软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(2):202−222 [doi: 10.13328/j.cnki.jos.004776]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于时间抽象状态机的 AADL 模型验证*

杨志斌[1,2], 胡 凯[2], 赵永望[2], 马殿富[2], Jean-Paul BODEVEIX[1]

[1](Institut de Recherche en Informatique de Toulouse, Université de Toulouse, Toulouse, France)

[2](北京航空航天大学 计算机学院,北京 100191)

通讯作者: 胡凯, E-mail: hukai@buaa.edu.cn

摘 要: 提出了一种基于时间抽象状态机(timed abstract state machine,简称 TASM)的 AADL(architecture analysis and design language)模型验证方法.分别给出了 AADL 子集和 TASM 的抽象语法,并基于语义函数和类 ML 的元语言形式定义转换规则.在此基础上,基于 AADL 开源建模环境 OSATE(open source AADL tool environment)设计并实现了 AADL 模型验证与分析工具 AADL2TASM,并基于航天器导航、制导与控制系统(guidance, navigation and control)进行了实例性验证.

关键词: AADL(architecture analysis and design language);TASM(timed abstract state machine);模型转换;形式验证

中图法分类号: TP311

中文引用格式: 杨志斌,胡凯,赵永望,马殿富,Bodeveix JP.基于时间抽象状态机的 AADL 模型验证.软件学报,2015,26(2):202−222. http://www.jos.org.cn/1000-9825/4776.htm

英文引用格式: Yang ZB, Hu K, Zhao YW, Ma DF, Bodeveix JP. Verification of AADL models with timed abstract state machines. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):202−222 (in Chinese). http://www.jos.org.cn/1000-9825/4776.htm

Verification of AADL Models with Timed Abstract State Machines

YANG Zhi-Bin[1,2], HU Kai[2], ZHAO Yong-Wang[2], MA Dian-Fu[2], Jean-Paul BODEVEIX[1]

[1](Institut de Recherche en Informatique de Toulouse, Université de Toulouse, Toulouse, France)

[2](School of Computer Science and Engineering, BeiHang University, Beijing 100191, China)

Abstract: This paper presents a formal verification method for AADL (architecture analysis and design language) models by TASM (timed abstract state machine) translation. The abstract syntax of the chosen subset of AADL and of TASM are given. The translation rules are defined clearly by the semantic functions expressed in a ML-like language. Furthermore, the translation is implemented in the model transformation tool AADL2TASM, which provides model checking and simulation for AADL models. Finally, a case study of space GNC (guidance, navigation and control) system is provided.

Key words: AADL (architecture analysis and design language); TASM (timed abstract state machine); model transformation; formal verification

广泛应用于航空、航天、武器装备、核能、汽车控制等关键信息领域的复杂嵌入式实时系统被称为安全攸关实时系统(safety-critical real-time systems).由于功能和非功能要求不断提高,系统复杂度急剧增加,如何设计与实现高质量的安全攸关实时系统,并有效控制开发时间和成本,是学术界和工业界共同面临的难题.近年来,模型驱动开发方法逐渐成为安全攸关实时系统设计与开发的重要手段[1],而作为复杂嵌入式实时系统的体系结构设计与分析语言标准,AADL(architecture analysis and design language)[2,3]日益受到关注,并逐渐发展成为一个新的研究热点.

AADL 语言提供了一种标准、全面的方式,对安全攸关实时系统的软/硬件体系结构、运行时环境、功能

* 基金项目: 法国航空航天基金; 软件开发环境国家重点实验室自主课题(SKLSDE-2014ZX-09, SKLSDE-2013ZX-30)
收稿时间: 2014-06-30; 修改时间: 2014-10-31; 定稿时间: 2014-11-26

以及非功能属性进行表达.目前,模型转换是 AADL 模型形式验证与分析的主要途径,如转换到 BIP(behavior interaction priority)[4]、Fiacre[5]、动作时序逻辑 TLA+[6]、同步语言 Signal[7]等,目的是为了重用这些模型上已有的验证和分析能力.

AADL 语言为安全攸关实时系统提供了丰富的表达能力,但其大部分语义仍然采用自然语言和例子进行解释.已有的 AADL 语义研究主要采用转换语义(translational semantics)的方式[8,9],即,用另一种形式语言(称为目标语言)来表达 AADL 语言的语义,如 BIP[4],Fiacre[5],TLA+[6],Signal[7],IF[10],RTMaude[11],ACSR[12],Lustre[13],Petri Net[14]等,这种方式使得语义更易于理解,也是 AADL 模型转换的基础.但是,大部分已有研究都针对较小的 AADL 子集,没有给出一个比较完整的形式语义定义;其次,主要采用自然语言作为元语言环境,关注的 AADL 子集和目标语言都没有给出精确定义,转换规则也不够形式化;最后,资源利用正确性是安全攸关实时系统的一个重要性质,各种目标语言具有不同的表达能力,但能够显式表达资源行为的语言较少.

抽象状态机(abstract state machine,简称 ASM)[15]广泛应用于软、硬件系统设计以及形式语义定义.为了支持实时系统的功能行为、时间属性以及资源消耗的描述,2006 年,MIT 在 ASM 的基础上提出了时间抽象状态机(timed abstract state machine,简称 TASM)[16,17].相对于 BIP,Fiacre,TLA+,Signal 等目标语言,TASM 能够同时支持功能、时间以及资源等行为的验证和分析,而且可读性较好.

因此,本文提出一种基于 TASM 的 AADL 形式转换语义,总体思路如下:

(1) 针对一个较为完整的 AADL 子集,形式地给出子集的抽象语法;

(2) 形式地给出 TASM 的抽象语法;

(3) 基于语义函数清晰刻画 AADL 语言到 TASM 语言的整体映射关系,并基于类 ML[18]的元语言形式定义转换语义规则.

同时,在形式转换语义的基础上,基于 AADL 的开源建模环境 OSATE[19],设计并实现 AADL 模型验证与分析工具——AADL2TASM,以支持通过模型检测工具 UPPAAL[20]以及仿真分析工具 TASM ToolSet[21]对 AADL 模型进行验证和分析.

本文第 1 节简要介绍 AADL 语言的基本概念.第 2 节给出本文 AADL 子集的抽象语法.第 3 节给出 TASM 语言的基本概念及其抽象语法.第 4 节详细介绍形式转换语义.第 5 节简要介绍 AADL2TASM 模型转换工具.第 6 节结合航天器导航、制导与控制系统实例,介绍 AADL2TASM 工具的应用情况.第 7 节给出相关工作比较.第 8 节是本文的总结,并讨论模型转换正确性问题.

1 AADL 语言简介

安全攸关实时系统是应用软件、运行时环境(虚拟机或操作系统)以及硬件平台深度融合的复杂系统,AADL 语言与之对应地提供了软件体系结构、运行时环境以及硬件体系结构的建模概念:通过线程(thread)、线程组(thread group)、进程(process)、数据(data)、子程序(subprogram)等构件以及连接来描述系统的软件体系结构;通过处理器(processor)、虚拟处理器(virtual processor)、存储器(memory)、外设(device)、总线(bus)、虚拟总线(virtual bus)等构件以及连接来描述系统的硬件体系结构;通过分发协议(dispatch)、通信协议(communication)、调度策略(scheduling)、模式变换协议(mode change)以及分区机制(partition)等属性来描述系统的运行时环境,在 AADL 语言中称为执行模型(execution model);最后,通过系统(system)构件进行组合,层次化地建立系统的体系结构模型.另外,行为附件(behavior annex)[22]以变迁系统(transition system)的形式增强了 AADL 对线程构件和子程序构件功能行为的详细描述能力,而且行为附件与执行模型有着紧密的联系,即,执行模型定义了线程和子程序周期性(非周期性或偶发)地读取、计算和发送数据,行为附件则是对计算状态内的执行行为进行详细刻画.因此,软/硬件体系结构、执行模型以及行为附件构成一个完整的 AADL 描述.

AADL 还包括其他方面的扩展,进一步丰富了 AADL 语言的表达能力.例如,故障模型附件(error model annex)[23]扩展了构件和连接的故障事件、故障概率等非功能属性,以支持系统可靠性分析;为了支持 ARINC653 标准,发布了 ARINC653 扩展附件[24].AADL 语言的基本概念可进一步参见文献[2,3,25].

2 AADL 子集及其抽象语法

从 2004 年 AADL 标准正式发布以来,其核心文档及扩展附件在不断的修订和发展.例如,目前核心文档有 AADL V1.0 (AS5506),AADL V2.0 (AS5506A)和 AADL V2.1 (AS5506B)这 3 个版本,AADL V2.2 也正在修订当中;扩展附件则包括 Annex AS5506/1,Annex AS5506/2,Annex AS5506/3 等系列文档,正在制定当中的 constraint annex[26]和 BLESS annex[27]将发布为 Annex AS5506/4.AADL 已成为一个庞大的标准体系,因此,AADL 语义形式化都是选择一个恰当的子集来进行的.

依据航天器等安全攸关实时系统的部分特征,本文选取的 AADL 子集如下:

(1) 体系结构方面,包括系统构件、进程构件、线程构件、处理器构件、存储器构件以及总线构件等结构元素;

(2) 执行模型方面,包括进程执行、模式变换、线程分发、线程执行、端口通信、调度、资源共享等执行模型属性;

(3) 扩展附件方面,主要包括行为附件.

该子集基本上能构成一个比较完整的 AADL 描述:按照系统、进程、线程以及行为附件这 4 个层次来建立系统的 AADL 模型,并将进程映射到处理器构件、连接映射到总线构件、数据映射到存储器构件,系统构件和进程构件可以定义模式及模式变换,而线程执行会受到进程加载、模式变换、分发、调度的影响,线程通信也可能受到模式变换的影响.由于 AADL 对执行模型有严格的定义,这使得整个系统的行为是确定并可预见的.

我们基于类型的方式给出 AADL 子集的抽象语法.在抽象语法表示中,结构元素和执行模型属性表达在同一个语法结构当中.其中,PORT,SOM,EVENT,DURATION,BASTATE,EXPRESSION 以及 VALUE 为预定义类型.同时,在系统构件中,我们仅考虑单处理器,并且默认将进程构件映射到处理器构件.

Type System:=
{ Iports: set of PORT;
 Oports: set of PORT;
Processes: set of Process;
 CPU: Processor;
 Memories: set of Memory;
 Buses: set of Bus;
 Initial_Mode: *SOM*;
 Mode_Transitions: set of *SOM_Transition*;
}

Type Process:=
{ Iports: set of PORT;
 Oports: set of PORT;
Threads: set of Thread;
 Connections: set of Connection;
 Initial_Mode: *SOM*;
 Mode_Transitions: set of *SOM_Transition*;
}

Type Processor:=
{ Scheduling: {*FixedTimeline*,*Cooperative*,*RMS*};
}

Type *SOM_Transition*:=
{ *SourceMode*: *SOM*;
 TransitionType: {*emergency*,*planned*};
 DestinationMode: *SOM*;
 Event: EVENT;
}

Type Thread:=
{ Iports: set of PORT;
 Oports: set of PORT;
Period: DURATION;
BCET: DURATION;
 WCET: DURATION;
 Deadline: DURATION;
 DispatchType: {*periodic*,*aperiodic*,*sporadic*};
Behavior: *BehaviorAnnex*;
 Modes: set of *SOM*;
}

Type Connection:=
{ *SourcePort*: PORT;
 ConnectionType: {*immediate*,*delayed*};
 DestinationPort: PORT;
 Modes: set of (*SOM*∪*SOM_Transition*);
}

Type *BehaviorAnnex*:=
{ States: set of BASTATE;
Transitions: set of *BA_Transition*;
}

Type B*A_Transition*:=
{ *SourceState*: BASTATE;
 DestinationState: BASTATE;
 Time: DURATION;
 Guard: EXPRESSION;
 Action: (*Iports*(*th*)→*VALUE*)×*Oports*(*th*)→*VALUE*;
}

3 TASM 及其抽象语法

3.1 TASM基本概念

一个TASM描述由两部分组成:环境和抽象机.环境包括环境变量及其类型,而基于变量的当前取值,抽象机进行计算,然后更新变量的取值,以实现系统的状态变迁.同时,TASM 在变迁上增加了时间和资源的定义,分别表示变迁的持续时间和资源消耗.

定义 1. *TASMSPEC*=⟨*E*,*ASM*⟩,为一个二元组:

(1) 环境 *E*=⟨*EV*,*TU*,*ER*⟩,*EV* 表示环境变量,其类型定义在 *TU* 中,主要包括整数类型、Boolean 类型、实数类型以及用户自定义类型;*ER*={(*rn*,*rs*)}表示资源环境变量的定义,如处理器、存储器、带宽、功耗等,*rn* 是资源的名称,*rs*=[*lower*,*upper*]则表示资源的大小.

(2) 抽象机 *ASM*=⟨*MV*,*CV*,*IV*,*R*⟩,监控变量 *MV*(monitored variables)为影响抽象机执行的变量的集合;受控变量 *CV*(controlled variables)为抽象机将要更新的变量的集合;内部变量 *IV*(internal variables)为抽象机内部使用的中间变量,不受环境的影响;*R*=⟨*n*,*t*,*RR*,*r*⟩为抽象机的执行规则,*n* 是规则的名称,*t* 表示规则执行的持续时间,可以是一个固定值,或一个区间[$t_{\min}$,$t_{\max}$],也可以是关键字 *next*(*t*:=*next* 表示机器处于等待状态,直到某个事件发生),*RR* 是型如 *rn*:=*rs* 的资源消耗,*r* 则是型如"**if** *Condition* **then** *Action*"的规则集合,其中,*Condition* 是监控变量的当前取值,*Action* 包括受控变量的赋值、skip 以及通信等,也支持"**else then** *Action*"规则.

ASM 可以有多条规则,我们用 $R_1,R_2,\ldots,R_n$ 表示,但规则之间是互斥执行的,即,每次只有一条规则满足条件并执行(假设为 R_i).R_i 执行完成之后,需要对环境变量的取值、时间以及资源消耗进行更新.因此,ASM 的执行可以表示为一个更新集序列.

除了时间和资源行为以外,TASM 还支持并发组合、层次组合以及同步通信等行为的描述.为此,TASM 将抽象机分为 3 类:主抽象机(main ASM)、子抽象机(sub ASM)和函数抽象机(function ASM).同步通信主要定义在主抽象机之间,并可以采用共享变量和通道两种方式.

此外,TASM 使用模型检测工具 UPPAAL[20]以及仿真分析工具 TASM ToolSet[21]对无死锁性、安全性、活性、时间正确性、资源利用正确性等关键性质进行验证和分析.

3.2 TASM抽象语法

本文仅采用共享变量的通信方式,因此,TASM 的抽象语法表示如下:

$$P ::= x := \exp \mid skip \mid \text{if } BExpr \text{ then } P \mid \text{else then } P \mid time(t_{\min},t_{\max}) \triangleright P \mid time\ next \triangleright P$$
$$\mid resource\ r(r_{\min},r_{\max}) \triangleright P \mid P \oplus P \mid P \otimes P$$
$$TASM ::= \langle E, P \parallel P \parallel \ldots \parallel P \rangle$$

P 定义了单个抽象机的行为,*x*:=exp 表示更新受控变量 *x* 的取值;*skip* 表示不作任何动作;*time* 表示规则执行的时间;*resource* 表示规则执行期间所消耗的资源,*r* 是资源的名称;*P*⊕*P* 表示同一个抽象机器不同规则之间的选择操作;*P*⊗*P* 表示同一条规则内不同动作之间的并发操作,但不能同时去更新同一个变量的取值;*P*||*P* 表示多个抽象机器之间的并发操作,它们共享相同的环境 *E*.

P 的递归定义可以抽象表示子抽象机和函数抽象机的调用.

单个 TASM 抽象机的执行语义是一个循环:依据环境变量的当前取值,选择一条满足条件的规则,等待规则的持续时间,并消耗资源;持续时间完成后,更新环境变量的取值,如果存在同步,则需要等待;该规则执行完之后,选择下一条规则继续执行.多个并发抽象机的语义则要考虑更新集的组合.

4 基于 TASM 的 AADL 形式转换语义

基于 AADL 子集的抽象语法和 TASM 的抽象语法,我们以"层次化、模块化"的方式给出转换语义定义.

4.1 系统构件

系统构件的语义是由其他建模元素的语义组合而成,我们用一个全局语义函数来表示:

$$
\begin{aligned}
&Translate(s:System)=\\
&\langle\\
&\quad \bigcup_{pr\in s.Processes} Trans\_ProcessData(pr)\cup\Big(\bigcup_{tr\in s.Mode\_transitions} Trans\_ModeData(tr)\Big)\cup\\
&\quad \Big(\bigcup_{sch\in s.Processor} Trans\_SchedulerData(sch)\Big)\cup\Big(\bigcup_{th\in pr.Threads} Trans\_ThreadData(th)\Big)\cup\\
&\quad \Big(\bigcup_{cn\in pr.Connections} Trans\_ConnectionData(cn)\Big)\cup\Big(\bigcup_{th\in pr.Threads} Trans\_BehaviorAnnexData(th)\Big),\\
&\quad \|_{pr\in s.Processes} Trans\_Process(pr)\,\|\,\Big(\|_{tr\in s.Mode\_transitions} Trans\_Mode(tr)\Big)\,\|\\
&\quad \Big(\|_{sch\in s.Processor} Trans\_Scheduler(sch)\Big)\,\|\,\Big(\|_{th\in pr.Threads} Trans\_Thread(th)\Big)\,\|\\
&\quad \Big(\|_{cn\in pr.Connections} Trans\_Connection(cn)\Big)\,\|\,\Big(\|_{th\in pr.Threads} Trans\_BehaviorAnnex(th)\Big)\\
&\rangle
\end{aligned}
$$

该语义函数分为 3 个层次:系统构件的子构件层次、进程构件的子构件层次以及行为附件层次,每个层次都包括两部分:语法结构(软/硬件构件的结构元素、执行模型属性以及行为附件的结构元素)映射为 TASM 环境变量,动态行为(执行语义以及行为附件的语义)映射为 TASM 抽象机.我们将使用类 ML 的元语言来表示映射和组合关系:

- **LET** *name* = *P* **AND** ... **IN** $\overset{P}{TASM}$
- **IF** *condition* **THEN** $\overset{P}{TASM}$ **ELSE** $\overset{P}{TASM}$

4.2 进程构件

进程构件代表系统的虚拟地址空间,当对应的处理器启动之后,进程将要执行的二进制镜像文件加载到虚拟地址空间,包括文件加载和进程启动(即初始化)两个步骤,而且整个过程可能会出错或被终止.

首先,将进程构件的输入/输出端口、时间属性以及进程状态转换为 TASM 环境变量.其中,数据端口用 *Integer* 变量表示,事件端口用 *Boolean* 变量表示,事件数据端口用两个变量(*Integer*,*Boolean*)来表示.进程定义的加载时间和启动时间,可以作为 TASM 规则的执行时间.

语义规则 4.2.1. AADL 进程构件的 TASM 环境变量表示.

***Trans_ProcessData*(*pr*)=**

{ *State*: {*unloaded*,*loading*,*starting*,*loaded*}:=*unloaded*;
Iport: *Iports*(*pr*)→*Integer*;
Oport: *Oports*(*pr*)→*Integer*;
LoadTime: *Integer*;
StartupTime: *Integer*;
...
}

其次,进程构件的执行语义转换为一个包含 7 条执行规则(Loading Begin,Loading Complete,Loading Abort, Starting Complete,Starting Abort,Process Stop,Process Normal)的 TASM 抽象机.TASM 可以很方便地在每个时刻或时段,用一条规则来表示相应的行为.共享变量:

- *Started*(*processor*)表示处理器是否启动成功,默认 *Started*(*processor*)=true;
- *Abort*(*processor*)表示处理器是否被中断;
- *Load*(*th*)表示该进程中的线程是否加载成功;
- *Stop*(*processor*)表示处理器是否停止,如果 *Stop*(*processor*)=false,进程进入 Process Normal 规则,且一直

处于 loaded 状态.

这些共享变量也定义在 TASM 环境中.

语义规则 4.2.2. AADL 进程构件执行语义的 TASM 表示.

$$
\begin{aligned}
&\textbf{\textit{Trans\_Process}}(\textbf{\textit{pr}}) =\\
&\textbf{LET } \mathit{TASM\_Process}(pr) =\\
&\quad // \text{ Rule Loading Begin}\\
&\mathit{time}\ 0 \triangleright (\text{if } \mathit{State}(pr) = \mathit{unloaded} \text{ and } \mathit{Started}(\mathit{processor}) = \text{true then}\\
&\qquad \mathit{State}(pr) := \mathit{loading}\ )\\
&\oplus \ // \text{ Rule Loading Complete}\\
&\mathit{time}\ \mathit{LoadTime} \triangleright (\text{if } \mathit{State}(pr) = \mathit{loading} \text{ and } \mathit{Abort}(\mathit{processor}) = \text{false then}\\
&\qquad \mathit{State}(pr) := \mathit{starting})\\
&\oplus \ // \text{ Rule Loading Abort}\\
&\mathit{time}\ 0 \triangleright (\text{if } \mathit{State}(pr) = \mathit{loading} \text{ and } \mathit{Abort}(\mathit{processor}) = \text{true then}\\
&\qquad \mathit{State}(pr) := \mathit{unloaded}\ )\\
&\oplus \ // \text{ Rule Starting Complete}\\
&\mathit{time}\ \mathit{StartupTime} \triangleright (\text{if } \mathit{State}(pr) = \mathit{starting} \text{ and } \mathit{Abort}(\mathit{processor}) = \text{false then}\\
&\qquad \mathit{State}(pr) := \mathit{loaded} \otimes (\underset{th \in pr.Threads}{\otimes} \mathit{Load}(th) := \text{true}))\\
&\oplus \ // \text{ Rule Starting Abort}\\
&\mathit{time}\ 0 \triangleright (\text{if } \mathit{State}(pr) = \mathit{starting} \text{ and } \mathit{Abort}(\mathit{processor}) = \text{true then}\\
&\qquad \mathit{State}(pr) := \mathit{unloaded})\\
&\oplus \ // \text{ Rule Process Stop}\\
&\mathit{time}\ 0 \triangleright (\text{if } \mathit{State}(pr) = \mathit{loaded} \text{ and } \mathit{Stop}(\mathit{processor}) = \text{true then}\\
&\qquad \mathit{State}(pr) := \mathit{unloaded})\\
&\oplus \ // \text{ Rule Process Normal}\\
&\mathit{time}\ \mathit{next} \triangleright (\text{else then}\\
&\qquad \mathit{skip})\\
&\text{IN } \mathit{TASM\_Process}(pr)
\end{aligned}
$$

4.3 模式变换

在系统构件和进程构件中都可以定义模式变换,对应各自子构件和连接的配置的变化.我们重点考虑模式变换对线程执行和通信的影响,而且模式变换发生在所属的进程构件启动之后.图 1 给出了系统模式变换的执行状态和动作.

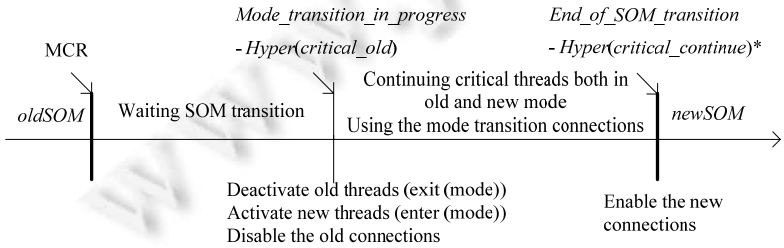

Fig.1 Execution states and actions of an AADL SOM transition

图 1 AADL 系统模式变换的执行状态和动作

系统开始处于旧模式(*oldSOM*)下,模式变换请求(mode change request,简称 MCR)到达之后,系统进入一个

模式变换的准备期,如果模式变换的类型为紧急变换(*Mode_Transition_Response=Emergency*),则此准备期的持续时间为 0;如果模式变换的类型为计划变换(*Mode_Transition_Response=Planned*),则此准备期将持续到旧模式下的关键线程的超周期(记为 *Hyper*(*critical_old*)),我们称为同步点(synchronization point),而且此时间为绝对时间.在此期间,旧模式下的线程将继续执行,这是为了保证模式变换期间线程执行和通信的确定性.在安全攸关实时系统中,任务的关键程度说明了任务的重要性,AADL 用"*Synchronized_Component*"属性为真的周期性线程来表示关键线程.

在此同步点上,系统进入 *Mode_transition_in_progress* 状态,并正式开始模式变换:添加新模式(*newSOM*)下且不属于旧模式(*oldSOM*)的线程;删除旧模式下且不属于新模式的线程和连接;同时属于两个模式的关键线程继续执行;同时属于两个模式的连接可以用于通信.完成这些动作之后,系统才正式进入新模式,并激活新模式下的连接.如果模式变换的类型为紧急变换,则模式变换的持续时间为 0;如果模式变换的类型为计划变换,则模式变换将持续到在此期间继续执行的关键线程的超周期(即,同时属于两个模式的关键线程),并且可能为多个超周期,我们记为 *Hyper*(*critical_continue*)*.

系统每次只响应一个模式变换请求,在模式变换过程中,其他 MCR 都会被忽略,而且不考虑多个 MCR 的优先级.

在 TASM 环境变量中,我们引入"*CurrentSOM*","*ModeTransitionInProgress*","*ArriveHyperPeriod*"等变量.

语义规则 4.3.1. AADL 系统模式变换的 TASM 环境变量表示.

***Trans_ModeData*(*tr*)=**

{ *CurrentSOM*: *SOM*;
ArriveHyperPeriod: *Boolean*;
ModeTransitionInProgress: *Boolean*;
SOMRequest: *Boolean*;
…
}

其执行语义转换为 *TASM_SOM_Transition* 和 MCR 两个抽象机,分别对应模式变换的行为和模式变换请求的产生.前者包括 Waiting Hyper Period,Mode Transition In Progress,Mode Transition 以及 Waiting MCR 等 4 条规则,并通过共享变量 *Activation*(*th*)和 *Activation*(*cn*)来表示线程和连接是否处于当前模式.对于计划变换,由于同步点 *Hyper*(*critical_old*)是绝对时间,因此用 Waiting Hyper Period 规则来表示超周期的到达,并在这个时刻进入 *Mode_transition_in_progress* 状态;对于立即变换,不需要同步点,而是立即进入 *Mode_transition_in_progress* 状态,并立即完成模式变换.MCR 抽象机包含两条规则,并用[0,*Hyper*(*critical_old*)]来表示随机产生 MCR 的时间.由于 *Hyper*(*critical_continue*)*可能为多个超周期,即,从 *Mode_transition_in_progress* 状态到 *End_of_SOM_transition* 状态的持续时间不确定,因此我们使用另一个抽象机 *Manage_Hyper_Continue* 来管理这个时间.

语义规则 4.3.2. AADL 系统模式变换执行语义的 TASM 表示.

***Trans _ Mode* (*tr*) =**
LET *oldSOM* = *tr.SourceMode*
AND *newSOM* = *tr.DestinationMode*
AND *TASM _SOM _Transition* =
// Rule Waiting Hyper Period
IF *tr.TransitionType* = *planned* **THEN**
time Hyper(*critical _old*) ▷ (if *State*(*pr*) = *loaded* and *CurrentSOM* = *oldSOM* and *SOMRequest* = false then
ArriveHyperPeriod := true)
⊕ // Rule Mode Transition In Progress
IF *tr.TransitionType* = *planned* **THEN**

$time\ 0 \triangleright$ (if $CurrentSOM = oldSOM$ and $SOMRequest$ = true and $ArriveHyperPeriod$ = true then

$ModeTransitionInProgress := \text{true} \otimes FlagNewSOM := \text{true} \otimes$

$\underset{th \in threads(newSOM) \setminus threads(oldSOM)}{\otimes} Activation(th) := \text{true} \otimes \underset{th \in threads(oldSOM) \setminus threads(newSOM)}{\otimes} Activation(th) := \text{false} \otimes$

$\underset{cn \in connections(oldSOM) \setminus threads(newSOM)}{\otimes} Activation(cn) := \text{false})$

ELSE

$time\ 0 \triangleright$ (if $CurrentSOM = oldSOM$ and $SOMRequest$ = true then

$ModeTransitionInProgress := \text{true} \otimes \underset{th \in threads(newSOM) \setminus threads(oldSOM)}{\otimes} Activation(th) := \text{true} \otimes$

$\underset{th \in threads(oldSOM) \setminus threads(newSOM)}{\otimes} Activation(th) := \text{false} \otimes \underset{cn \in connections(oldSOM) \setminus threads(newSOM)}{\otimes} Activation(cn) := \text{false})$

$\oplus$ // Rule SOM transition

IF $tr.TransitionType = planned$ **THEN**

$time\ 0 \triangleright$ (if $ModeTransitionInProgress$ = true and $FlagNewSOM$ = false then

$CurrentSOM := newSOM \otimes SOM\,\text{Re}quest := \text{false} \otimes$

$\underset{cn \in connections(newSOM) \setminus threads(oldSOM)}{\otimes} Activation(cn) := \text{true} \otimes$

$ModeTransitionInProgress := \text{false} \otimes ArriveHyperPeriod := \text{false})$

ELSE

$time\ 0 \triangleright$ (if $ModeTransitionInProgress$ = true then

$CurrentSOM := newSOM \otimes \underset{cn \in connections(newSOM) \setminus threads(oldSOM)}{\otimes} Activation(cn) := \text{true} \otimes$

$SOMRequest := \text{false} \otimes ModeTransitionInProgress := \text{false})$

AND $Manage\_Hyper\_Continue =$

$time\ Hyper(critical\_continue) \triangleright$ (if true then

$FlagNewSOM := \text{false})$

AND $MCR =$

// Produce MCR

$time(0, Hyper(critical\_old)) \triangleright$ (if $State(pr) = loaded$ and $CurrentSOM = oldSOM$ and $SOMRequest$ = false then

$SOMRequest := \text{true})$

$\oplus$ // Rule Waiting Next Event

$time\ next \triangleright$ (else $then\ skip$)

IN

IF $tr.TransitionType = planned$ **THEN**

$TASM\_SOM\_Transition \,||\, Manage\_Hyper\_Continue \,||\, MCR$

ELSE

$TASM\_SOM\_Transition \,||\, MCR$

4.4 考虑数据端口通信的线程构件

考虑数据端口通信的线程构件的语义是由线程构件的基本行为以及线程分发和数据端口通信的执行语义构成,表示为语义规则 4.4.1 和语义规则 4.4.2.

语义规则 4.4.1. 考虑数据端口通信的线程构件的 TASM 环境变量表示.

***Trans_ThreadData*(*th*)=**

{*State*: {*halted*,*waiting_mode*,*waiting_dispatch*,*waiting_execution*,*execution*,

completed,*waiting_next_dispatch*}:=*waiting_mode*;

Iport: $Iports(th) \rightarrow Integer$;

Oport: $Oports(th) \rightarrow Integer$;

RscUsage: *RESOURCES*→*Integer*;
WaitingNextDispatch: *Boolean*;
…
}
***Trans_ConnectionData*(*cn*)=**
{ *ConnectionType*: {*immediate*,*delayed*};
…
}

语义规则 4.4.2. 考虑数据端口通信的线程构件执行语义的 TASM 表示.

***Trans _Thread* (*th*) =**
LET *TASM _Thread*(*th*) =
// *Rule Initialization*
time 0 ▷ (if *State*(*th*) = *halted* and *Load*(*th*) = true then
State(*th*) := *waiting _mode*)
// Rule Activation
Time 0 ▷ (if *State*(*th*) = *waiting _mode* and *Activation*(*th*) = true then
State(*th*) := *waiting _dispatch*)
⊕ // Rule Dispatch
Time 0 ▷ (if *State*(*th*) = *waiting _dispatch* and *Dispatch*(*th*) = true then
State(*th*) := *waiting _execution* ⊗ *Iport*(*th*) := *IportBuffer*(*th*) ⊗ *Dispatch*(*th*) := false)
⊕ // Rule Waiting Execution
Time 0 ▷ (if *State*(*th*) = *waiting _execution and Get _CPU*(*th*) = true then
State(*th*) := *execution* ⊗ ***Trans _Connection _Read*(*th*)**)
⊕ // Rule Execution
Time (*BCET*(*th*),*WCET*(*th*)) ▷ *Resource Processor* 100 ▷ (if *State*(*th*) = *execution* then
Oport(*th*) := *ComputeOutPut*(*Iport*(*th*)) ⊗ *State*(*th*) := *completed* ⊗ *Get _CPU*(*th*) := false)
⊕ // Rule Write Data
Time 0 ▷ (if *State*(*th*) = *completed* then
State(*th*) := *waiting _next _dispatch* ⊗ ***Trans _Connection _Write _Imm*(*th*)**)
⊕ // Rule Waiting Next Event
Time next ▷ (else then *skip*)
⊕ // Rule Deactivation
time 0 ▷ (if *State*(*th*) = *waiting _dispatch and Activation*(*th*) = false then
State(*th*) := *waiting _mode*)
AND *Dispatcher*(*th*) =
IF *Dispatch _protocol* = *Periodic* **THEN**
// Rule Dispatch Thread
time 0 ▷ (if *Activation*(*th*) = true and *State*(*th*) = *waiting _dispatch* and *Dispatch*(*th*) = false then
Dispatch(*th*) := true ⊗ *WaitingNextDispatch* := true)
⊕ // Rule Waiting Period
time Period(*th*) ▷ (if *WaitingNextDispatch* = true then
WaitingNextDispatch := false ⊗ ***Trans _Connection _Write _Delay*(*th*)**
⊗ *State*(*th*) := *waiting _mode*)
⊕ // Rule Waiting Next Event

time Next ▷ (else then *skip*)
IF *Dispatch _ protocol* = *Aperiodic* **THEN**
// Rule Dispatch Thread
time [0, Max] ▷ (if *Activation*(*th*) = true and *State*(*th*) = *waiting _ dispatch* and *Dispatch*(*th*) = false then
Dispatch(*th*) := true ⊗ *WaitingNextDispatch* := true)
⊕ // Rule Prepare Next Dispatch
time 0 ▷ (if *State*(*th*) = *completed* and *WaitingNextDispatch* = true then
WaitingNextDispatch := false ⊗ *State*(*th*) := *waiting _ mode*)
⊕ // Rule Waiting Next Event
time next ▷ (else then *skip*)
ELSE // *Sporadic*
// Rule Dispatch Thread
time [*Period*, Max] ▷ (if *Activation*(*th*) = true and *State*(*th*) = *waiting _ dispatch* and *Dispatch*(*th*) = false then
Dispatch(*th*) := true ⊗ *WaitingNextDispatch* := true)
⊕ // Rule Prepare Next Dispatch
time 0 ▷ (if *State*(*th*) = *completed* and *WaitingNextDispatch* = true then
WaitingNextDispatch := false ⊗ *State*(*th*) := *waiting _ mode*)
⊕ // Rule Waiting Next Event
time next ▷ (else then *skip*)
IN *TASM _Thread*(*th*) || *Dispatcher*(*th*)

$$\textbf{\textit{Trans\_Connection\_Read}}(\textbf{\textit{th}}) = \bigotimes_{\substack{ip\in Iports(th)\wedge cn\in Connections\wedge \\ DestinationPort(cn)=ip\wedge ConnectionType(cn)=immediate}} ip := IportBuffer(th)$$

$$\textbf{\textit{Trans\_Connection\_Write\_Imm}}(\textbf{\textit{th}}) = \bigotimes_{\substack{op\in Oports(th)\wedge cn\in Connections\wedge \\ ConnectionType(cn)=immediate\wedge SourcePort(cn)=op}} IportBuffer(DestinationPort(cn)) := op$$

$$\textbf{\textit{Trans\_Connection\_Write\_Delay}}(\textbf{\textit{th}}) = \bigotimes_{\substack{op\in Oports(th)\wedge cn\in Connections\wedge \\ ConnectionType(cn)=delayed\wedge SourcePort(cn)=op}} IportBuffer(DestinationPort(cn)) := op$$

4.4.1 线程构件的基本行为

线程构件表示一个二进制镜像中顺序执行的指令序列,是 AADL 主要的执行和调度单元.任何一个线程构件在执行之前,都需要将其二进制镜像文件加载到进程,加载成功才能处于等待分发状态(waiting dispatch);如果系统存在不同模式,只有处于当前模式的线程才能被分发,否则,线程处于等待模式状态(waiting mode);处于等待分发状态的线程可以被周期性时钟或事件来分发,包括周期、非周期、偶发等多种分发协议;分发后的线程需要等待调度执行(waiting execution);得到处理器的线程进入执行状态,执行的具体行为则可用行为附件(behavior annex)来详细刻画;对于线程间通信,AADL 采用“Input-Compute-Output”的计算模型,默认情况下,在线程分发的时刻读取所有输入端口的数据并等待计算,在计算完成时刻将结果写入输出端口,但不同的通信机制(immediate/delayed)会影响线程的读、写时间.如图 2 所示.

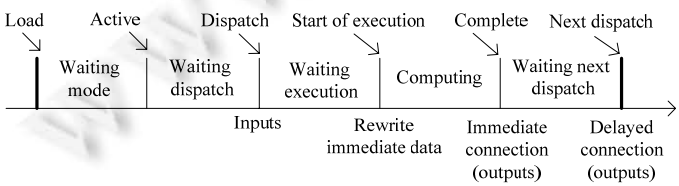

Fig.2 Execution states and actions of an AADL thread component
图 2 AADL 线程构件的执行状态和动作

首先,将线程构件的输入/输出端口、资源利用率(最大为 100%)以及线程状态转换为 TASM 环境变量,并为

每个输入端口定义一个变量 *IportBuffer*,用于缓存数据;其次,将线程构件的执行语义转换为一个包含 8 条执行规则(Initialization,Activation,Dispatch,Waiting Execution,Execution,Write Data,Waiting Next Event,Deactivation)的 TASM 抽象机 *TASM_Thread*(*th*):

- Initialization 规则:用来处理线程和进程加载的关系,通过共享变量 *Load*(*th*):{true,false}来同步,进程构件的执行语义在第 4.2 节已经定义.
- Activation 规则:用来表示线程进入当前模式(Enter(mode)),并通过共享变量 *Activation*(*th*)=true 来同步,模式变换的执行语义在第 4.3 节中已经定义.
- Dispatch 规则:用来处理线程和分发器(dispatcher)的关系,分发器可以周期、非周期或偶发性地分发线程,通过共享变量 *Dispatch*(*th*):{true,false}来同步,并将 *IportBuffer* 中的数据读取到输入端口中.
- Waiting Execution 规则:线程执行会受到通信依赖或调度的影响,我们使用共享变量 *Get_CPU*(*th*):{true,false}来同步,调度器(scheduler)的执行语义将在第 4.5 节中定义.
- Execution 规则:如果没有定义行为附件,此规则的执行时间为[*BCET*,*WCET*],我们用"*ComputeOutPut*"函数来抽象表示其执行,并且可以定义处理器、存储器、总线、功耗的利用率,如果定义了行为附件,则需要对这条规则进行求精,将在第 4.6 节中给出.
- Write Data 规则:执行完成之后,直接将输出端口的数据写入接收线程的 *IportBuffer* 当中.
- Waiting Next Event 规则:用于处理条件不满足而需要等待的情况,例如进程加载失败、不处于当前模式、没有被分发、没有得到处理器资源等,该规则使用了 TASM 非常重要的同步机制:*t*:=*next*.
- Deactivation 规则:用来表示线程退出当前模式(Exit(mode)),并通过共享变量 *Activation*(*th*)=false 来同步,模式变换的执行语义在第 4.3 节中已经定义.

4.4.2 线程分发

在 AADL 中,分发器并不是一个独立构件,而是线程构件的一个执行模型属性,支持周期、非周期、偶发、实时、混成、后台这 6 种分发协议,本文主要考虑常用的前 3 种情况.

线程分发器的执行语义用另一个抽象机 *Dispatcher*(*th*)来表示,并和线程基本行为的抽象机并发执行,它们之间通过共享变量 *Dispatch*(*th*)和线程状态变量 *State*(*th*)来同步.我们用 *t*:=[0,max]表示非周期事件的到达时间,用 *t*:=[*period*,max]表示偶发事件的到达时间.其中,Dispatch Thread 规则表示:

- 如果线程属于当前模式,并且处于等待分发状态,则分发该线程.
- 对于周期性线程,线程被分发之后,分发器需要等待一个周期,再让线程进入下一次分发.由于需要考虑模式变换的影响,因此将线程状态设为等待模式.
- 对于非周期或偶发线程,线程执行完成之后,就可以进入等待模式状态.

4.4.3 数据端口通信

本文主要关注周期性线程之间的数据端口通信.端口通信的基本方式是采样(sampled),但由于并发和抢占的原因,通信行为可能存在不确定性,从而导致整个系统存在延迟、抖动或不稳定.为此,AADL 提供了两种确定性的数据端口通信机制:即时通信(immediate)和延迟通信(delayed).

- 即时通信要求

(1) 发送线程和接收线程都是周期性线程.

(2) 对于分发:如果周期相同(称为 synchronous),要求两个线程同时分发;如果周期不同,当发送线程周期是接收线程周期的倍数(称为 oversampling),即,发送线程分发一次,而接收线程会分发多次,但一般要求发送线程的分发和接收线程的第 1 次分发是同时的;当接收线程周期是发送线程周期的倍数(称为 undersampling),即,发送线程分发多次,而接收线程分发一次,一般要求发送线程的第 1 次分发和接收线程的分发是同时的.

(3) 对于执行:虽然两个线程是同时分发的,但接收线程的真正执行要等到发送线程完成之后才能开始,即,发送线程在其 Complete 时刻将数据写入输出端口,而接收线程需要在此刻重新读取发送线程的

输出数据,并开始执行.

- 延迟通信要求

(1) 发送线程和接收线程都是周期性线程.

(2) 对于分发:存在 Synchronous,Oversampling,Undersampling 这 3 种情况,但不要求两个线程同时分发.

(3) 对于执行:发送线程是在其 Deadline 时刻(一般等于周期)才将数据写入输出端口,而接收线程在其 Dispatch 时刻读取上一个周期发送线程的输出数据,因此不需要重读新数据.

首先,将连接类型等信息映射为 TASM 环境变量;其次,其转换语义是在线程构件以及周期性分发协议的语义之上,进一步扩充通信对端口读写时间的影响,我们将 *Trans_Connection*(*th*)分为 *Trans_Connection_Read*(*th*), *Trans_Connection_Write_Imm*(*th*)以及 *Trans_Connection_Write_Delay*(*th*)),分别对应即时通信的接收线程在开始执行之前重读最新数据、即时通信的发送线程在 Complete 时刻输出数据以及延迟通信的发送线程在 Deadline 时刻输出数据.隐含的线程执行顺序将显式地表示为一个调度器,并在第 4.5 节给出.而且,语义规则是针对任意的周期性线程,Synchronous,Oversampling,Undersampling,Immediate 以及 Delayed 等多种情况都考虑在其中.

4.5 调度器

本文讨论的调度主要是针对单处理器,且为非抢占式调度:

- 首先,两种确定性的数据端口通信方式不仅规定了发送线程和接收线程的端口读写时间,还隐含地给定了线程执行顺序,我们称为静态调度(offline scheduling).静态调度在安全攸关实时系统领域经常使用,即,使用调度表静态地给出执行顺序,而且可以方便地表示任务之间的依赖关系.
- 其次,AADL 支持在处理器构件中显式定义调度策略,包括固定时钟驱动、轮转、RMS、EDF、SporadicServer、SlackServer 以及 ARINC653 等调度策略.我们主要考虑固定时钟驱动、轮转调度以及 RMS 等调度策略,且用于独立任务.依据 AADL 时间参数,可以得到静态的调度表.

首先给出调度器所对应的 TASM 环境变量,变量 CPU 包括 *free* 和 *busy* 两个状态.

语义规则 4.5.1. AADL 线程调度对应的 TASM 环境变量表示.

Trans_SchedulerData=

{ *Deadline*: *Integer*;

CPU:{*free*,*busy*};

…

}

其次给出调度器对应的 TASM 抽象机.按照调度表,我们构造线程执行的固定优先级顺序(这里用令牌 $Token=\{th_1,\ldots th_i,\ldots th_n\}$来表示),以确保多个线程不同时使用处理器.

语义规则 4.5.2. AADL 线程调度执行语义的 TASM 表示.

Trans _ Scheduler =

LET *TASM _ Scheduler* =

...

⊕ // Rule Scheduling Thread *th*

time $0 \triangleright$ (if *State*(*th*) = *waiting _ execution* and *Token* = *th* and *CPU* = *free* then

Get _ CPU(*th*) := true ⊗ *CPU* := *busy*)

...

⊕ // Rule Waiting Next Event

time next $\triangleright$ (else then *skip*)

IN *TASM _ Scheduler*

在基本调度策略的基础上,可以进一步考虑资源共享对调度的影响.线程构件需要增加 Waiting resource 状态,而资源可以表示为带{*idle*,*used*}两个状态的抽象机,并与线程构件执行语义的抽象机并发执行.

4.6 行为附件

AADL 线程构件使用的是“Input-Compute-Output”计算模型,输入、输出行为是由线程的基本执行语义和线程通信语义决定的,而行为附件则是对计算行为进行细化和求精.

图 3 给出了行为附件与线程执行模型的关系:线程执行模型的分发机制将数据写入输入数据端口;行为附件读取这些数据做逻辑判断并计算,计算完成之后,将结果写入输出数据端口.同时,行为附件还可以通过输入事件端口、输入事件数据端口接收事件和数据,如接收中断请求、对非周期性线程的分发条件进行细化等;通过输出事件端口、输出事件数据端口发送事件,如发送中断请求、子程序调用、模式变换请求等.因此,行为附件的执行语义是对线程构件执行语义中“Execution”规则的细化和求精.

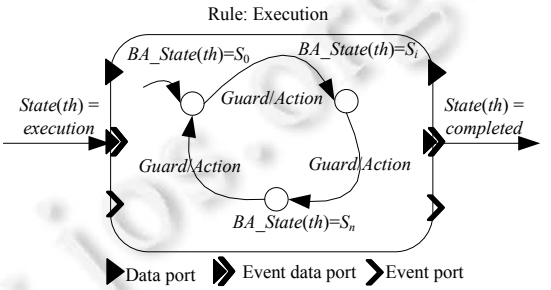

Fig.3 Relation between the behavior annex and execution model

图 3 行为附件与线程执行模型的关系

首先,我们给出基本的转换思路:状态变量(state variables)映射为 TASM 环境变量;状态(state)映射为抽象机的内部状态(internal variables);每条变迁映射为一条 TASM 规则,而规则的条件由当前线程构件的状态(*State*(*th*))、当前行为附件的状态以及行为附件的变迁条件(guard)组成,规则的执行部分则包括行为附件的执行动作(action)以及对行为附件下一个状态的赋值;我们还增加一条规则(behavior annex completion),让线程构件进入下一个状态;最终,用这些规则来替换线程构件执行语义中的“Execution”规则.

其次,还需要对 *Guard* 和 *Action* 的语义进行细化:

Guard 主要包括事件接收(⟨*event*⟩)、端口数据的逻辑判断(⟨*BExpr*⟩),而后者可拆分为两部分:“on ⟨*BExpr*⟩”和“when ⟨*BExpr*⟩”,其表示形式如下:

⟨*guard*⟩::=⟨*BExpr*⟩|[**on** ⟨*BExpr*⟩→] ⟨*event*⟩ [**when** ⟨*BExpr*⟩].

⟨*event*⟩表示从事件端口接收事件(*P*?)或从事件数据端口接收事件和数据(*P*?(*x*)),*P* 为端口名,*x* 为状态变量,我们可以引入 *Boolean* 变量来表示接收事件,*x* 的数据则保存到对应的环境变量当中;⟨*BExpr*⟩是对数据端口或事件数据端口的数据进行逻辑判断,可以映射为对 TASM 环境变量的操作,on ⟨*BExpr*⟩表示对当前状态下的端口数据进行逻辑判断,而 when ⟨*BExpr*⟩表示对将要读取的数据进行逻辑判断.我们将其简化表示为⟦*guard*⟧.

Action 主要包括变量赋值、数据或事件的发送、计算时间、等待时间等.变量赋值可以直接映射为对 TASM 环境变量的赋值;对于数据发送(*P*!(*x*),*P*=*x*)和事件发送(*P*!),可以引入 *Boolean* 变量来表示发送事件,*x* 则保存到对应的环境变量当中;在变迁上定义时间属性,可以更精确地给出执行和等待的时间序列,计算时间(*Computation*(min,max))表示使用 CPU 的时间,等待时间(*Delay*(min,max))表示被挂起或中断的时间,由于行为附件的 *Action* 是作用在变迁上,因此这些时间属性可以直接映射为 TASM 规则的执行时间.

在 TASM 环境变量中,我们引入“*CurrentBAState*”,“*isInitial*”,“*isFinal*”等变量.“*isInitial*”用于判断行为附件的状态是否为初始状态;“*isFinal*”用于判断行为附件的状态是否为完成状态,而其执行语义为线程构件执行语义“Execution”规则的求精.

语义规则 4.6.1. AADL 行为附件的 TASM 环境变量表示.

***Trans_BehaviorAnnexData*(*th*)=**

{ *CurrentBAState*: *BAState*;

isInitial: *BAState*→*Boolean*;

isFinal: *BAState*→*Boolean*;

…}

语义规则 4.6.2. AADL 行为附件执行语义的 TASM 表示.

***Trans _Thread*(*th*) =**

...

⊕ // Rule Execution

***Trans _ BehaviorAnnex*(*th*)**

⊕// *Rule Write Data*

...

***Trans _ BehaviorAnnex*(*th*) =**

$$\bigotimes_{BA=th.Behavior \wedge BA\_tr \in BA.Transitions} Time\ Time(BA\_tr) \triangleright$$

(if $State(th) = execution$ and $CurrentBAState = SourceState(BA\_tr)$ and $[\![Guard(BA\_tr)]\!]$ = true then

$CurrentBAState := DestinationState(BA\_tr) \otimes Oport(th) := Action(BA\_tr)(Iport(th))$

⊕// Rule Behavior Annex Completion

$Time\ 0 \triangleright$ (if $isFinal(CurrentBAState)$ = true then

$State(th) := completed$)

5 AADL2TASM 模型转换工具

在转换语义的基础上,基于 AADL 的开源建模环境 OSATE[19],设计并实现模型转换工具 AADL2TASM,以支持对 AADL 模型进行形式验证和分析.其体系结构如图 4 所示.

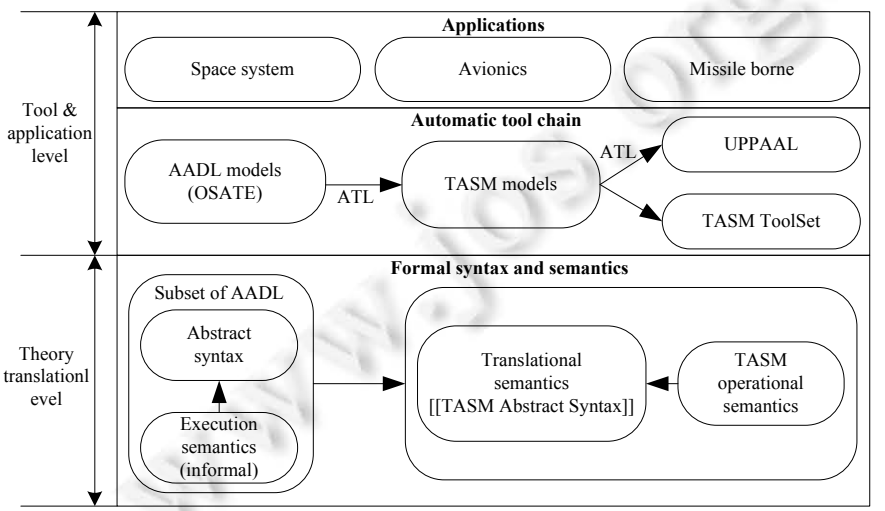

Fig.4 Architecture of the model transformation toolAADL2TASM

图 4 AADL2TASM 模型转换工具的总体架构

(1) 在建模方面,开源建模环境 OSATE 建立在 Eclipse 平台上,支持以插件形式实现 AADL 模型的验证和分析工具,也可以使用 TOPCASED 建模环境.

(2) 在模型转换方面,采用模型转换语言 ATL(atlas transformation language)[28]来实现 AADL 模型到 TASM 模型的自动转换,TASM 使用模型检测工具 UPPAAL 以及仿真分析工具 TASM ToolSet 对模型进行验证和分析.为了将 UPPAAL 也紧密集成到 OSATE 当中,基于 MIT 给出的转换算法[17],采用 ATL 实现 TASM 模型到 UPPAAL 模型的自动转换,并将两个模型转换工具集成为统一的 OSATE 插件,方便用户使用.

(3) 在性质验证与分析方面,TASM ToolSet 支持对模型的完整性和一致性进行验证,以及对时间行为和资源行为进行仿真分析;而 UPPAAL 支持对死锁、安全性、活性以及实时性质进行验证,两者的优势可以互补.

(4) 在系统应用方面,在工业界项目的支持下,我们分别对航天器导航、制导与控制系统、机载飞行控制系统以及弹载飞行控制系统进行了建模和实例性验证.

6 应用实例

6.1 系统介绍

导航、制导与控制系统,即 GNC 系统,是航天器在轨运行的核心保障系统,承担着航天器姿态和轨道确定与控制的重要任务[29].GNC 系统一般由导航传感器、控制计算机和执行机构组成.其中,

- 导航传感器包括导航相机、星敏感器、陀螺、加速度计等,主要用于采集各种数据.
- 控制计算机也称为姿态与轨道控制系统(attitude and orbit control system,简称 AOCS),通过收集和处理各种传感器的测量数据来完成制导和控制任务,主要执行轨道确定、轨道控制、姿态确定、姿态控制等功能,同时负责与其他分系统进行交互,AOCS 一般采用双机备份的方式来提高可靠性.
- 执行机构包括反作用飞轮、喷嘴、轨控发动机等,反作用飞轮和喷嘴用来控制姿态,而轨控发动机则用来执行轨道机动和修正.

导航传感器和控制计算机之间存在一个接口装置,用来对采集的数据进行前期处理,称为数据处理单元(data process unit,简称 DPU),而 DPU 和导航传感器统称为局部终端处理单元(local terminal unit,简称 LTU).其简化的系统体系结构如图 5 所示.

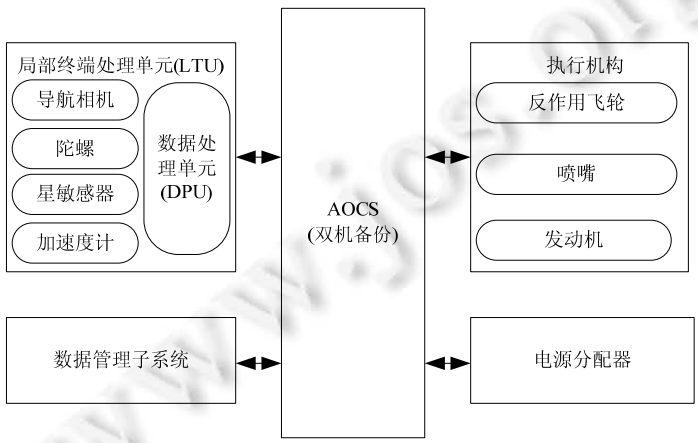

Fig.5 Simplified architecture of the GNC system

图 5 GNC 系统的简化体系结构

6.2 系统建模

我们以 AOCS 子系统为例.AOCS 子系统由一个系统构件来表示,包括进程构件 *S_AOCS_Process*、处理器构件 *AOCS_Proc*、存储器构件 *AOCS_Mem*、总线构件 *AOCS_LAN* 以及 3 个外设构件,它们之间采用总线构件

来通信.同时,GNC 系统涉及星箭分离、速率阻尼、对日捕获、对月捕获、三轴稳定、变轨机动等模式,我们以稳定(stabilization)和机动(maneuver)两个模式为例,如图 6 所示.

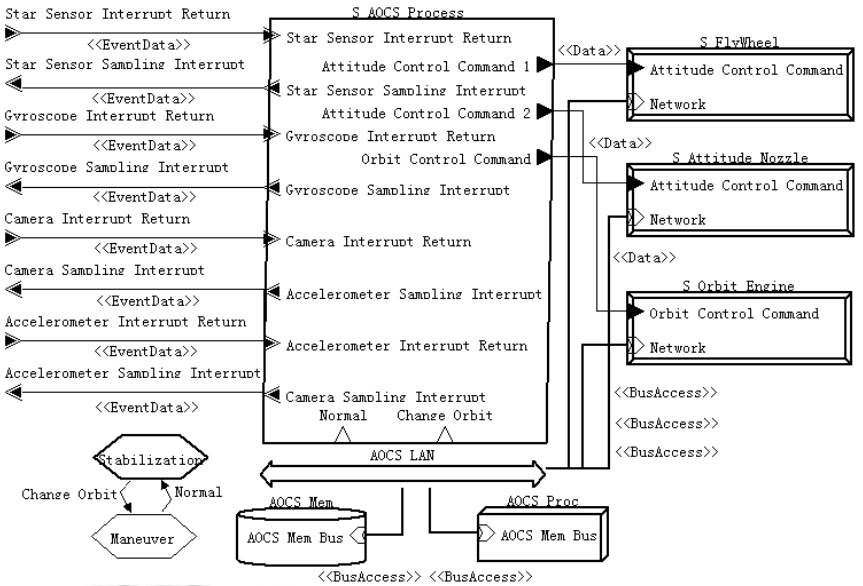

Fig.6 AADL graphic model of the AOCS subsystem

图 6 AOCS 子系统的 AADL 图形化模型

AOCS 子系统涉及 4 种任务的执行:控制任务、遥测遥控任务、系统监测任务以及空闲任务,我们主要考虑控制任务.控制任务用来完成对航天器姿态与轨道的控制,包括发送信息采集指令(*Star_Sensor_Data_Sampling*, *Gyroscope_Data_Sampling*,*Camera_Data_Sampling*,*Accelerometer_Data_Sampling*)、姿态确定(*attitude filter*)、轨道确定(*orbit filter*)、姿态制导 1(*attitude guidance* 1)、姿态控制 1(*attitude control* 1)、姿态制导 2(*attitude guidance* 2)、姿态控制 2(*attitude control* 2)、任务制导律(*guidance law*)等子任务.整个控制任务的周期设为 360ms,因此,这些子任务的周期也设为 360ms,但执行时间不同.在稳定模式下,*Star_Sensor_Data_Sampling*, *Gyroscope_Data_Sampling*,*Attitude Filter*,*Camera_Data_Sampling*,*Orbit Filter*,*Attitude Guidance* 1,*Attitude Control* 1 等子任务按顺序执行;在机动模式下,*Gyroscope_Data_Sampling*,*Attitude Filter*,*Accelerometer_Data_Sampling*,*Orbit Filter*,*Guidance Law*,*Attitude Guidance* 2,*Attitude Control* 2 等子任务按顺序执行.所有子任务均采用周期性线程来描述,任务之间采用的数据端口通信机制为即时通信.表 1 给出了 AOCS 子系统任务的周期、执行时间、资源消耗(功耗、存储)以及所属的模式.

Table 1 Parameters of the tasks of the AOCS subsystem

表 1 AOCS 子系统任务的参数

Thread	Period (ms)	Execution time (ms)	Power (W)	Memory (KB)	Mode
Star_Sensor_Data_Sampling	360	32	[2,10]	256	Stabilization
Gyroscope_Data_Sampling	360	32	[5,20]	256	Stabilization maneuver
Camera_Data_Sampling	360	32	[5,10]	1 024	Stabilization
Accelerometer_Data_Sampling	360	32	[1,3]	256	Maneuver
Attitude_Filter	360	64	[5,10]	1 024	Stabilization maneuver
Orbit_Filter	360	64	[5,10]	1 024	Stabilization maneuver
Attitude_Guidance 1	360	64	[2,10]	512	Stabilization
Attitude_Control 1	360	64	[2,10]	512	Stabilization
Attitude_Guidance 2	360	64	[2,10]	512	Maneuver
Attitude_Control 2	360	64	[2,10]	512	Maneuver
Guidance_Law	360	32	[5,10]	256	Maneuver

6.3 关键性质验证和分析

基于 AADL2TASM 模型转换工具,实现 AADL 模型到 TASM 模型的自动转换.

首先,直接基于 TASM ToolSet 对转换得到的 TASM 模型进行分析.TASM ToolSet 支持对模型的完整性和一致性进行验证,以及对时间行为和资源行为进行仿真分析.图 7 给出了 TASM 模型中所有抽象机的执行时序.

同时,TASM ToolSet 为每种资源给出一个总体资源消耗分析图,由于资源消耗一般定义为一个区间,因此,TASM ToolSet 按照区间的最小值、最大值、平均值以及随机值这 4 种情况对系统的资源消耗情况进行分析,如图 8 所示.

Fig.7 Time simulation of the TASMToolSet

图 7 基于 TASMToolSet 的时间仿真分析

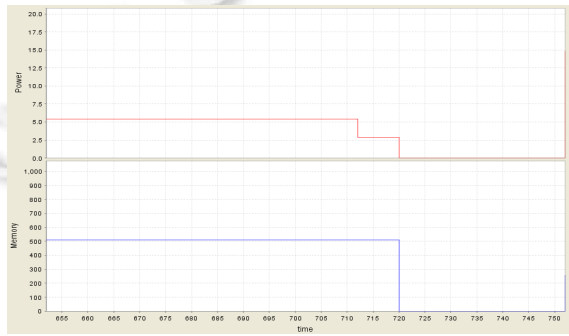

Fig.8 Resource simulation of the TASMToolSet

图 8 基于 TASMToolSet 的资源仿真分析

其次,基于 UPPAAL 对死锁、安全性、活性以及实时性质进行验证,UPPAAL 和 TASM ToolSet 可以互补,共同为 TASM 模型提供丰富的验证与分析能力.我们使用 TASM2UPPAAL 模型转换工具,自动生成对应的时间自动机表示,如图 9 和图 10 所示.

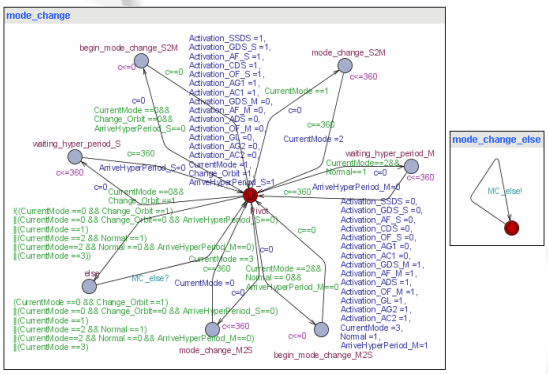

Fig.9 Timed automaton expression of mode change

图 9 模式变换的时间自动机表示

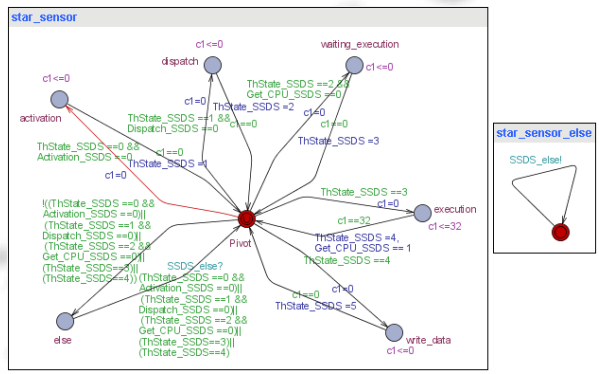

Fig.10 Timed automaton expression of the task of *Star_Sensor_Data_Sampling*

图 10 *Star_Sensor_Data_Sampling* 任务的时间自动机表示

TASM 支持的数据类型有 *Integer*,*Boolean*,*Float* 以及用户自定义类型,而 UPPAAL 只支持 *Integer* 类型,对于用户自定义类型,如线程的状态,设有 n 个成员,在 UPPAAL 中用数组 *int*[0,n−1]来表示,“0”对应第 1 个成员,“n−1”对应第 n 个成员,对于 *Boolean* 类型,用 *int*[0,1]来表示,这里,“0”代表 true,“1”代表 false;TASM 和 UPPAAL 对时间的表达也不同,TASM 的时间定义在变迁上,而 UPPAAL 的时间定义在状态上,因此,对于 TASM 规则上的时间,在 UPPAAL 上通过增加一个中间状态来表示,如图中的 *waiting_hyper_period_S*,*begin_mode_change_S2M*