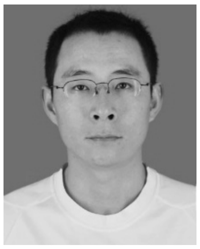
[35] An Integrated. (2003). *Multi-Layer Approach to Software-Enabled Control: Mission Planning to Vehicle Control. DAPAR*. Accessed: Nov. 2017. [Online]. Available: https://www.aem.umn.edu/people/faculty/balas/darpa_sec/SEC.Software.html

[36] Xtend. (2017). *Eclipse*. Accessed: Nov. 2017. [Online]. Available: http://www.eclipse.org/xtend/



YONGWANG ZHAO received the Ph.D. degree in computer science from Beihang University in Beijing, China, in 2009. He is currently an Associate Professor with the School of Computer Science and Engineering, Beihang Univerisity. He has also been a Research Fellow with the School of Computer Science and Engineering, Nanyang Technological University of Singapore from 2015 to 2016. His research interests include formal methods, OS kernels, information-flow security, and AADL.

DIANFU MA received the Ph.D. degree in computer science from Beiheng University, China. He is currently a Professor with Beihang University. He was the Executive Director of Chinese Computer Federation. He has published over 50 academic papers in international journals or conferences. His research interesting includes services computing, real-time systems and high dependable software. He received the 3rd prize of Science and Technology Innovation Award from the Ministry of Education of China in 2003, and first prize of Science and Technology Innovation Award of Beijing in 2011.



FENG ZHANG is currently pursuing the Ph.D. degree in computer science from the School of Computer Science and Engineering, Beihang University, Beijing, China. His research interests include formal methods, model-based methods, and OS kernels.

WENSHENG NIU received the Ph.D. degree from Xi'an Jiaotong University. He is a Senior Research Fellow with the China Aeronautics Computing Technique Research Institute, Xi'an, China. He is also a Professor with the School of Computer Science and Engineering, Beihang Univerisity. His research interests include safety-critical systems, embedded systems, and network security.