顺序图与状态图的递归语义一致性研究

周 翔[1,2] 邵志清[1]

(华东理工大学信息科学与工程学院 上海 200237)[1] (青岛大学信息工程学院 青岛 266071)[2]

摘 要 建模面向对象的软件系统是UML的动态图的重要应用,采用顺序图描述消息传送,动态图描述活动。在大型系统开发过程中,往往存在大量顺序图和状态图,由于语义的缺失,导致这些图形表达涵义模糊不清,特别是在递归的信息传送上,由于表达方式的特点,状态图很容易造成实现过程的歧义甚至死锁。提出了基于ASM的多agent实时控制方法,结合形式化的规则定义,通过多个层次agent控制状态的跃迁,保证状态图在描述复杂信息传送时,能够和顺序图的时序保持一致,这对提高系统的可靠性具有一定的现实意义。

关键词 顺序图,状态图,ASM,递归语义

中图法分类号 TP311 **文献标识码** A

Recursive Semantic Consistency of Sequence Diagram and State Diagram

ZHOU Xiang[1,2] SHAO Zhi-qing[1]

(Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China)[1]

(Department of Computer Science and Engineering, Qingdao University, Qingdao 266071, China)[2]

Abstract The dynamic diagrams in the UML are used extensively to model object-oriented software systems, in which sequence diagrams describe the message transfer and state diagrams emphasize the behavior. However, the lack of semantics may result in the confusion of these diagrams that are often used in the development of large systems. In particular, this confusion could lead to the deadlock of state diagrams during the recursive transfer. This paper proposed a solution to this problem using real time multi-agent ASM, combined with formal rules. Specifically, we improved the reliability by using multi-level agent to control the transition so that state diagram is consistent with sequence diagram during the complex message transfer.

Keywords Sequence diagram, State diagram, Abstract state machine, Recursive semantics

1 引言

顺序图、活动图、状态图作为UML动态图的基本组成元素,在动态建模中起到非常重要的作用。顺序图描述了项目中不同对象之间的消息传递、消息间的时序及依赖关系。活动图和状态图则侧重于描述对象的动态行为过程。

UML在它的元模型中对这3种动态图分别进行了约束,但这种规范是半形式化的,即其一部分是通过自然语言描述的,因此造成了动态图在实现上语义不清晰、易歧义的问题。比如顺序图的时序、活动图的并发等问题都容易造成模型与语义的分歧,因此模型的正确性很难得到保证[1-5]。

在现有面向对象建模中,系统往往包含众多交互对象,而在交互过程中不可避免出现消息传递,对于一些交互频繁的对象,相互之间具备一定的依赖关系,其流程的推进往往取决于其他对象的计算结果,因而不可避免地存在相互调用的递归现象。顺序图由于其表达方式的特点,采用对象、时间线的方式能够给出相对直观的时序描述,状态图的建模元素是状态和跃迁,通过事件、警戒条件等刻画消息间的依赖关系,但由于其元模型语义的局限性,容易和系统的实际时序需求相互矛盾,甚至出现死锁。

基于上述问题,本文使用ASM作为形式化工具,采用图形和规则相结合的方式,既保留了UML图形化易读的优点,又通过ASM的规则对状态图的语义进行了有效的形式化补充。在原本模型的基础上,通过ASM的语义规则及实时agent流程控制[6],为状态图提供了清晰的语义描述,使其可以描述递归过程,建立与顺序图时序一致的模型。

2 问题分析

分析以下关于电梯运行的简单实例。

当用户在某层按下电梯按钮请求上楼或下楼时,电梯检测到这个信号并将其传送给控制器,控制器随后需要查询多部电梯分别所处的楼层、正在运行的方向等等状态信息,并综合这些信息进行选择,最后将选择结果返回电梯。

从以上需求,首先得到问题需要的两个对象:elevator是电梯对象,controller是电梯管理器对象。每当楼层的电梯按钮被按下时,elevator将发送press消息给controller,随后

到稿日期:2009-10-28 返修日期:2010-02-25 本文受上海市科技发展基金项目(06dz15004-1863)资助。

周 翔(1977—),女,博士生,讲师,主要研究方向为软件方法学、形式化语义,E-mail:Shannon_sui@163.com;邵志清(1966—),男,教授,博士生导师,主要研究方向为软件开发与验证方法。

controller需要选择最佳的电梯为用户服务，因此choose函数发送请求查询当前电梯的状态，如楼层及运行方向等，再将选择发送给电梯。

图1即为该实例的顺序图模型，简洁地表示了消息之间的传递，其中query为elevator的内置查询模块，choose为controller的内置选择模块。

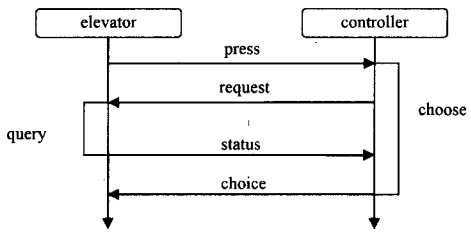

图1 电梯模型的顺序图

接下来绘制该问题对应的状态图，根据UML的语法要求，可以得到如图2所示的状态图。当电梯按钮被按下时，向控制器发送请求选择服务的电梯对象；控制器接收到请求后将请求转发给选择模块，选择模块需要先查询当前所有电梯对象的状态再根据算法选择最优对象为之服务。直观来看，图2和图1是相互对应的，但是根据UML元模型的run-to-completion语义，当状态S1发送choose请求后，S4会转而发送请求查询当前电梯对象的状态，而此时S1处于不稳定状态，不能响应，因此陷入死锁。本文将主要讨论在类似的顺序过程中顺序图与状态图语义的一致性问题，即在包含多个动态图的复杂建模过程中，保证多图在动作时序上的一致，从而更好地避免死锁及歧义。

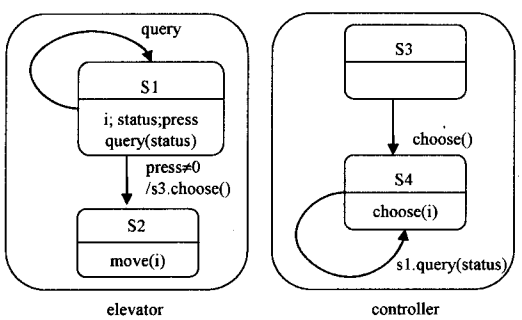

图2 电梯模型的状态图

3 抽象状态自动机ASM方法

以往对UML的这种半形式化语义问题曾展开过大量的工作，主要采用的规范有Z、B、时序逻辑、petri网、自动机等[7,8]。对于Z、B及时序逻辑而言，虽然有效地保证了规范的正确性，但往往要求程序员掌握一定的逻辑知识，具有较高的理论水平，因此增加了实现时的难度，容易在实现阶段产生错误；petri网的优点是可以对并发、同步行为以及资源共享建模，而且便于验证及行为分析，但是不容易绘制petri网图，特别是复杂系统中多个对象及对象间的关系建模时存在困难；自动机方法需要结合多种状态机，如方法状态机、协议状态机，在模型上保持与UML的基本一致，将跃迁和事件分离开，分别用两个状态机进行描述，模型易读性强，但在分离过程中，又会增加保证原有建模信息完整性的额外负担，在大型系统中难以保证建模的效率。

针对以上类型的问题，本文采用抽象状态自动机ASM (Abstract State Machine)作为形式化规范[9]。ASM使用了传统的数学结构来描述计算的状态，构建了清晰易懂的精确模型。相比较于非形式化方法，在定义系统规范时能得到比原系统更清晰的描述，同时ASM允许直接使用问题领域的术语和概念，并使用了最少的符号化编码及相当简单的类似于伪代码的语法，增强了对可靠性的验证能力。采用ASM作为形式化工具的主要原因是它允许在一个高抽象层次上的操作描述，这样后续对语义定义的更改只需要付出比较小的合理代价，而且高抽象层次在一定程度上也避免了过度规范的问题。因而，对于UML这种建模工具而言，采用ASM可以大大地提升未来实现的效率，也可以对其提供可靠的保证。

文献[10]中给出了状态图的各项动作的语义规则，但在具体的实现过程中，仍无法满足复杂时序的需求。本文对状态图进行形式化采用的是定义规则以及多agent实时ASM控制流程相结合的方法，既给出了规范，又对时序进行了有效的控制。

3.1 语法定义

首先介绍一下ASM的语法，主要由以下3种简单的语法规则组成。

1. 更新规则：$f(s_1,s_2,\cdots,s_n):=(t_1,t_2,\cdots,t_n)$，令函数$f$在元组$(s_1,s_2,\cdots,s_n)$处的值更新为$(t_1,t_2,\cdots,t_n)$。

2. 条件规则：if g then R1 else R2 endif，若g值为真则触发R1，否则触发R2。

3. 一致性规则：do in-parallel R1R2 enddo，若规则R1和R2一致则同时触发R1，R2，否则就什么也不做。

ASM使用规则描述状态之间的跃迁，并由agent负责控制流的切换。状态图的基本要素是状态、跃迁以及事件。对于并发状态，可以采用多agent方式推进控制流，而对于前面提出的递归交互类问题，则需要进一步细化状态的各个位点的语义，以及agent之间控制权的切换过程。

3.2 语义描述

为满足UML建模的需求，首先定义agent，每个控制流对应一个agent，状态的运行由agent控制，而每个agent存在以下两个状态：

1. 等待(wait)：等待其他agent的结果，或是来自父agent的调用，初始化agent时总是处于该状态。

2. 运行(run)：执行相应状态的动作。每当agent执行到某个位点则进入状态，若需要申请其他状态的服务，则为该子状态创建一个新子agent，产生一个新的控制流程。

当agent对应的流程结束时，相应的agent也随即消亡。

模型的正确性首先要保证各个图形的一致，即系统的完备性。根据前面提出的问题，避免状态图死锁就需要保证在顺序控制流的每个时刻有且仅有一个agent处于运行状态，从而使状态图的正确性随着agent之间控制权的切换而得以保证。

下面根据状态图的语法要素、产生语义歧义的主要原因以及ASM中的语法，为状态图的agent定义以下两个规则：

1. 跃迁规则：鉴于UML的run-to-completion语义，在递归时容易出现死锁的情形，以下定义了层次化的agent结构，由agent来控制并实现递归嵌套的过程：每当触发跃迁时，首先判断跃迁的事件是否请求了来自于其他控制流的服务，如果没有则可以正常触发；否则，为新状态创建一个agent作为当前agent的子agent。

```
Rule trigger
    if state(event)! =this
    then
        agenti:=creatagent();
        agenti. parent:=agent(this);
        agent(this). status:=wait;
        agenti. state:=eventstate();
        agenti. status:=run;
End
```

2. 退出规则：在当前 agent 结束时，将控制权交还其父 agent。

```
Rule exit
    currentagent:=agent(this). parent;
    Killagent(this);
    currentagent. status:=run;
End
```

下面验证以上规则可保证每个时刻恰有一个 agent 处于运行状态。

证明：开始时，初始状态对应的 $agent_0$ 处于运行状态，下面对调用次数 n 进行归纳证明。

假设第 k 次调用后恰有一个 agent，$agent_k$ 处于运行状态，则随后或者进行第 $k+1$ 次递归调用，或者 $agent_k$ 运行结束。第一种情况下，根据 Rule1 将生成一个新的 $agent_{k+1}$，然后 $agent_k$ 转入等待状态，结论正确；

第二种情况下，根据 Rule2，$agent_k$ 结束，并将控制权返回其父 agent，结论正确。

因此根据 Rule1 和 Rule2，可以保证系统在任意时刻有且仅有一个 agent 处于运行状态。

4 实例分析验证

状态图和顺序图的不一致的来源是两个控制流之间互相请求对方的服务，因而均处于不稳定的状态，导致任一控制流都无法继续请求而造成死锁。而在 ASM 的多 agent 规则演绎下，检验是否一致、避免死锁的问题就演变成为层次化 agent 结构能否正确结束，即每个 agent 能否最终转入运行状态直至正常退出的问题。

对于图 1 的模型，其语义逻辑关系为每当客人按下按钮，控制器必将返回自己选择的电梯。因此顺序图的前置条件和后置条件分别是：

press≠0|⇒choice

而应用在图 2 的模型上，其逻辑关系抽象为 elevator 的状态 S1 必将跃迁至状态 S2，即：

S1|⇒S2

在顺序系统中，每个时刻只有一个状态处于运行状态，且恰有一个 agent 处于运行状态。因此，根据跃迁规则 trigger 和退出规则 exit，可以避免出现原模型中控制流相互请求，继而死锁的问题。生成图 2 对应的 agent 层次树模型的具体过程如下。

在模型初始化时，进入 elevator 对象的状态 S1，相应生成层次树的树根 agentA1，并将控制权移交 A1，在 elevator 对象内部运行时，若调用 query()，控制权无需交接，而一旦按下电梯按钮，press≠0，启动了对另一对象 controller 的外部调用，即 S3. choose()，则触发跃迁规则 trigger，生成新的 agentA2，并将 A1 的状态置为 wait，进入 controller 对象的状态 S3。

S3 向 S4 的跃迁在 controller 对象的内部进行，因此不触发规则，但当在 S4 状态下进行电梯状态查询时，要请求对外部对象调用 S1. query(status)，从而生成新 agentA3，进入 elevator 对象的 S1 状态进行查询。

返回时，遵循自底向上的方式，根据 exit 规则，逐层地退出 agent。

图 2 对应的 agent 树如图 3 所示。其中状态间实线表示的是同一控制流中的跃迁（如 S1－－>S1），而虚线表示的是不同控制流之间的状态跃迁（如 S1－－－>S2）。当某一 agent 中包含虚线跃迁时，就会引发 trigger 规则，创建新的子 agent，生成新控制流，否则该 agent 将正常运行直至退出。

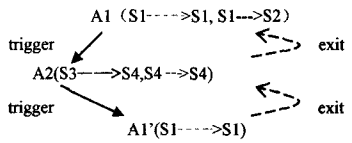

图 3　图 2 对应的 agent 树

利用这种规则和模型相配合的方式，将原有模型中易产生歧义和死锁的递归调用转为层次更为清晰的树结构。对于更复杂的模型，树的层次会更繁杂，但根据树的遍历算法，易证在这种结构中，必然可以从树根出发遍历所有树中节点（由 trigger 规则，生成新的树节点），并依次结束，逐层退出（exit 规则）返回根节点。对本实例而言，就意味着图 2 所示的状态图中 S1 状态可以完全结束并退出进入 S2 状态，达到与图 1 所示的顺序图一致的语义要求。

所以，采用 ASM 实时 agent 与规则相结合的方法，可以方便地解决与顺序图出现语义歧义、死锁的问题。同时，在实现过程中，可以借助于 ASM 支持逐层精化的特点，按照需求添加必要的规则，分层次地进行精化和验证，从而自底向上地得到正确的实现。

结束语　如上所述，对于状态图而言，run-to-completion 语义虽然保证了其状态运行的完整性，但也限制了它在目前复杂系统领域的进一步应用，特别是无法对递归调用建模，即在接收到第二条消息时，系统仍处在对第一条消息的反应过程中。虽然可以借助于 petri 网、时序逻辑、方法状态机、协议状态机等方式来解决语义上的缺陷，但过程太复杂，并且对程序员的逻辑知识有较高的要求。本文借助于 ASM 的多 agent 实时语义规则，由 agent 来控制状态的迁移，通过控制权的交接，切换多个 agent，可以避免递归系统中出现死锁，方法简洁易懂，便于实现，而且层次树的结构一方面便于验证，另一方面可以通过 agent 的层次在实现过程中对递归深度进行实时的监控，有助于得到更健壮的模型。

本文的实例是相对简单的顺序系统模型，对于复杂的并发结构而言，也可以采用类似的方法，但是在并发流程的推进过程中，可能会出现交叉调用的情况，影响可靠性的因素更多更复杂。在今后的工作中，将进一步研究并发的多控制流复杂系统的一致性。

参 考 文 献

[1] Schäfer T, Knapp A, Merz S. Model Checking UML State Machines and Collaborations[J]. Electronic Notes in Theoretical Computer Science, 2001(47): 1-13

（下转第 223 页）

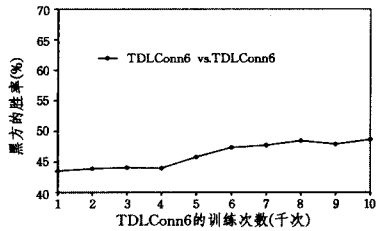

图 6　TDLConn6 自学习训练时黑方胜率

TD 训练过程中，为观察 TDLConn6 的水平变化，对于 S_T 中的每个局面，都让 TDLConn6 与 NEUConn6 对弈一局（分先后手）。这时，1)将图 3 中的一个引擎换成 NEUConn6；2)关闭 TDLConn6 中的随机策略，在整个对弈过程中，每个引擎都选择各自的最佳着法。TDLConn6 与 NEUConn6 的对弈结果如图 7 所示。在图 7 中的横坐标 i 处，TDLConn6 使用权值向量 w_i（训练 $1000\times i$ 盘棋时所保存下来的）与 NEUConn6 对弈。经 10020 盘（耗时约 157h）的自学习，TDLConn6 执黑时，在 S_T 中的胜率提高了 3%～5%；TDLConn6 执白时，胜率提高了 8%左右。程序水平变化的整体趋势是，随着训练的次数增多，水平逐渐提高。可见，应用 TD 学习有助于提高六子棋程序的博弈水平。

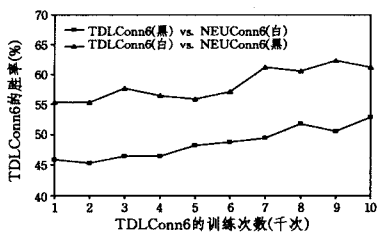

图 7　TDLConn6 与 NEUConn6 的对弈结果

结束语　本文将 TD(λ)算法应用到六子棋机器博弈中。实验结果已经表明，从零知识开始进行自学习训练的方案能够在西洋双陆棋中取得良好效果，但不适合六子棋博弈。为此，在估值函数的设计上，本文将先验知识平滑地嵌入到神经元网络中。经过 10020 盘自学习训练，程序的水平明显提高。可以断言，若像 TD-Gammon 那样进行上百万盘的自学习训练（大约将近 3 年时间），TDConn6 的博弈水平必然会更好。

参 考 文 献

[1] Sutton R S. Learning to Predict by the Method of Temporal Differences[J]. Machine Learning, 1988, 3(1): 9-44

[2] 王骄，王涛，等. 中国象棋计算机博弈系统评估函数的自适应遗传算法实现[J]. 东北大学学报：自然科学版，2005，26(10)：949-952

[3] Autonès M, Beck A, et al. Evaluation of Chess Position by Modular Neural Network Generated by Genetic Algorithm[J]. Genetic Programming, 2004, 3003: 1-10

[4] Schaeffer J, Burch N, Bjornsson Y, et al. Checkers Is Solved[J]. Science, 2007, 317(5844): 1518-1522

[5] Wu I-Chen, Huang Dei-Yen. A New Family of k-in-a-row Games [C]// Proceedings of The 11th Advances in Computer Games Conference. 2005: 88-100

[6] Xu Chang-ming, Ma Z M, Xu Xin-he. A Method to Construct Knowledge Table-base in k-in-a-row Games[C]// Proceedings of ACM Symposium on Applied Computing. 2009: 929-933

[7] Baxter J, Tridgell A, Weaver L. KnightCap: A Chess Program that Learns by Combining TD(λ) with Game-Tree Search[C]// Proceedings of the 15th International Conference on Machine Learning. Madison, 1998: 28-36

[8] Baxter J, Tridgell A, Weaver L. Experiments in Parameter Learning Using Temporal Differences[J]. ICGA Journal, 1998, 21(2): 84-99

[9] Sutton R S, Barto A G. Reinforcement Learning: An Introduction [M]. Cambridge: MIT Press, 1998

[10] Baxter J, Tridgell A, Weaver L. TDLeaf(λ): Combining Temporal Difference Learning with Game-Tree Search[C]// Proceedings of the 9th Australian Conference on Neural Networks. 1998: 168-172

[11] Tesauro G. Temporal difference learning and TD-Gammon[J]. Communications of the ACM, 1995, 38(3): 58-68

[12] Tesauro G. TD-Gammon: a Self-Teaching Backgammon Program, Achieves Master-Level Play[J]. Neural Computation, 1997, 6: 215-690

[13] Tesauro G. Practical Issues in Temporal Difference Learning [J]. Machine Learning, 1992, 8: 257-277

[14] Allis L V, van den Herik H J, Huntjens M P H. Go-Moku Solved by New Search Techniques[J]. Journal of Computational Intelligence, 1995, 12(1): 7-24

[15] Allis L V. Searching for Solutions in Games and Artificial Intelligence [D]. Maastricht, The Netherlands: University of Limburg, 1994

(上接第 174 页)

[2] 董威，王戟，齐治昌. UML Statecharts 的模型检验方法[J]. 软件学报，2003(14)：750-756

[3] Zhou Xiang, Shao Zhi-qing. ASM Semantic Modeling and Checking for Sequence Diagram[C]// ICNC'09. Vol. 5, 2009: 527-530

[4] 占学德，缪淮扣. 基于 UML 状态图测试的充分性准则[J]. 计算机科学，2005，32(5)：230-235

[5] 董涛，顾庆，陈道蕾. 基于状态图的测试技术研究[J]. 计算机科学，2007，34(10)：264-268

[6] Ober I. An Asm Semantics of UML Derived from the Metamodel and Incorporation Actions, Advances in Theory and Applications[C]// 10th International Workshop. ASM, 2003

[7] Bernardi S, Donatelli S, Merseguer J. From UML Sequence Diagrams and Statecharts to analysable Petri Net models[C]// WOSP '02. Italy: Rome, 2002: 35-45

[8] Tenzer J, Stevens P. Modelling recursive calls with UML state diagrams[J]. Lecture Notes in Computer Science, 2003, 2621: 135-149

[9] Borger E, Stark R. Abstract State Machines[M]. Springer-Verlag, 2003

[10] Compton K, Huggins J, Shen W. A Semantic Model for the State Machine in the Unified Modeling Language[C]// UML 2000 workshop. 2000: 25-31