第 3 章　CSMSL 的模型体系及其并行仿真引擎

3.1　CSMSL 仿真模型技术体系

从软件实现上，CSMSL 的仿真组件（软件存在形式）包括两种类型（见图 3-1）：原子组件模型（有的文献又称元素组件模型[98]）和复合组件模型。原子组件模型是指具有完备的输入/输出接口，功能、行为不可再分的基本模型；复合组件模型是由若干原子和复合组件模型复合构成的模型。

为了使复合组件模型具有相对独立的仿真推演（仿真时间管理等）功能，CSMSL 的复合组件模型内部被赋予了引擎机构，引擎使组件能够根据外部输入事件和当前状态调度内部原子组件或者复合组件的行为[96-98]。

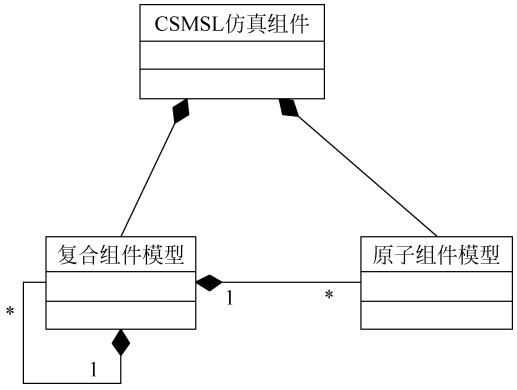


图 3-1　软件角度的仿真服务组件分类图

CSMS 仿真服务组件规范参照了 COSIM 标准规范[96-98]，从模型接口层、模型架构层、模型实现层三个层面对组件模型进行描述，如图 3-2 所示。

模型接口层规范直接面向用户，其目的是规范仿真服务组件的描述，并建立对

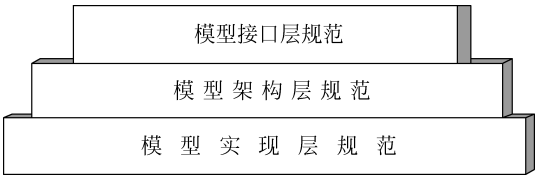


图 3-2　仿真组件模型规范层次结构图

仿真服务组件的一致调用接口。仿真服务组件接口遵循面向服务的原则，支持被用户以功能接口的方式调用，而不必关注模型的内部结构和状态。

模型架构层规范向上是针对接口层的特性设计的底层支撑服务，向下则是作为模型实现层的元模型，对采用特定技术标准实现基础架构进行约束和指导。模型架构层规范主要规定 CSMSL 仿真服务组件的静态结构和彼此间的互操作机制，包括端口结构、信息交互机制、时间管理机制和行为调度机制等。这些机制囊括了 CSMSL 仿真服务组件运行时的特征，也是仿真运行时必需的公共服务。在 CSMSL 仿真服务组件的设计中会将实现这些机制的软件模块独立出来，作为所有仿真服务组件共用的基础架构。

模型实现层包括两方面的内容：模型基础架构软件框架和仿真引擎软件框架。CSMSL 仿真服务组件重在模型的集成、调用和发展，因此其基础架构在具体实现上采用两种策略：采用 OMG/MDA，根据需要采用不同的技术实现；开发适配器，兼容不同机制或不同实现技术的模型。一般可以 MS COM 和 C++等作为实现技术，针对组件基础架构的设计和组件功能体的实现方法进行阐述。

3.1.1　模型接口层规范

CSMSL 仿真服务组件是一种软件模块，但它更是一种仿真模型，其适用的范围为连续系统、离散系统和混合系统。因此，CSMSL 仿真服务组件的接口规范需要以一种系统建模理论作为基础。本书采用了混合系统理论，并以混合输入/输出自动机作为模型框架。以此为基础，结合在仿真时间管理方面的扩展，以及基于面向服务原则的抽象约束共同构建 CSMSL 仿真服务组件的接口规范。

1. 混合系统与混合自动机模型

在自然界中，混合系统（Hybrid System，HS）指的是一类其行为由不同特性的进程或实体来定义的系统[99]。在工程中，混合系统则是现代计算机等数字控制技术渗透到连续处理系统的产物，可以理解为由连续和离散特性相互混合而成的系统[100]。

目前，混合系统存在多种采用不同途径建立的模型，但是它们的本质都是将连续时间动态特性和离散事件动态特性结合起来，如以区段演算（Duration Calculus）及其扩展为工具的方法、以 AHS 系统进行混合控制的方法等。混合自动机则是其

中一种典型的模型框架，它将描述连续动态行为的微分方程嵌入到传统的离散状态机模型中，从而使自动机模型具备描述连续行为的能力，如图 3-3 所示：每个离散状态定义相应的流条件(flow condition)，描述在此离散状态下的连续动态过程，如图 3-3 中 B_1、B_2；离散状态的切换将导致连续变量的不连续跳变，如图 3-3 中 B_1 与 B_2 间的跳转。对应于每一个离散状态，连续变量需满足一定的不变集条件(定义域)，若其连续演化使得变量超出这个不变集条件，连续演化将停滞。

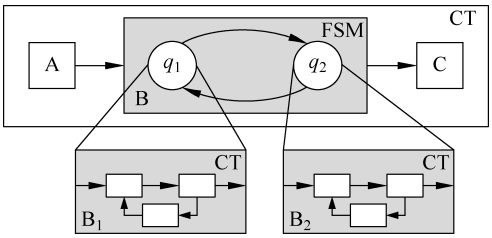


图 3-3　混合自动机模型结构示意图

基本的混合自动机模型定义是一个八元组[101]：$H=(Q,X,\text{Init},f,\text{Inv},E,G,R)$，在此基础上为其增加输入/输出结构，则成为开放的混合自动机模型，又称为混合输入/输出自动机(HIOA)，本书以 HIOA 模型作为 CSMSL 仿真组件接口规范的基础。

定义 3-1　混合输入/输出自动机 HIOA 模型[102]。

混合输入/输出自动机的形式化模型为：

$$H=(Q,X,U,Y,\text{Init},f,h,\text{Inv},E,G,R)$$

其中，

- Q 是离散状态集合；
- X 是连续变量集合，$X=\mathbf{R}^n$；
- U 是输入变量集合，其中包括离散量和连续量：$U=U_D\cup U_C$；
- Y 是输出变量集合，其中包括离散量和连续量：$Y=Y_D\cup Y_C$；
- Init 是初始状态集合，$\text{Init}\subseteq Q\times X$；
- $f:Q\times X\times U\to TX$ 是一个向量场(X 的切向空间)，用于描述连续演变；
- $h:Q\times X\times U\to Y$ 是内部状态和从输入到输出的映射；
- $\text{Inv}:Q\to P(X\times U)$ 为每个离散状态 $q\in Q$ 分配一个不变集，即连续演变的定义域；
- $E\subset Q\times Q$ 是离散转换集；
- $G:E\to P(X)$ 为每个转换 $e=(q,q')\in E$ 分配一个守卫条件，其中 $P(X)$ 是 X 的幂集，$P(X)=2^X$；
- $R:E\times X\times U\to P(X)$ 描述离散转换后瞬间系统的状态，即针对转换 $e\in E$、$x\in X$ 和 $u\in U$ 的复位关系，其中 $P(X)$ 是 X 的幂集，$P(X)=2^X$。

在 HIOA 模型中，U、Y 和 h 属于模型的外部视图，对用户是可见的；而其他元素则属于模型的内部视图，是在模型内部需要实现的内容，可以向用户屏蔽。

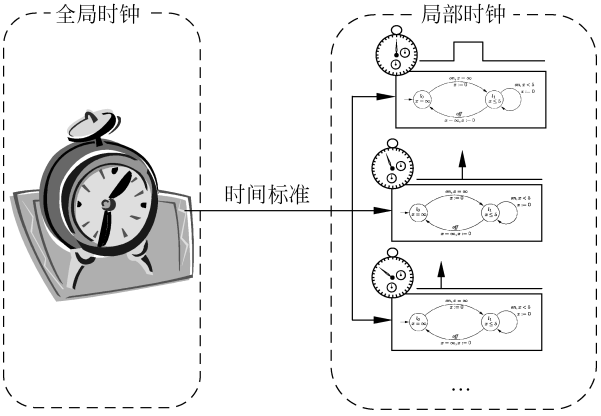
2. 时间管理与 HIOA 模型扩展

混合系统自动机模型可以看作是对时间自动机的一般化，因此在 HIOA 中也隐含了时间的概念。但是，它能否满足仿真中时间管理的需求？如果不能，如何扩展？这些都是需要分析的内容。

时间管理自动机（Timed Automata，TA）中时间管理的本质是由一个统一的全局时钟通过局部时钟控制和约束自动机行为的发生。其中，全局时钟相当于记录时间自然流逝的时钟，为全局提供一个时间基准；而局部时钟则相当于秒表，提供局部定时服务。TA 的时间管理与仿真时间管理的差别存在两个方面：局部时钟的含义；全局时钟与局部时钟的关系。

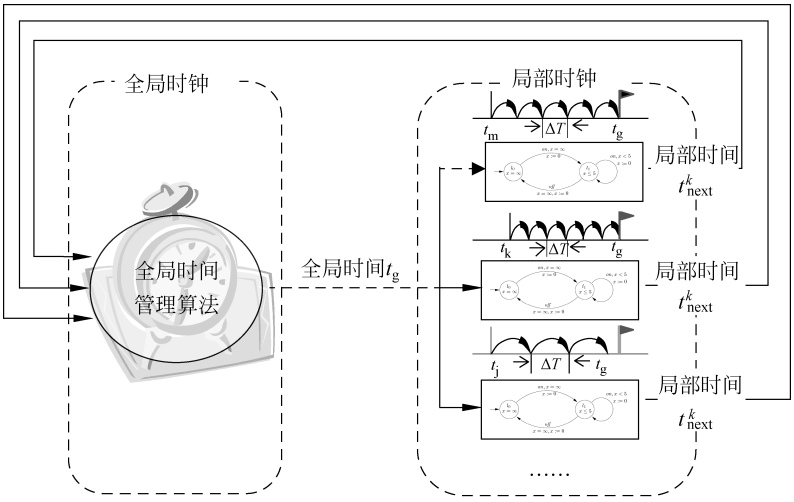
（1）局部时钟的含义：在 TA 中局部时钟就是一个定时器，它们以与全局时钟相同的速率推进，但彼此间独立，可以随时独自复位。在仿真时间管理中，局部时钟不独立存在，而与模型融合在一起，是模型解算进度的指示。而且，当模型间存在耦合关系时，它们的局部时钟间也存在限制关系，需要通过全局时间管理算法协调后续的推进时间。

（2）全局时钟与局部时钟的关系：在 TA 时间管理中，全局时钟与局部时钟的关系是单向的，全局时钟为局部时钟提供时间标准如图 3-4(a)所示；而在仿真时间管理中，二者关系是双向的，一方面全局时钟为局部时钟提供时间标准，另一方面局部时钟为全局时钟提供时间推进的依据，如图 3-4(b)所示。



(a)

图 3-4　全局时钟与局部时钟的关系



(b)

图 3-4 （续）

3.1.2 模型架构层规范

模型架构层规范向上是对接口层的特性而设计的底层支撑服务，向下则是作为模型实现层的元模型，对采用特定技术标准实现基础架构的约束和指导。模型架构层规范主要规定 CSMSL 仿真组件的静态结构和彼此间的互操作机制，包括信息交互机制、时间管理机制和行为调度机制。

1. 模型端口与信息输入/输出

为实现模型功能，CSMSL 仿真服务组件需要外界的信息；同时也需要向外界提供模型产生的信息。但是，CSMSL 仿真服务组件本着重用的原则，设计为自包含的实体，在模型内部禁止对外界的直接访问，同时也禁止外界对模型内部信息的直接访问。因此，需要针对模型的信息输入/输出建立相应的机制。

1）端口与端口项的概念

在 CSMSL 架构设计中采用了面向模型的信息传输机制，使模型间的信息传输从传统的函数参数传递跃升为专用的信息通道，真正从物理实现上建立了模型间信息交互机制，称该信息通道为“端口 Port”。

2）模型信息输入/输出方式

在模型内部的功能实现部分设置与端口项数据类型一致的变量，模型信息输入/输出本质上就是与外界进行信息交换时，功能体变量与端口之间的交换机制。

模型通过 3 种方式使用外界信息：请求式输入机制、反射式输入机制和请求式扩展输入机制。

2. 模型的结构

模型的结构是针对组件模型而言的，模型结构包括两方面的内容：模型的引用，即该组件模型由哪些子模型构成；模型之间的连接，即该组件模型、子模型、内部引擎之间的信息交换关系。模型连接是 CSMSL 仿真服务组件中的核心机制之一，它是实现模型交互的基础。

1）模型引用

模型引用描述的是一个组件模型在实现中涉及的原子组件模型和其他子组件模型的集合，由于其他子组件模型又会引用其他子模型，因此组件模型具有层次化结构的特征。在运行时，一旦一个模型被载入系统，则该模型所引用的所有子模型都需要被载入。

2）模型的连接

模型的连接描述了组件模型实现某功能时其内部子模型间的信息交互情况，在 CSMSL 建模时通过建立不同模型的端口项之间和端口项与引擎之间的连接关系来表达。而且，一个连接的源和端之间需在输入-输出类型、数据类型、离散-连续类型等方面匹配。

3）基于模型连接的信息交互

在运行过程中，按照建模时描述的连接信息完成模型间的信息交互，实现信息的互连。

3. 模型实现层规范

模型实现层规范包括两部分：模型支撑框架，用于约束模型端口机制的维护、模型功能体与端口的信息交互、模型信息输出通知与请求机制等的实现；引擎框架，用于约束时间管理、行为调度、模型间的连接等。模型支撑架构是组件模型和原子组件模型共有的部分，引擎架构则仅用于组件模型。

CSMSL 仿真引擎的功能与模型架构层规范中的“模型的结构”“模型的时间管理”和“模型的行为调度”对应，包括 3 个部分：时间管理、行为调度和结构管理，它们分别由时间管理器、行为调度器和结构管理器构成。时间管理器内部设置针对“分布时间-系统时间/节点时间”两个层次的时间管理算法，并对行为调度器和结构管理器传递来的事件按照时间或者接收顺序进行管理；行为调度器内部设置形式化计算模型，根据时间管理器输出的事件决定需要调度的模型；结构管理器包括对所属子模型的引用和模型间的连接两部分内容，分别实现对模型行为的激发和模型间信息的交互。时间管理器、行为调度器和结构管理器都是计算模型框架，而不是针对具体系统的实现，输入不同的逻辑信息后可以表现出相应的行为特性。其中，时间管理器和行为调度器从算法到实现都较为复杂。

3.2 CSMSL 并行仿真引擎负载均衡组件调度方法

并行仿真引擎按执行架构来讲主要有两种：主从架构(master-slave)和对等(peer-to-peer)架构。主从架构的多线程仿真引擎的特点是使用几个主要的线程来分发事件处理任务给多个从线程，从线程仅处理被分配的事件，然后将结果返回并等待新任务，这种架构的优势是可以在一定程度上平衡多核之间的任务负载，如 Miller 的 Threaded Warped、陈丽丽的时间弯曲系统以及 Vitali 的乐观仿真的负载共享架构等基本上都属于此类。但是 Threaded Warped 中仅有一个主线程在分发任务，如果从线程数目众多且处理事件比较快时，主线程会成为性能瓶颈，导致可扩展性差；陈丽丽的工作对此做了一定程度的改进，所有的线程都重复地去由分布的多个事件队列构成的全局队列中寻找最近的事件来处理，在这种架构下，线程之间的同步代价较大，很多线程都去寻找最近的事件时会产生较为剧烈的竞争；Vitali 则从操作系统层面上入手，将多核资源(处理核的时间片)按照逻辑进程模型(Logical Process，LP)所需的计算量成比例地分配给各个 LP，但是这种频繁的线程切换方式会带来显著的计算资源开销。因此，许多研究者选择使用对等架构来构建多线程的并行仿真引擎，例如国防科技大学唐文杰提出的 HPSK、国防科技大学彭勇提出的 HLA 成员并行框架以及 Angelo 提出的基于谷歌 Go 编程语言的时间弯曲引擎等，这类架构的引擎的每个线程都是主动执行的，每个线程都在时间管理算法的约束下不断地发送调度事件给其维护的 LP 模型，故对等架构下可能会产生负载不均衡问题。但是到目前为止，尚未见到有面向多核环境的对等架构并行仿真引擎的负载均衡研究出现。

并行仿真引擎中的对等架构体现在两个层面，分别是仿真引擎节点内及节点间的对等架构。所谓节点内对等架构，指在一个引擎节点内，不同的引擎计算线程间不存在主从关系，各引擎线程间完全对等。所谓节点间对等架构，指在不同引擎节点间不存在主从关系，各引擎节点间完全对等。基于这种对等架构，需要同时实现节点内多核之间的负载均衡及节点间的负载均衡。

具体方法如下：在节点内，仿真引擎包含多个工作线程，各个工作线程并行地在本节点对象池中查找含有可安全处理事件的组件实例，并对可安全处理的事件进行处理。由于该架构下各个工作线程均是全负载进行计算的，不会存在空闲的工作线程，因此从某种程度上实现了负载均衡。但由于安全事件时间戳的计算会消耗大量通信资源，因此应尽量减少无效的可安全处理事件的查询次数，这里提出空载率的概念。空载率指一个组件实例在单位查询比率下出现无效查询的概率。基于空载率均衡的思想，对空载率高的组件实例调低被选取的概率，使空载率降低，降低通信负载；而对空载率低的组件实例调高被选取的概率，使空载率升高，将计算资源更多地向其倾斜，使其事件尽快处理，加快仿真计算速度。

如图 3-5 所示，在节点间，不设中心控制节点，而是各个节点根据其获得的全局负载信息独立决策进行负载均衡，按照对等、自主原则，节点仅能够在计算负载比较高的时候控制对外迁出 LP。该方法的优势在于其扩展性好，比较适用于仿真系统的模型分布在大量节点上并行执行的情况。每个节点按照一定规则（周期性或者通过指标判断）启动自主性的负载均衡。节点在确定要负载均衡后，首先获取本地存储的全局各节点的计算负载并对其进行排序，如果本节点属于负载过重的前几位，则计算本节点上 LP 模型与本节点、轻载节点的通信差值，将该差值作为优先级指标，确定 LP 模型的迁移方案。高优先级表明该 LP 模型与其他节点的通信比与本节点的通信频繁，所以选择高优先级的 LP 模型迁移到低负载的节点比较合理，但是迁移后不能导致两个节点之间的计算负载差值超过门限值。

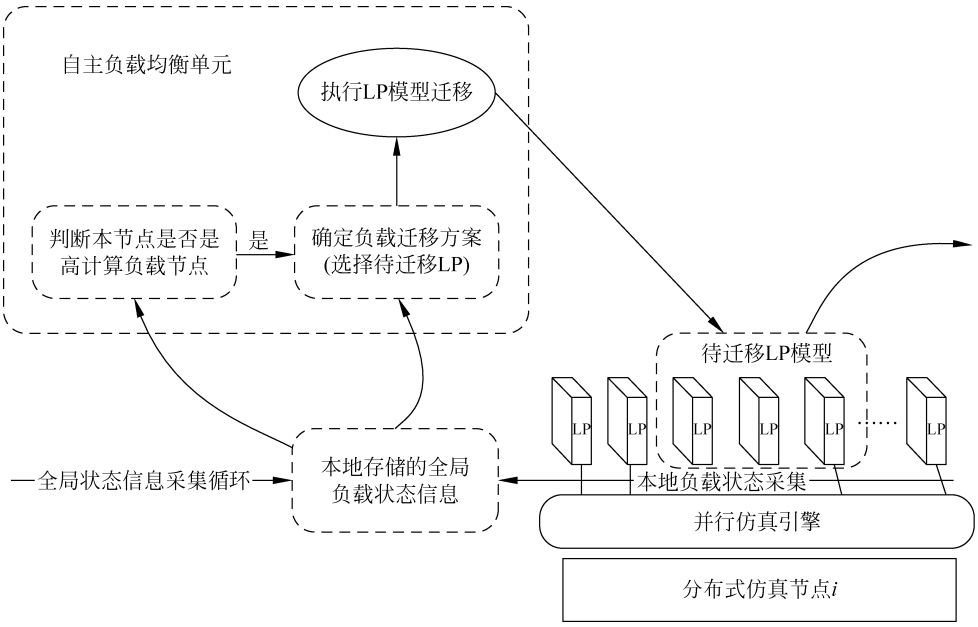


图 3-5　自主负载均衡的机制

实现负载均衡主要分以下 5 步。

S1，设立初始查询概率集。

各个引擎线程按一定的查询概率对当前引擎节点内组件池内的组件进行可安全处理事件查询，如图 3-6 所示。

设组件的查询概率集为 Q。

$$Q=\{Q_j \mid j \in [1,2,\cdots,N]\} \tag{3-1}$$

式(3-1)中$[1,2,\cdots,N]$为引擎节点编号集，Q_j 为 j 引擎节点上的查询概率集，

$$Q_j=\{q_{ij} \mid i \in [1,2,\cdots,N_j]\} \tag{3-2}$$

式(3-2)中$[1,2,\cdots,N_j]$为 j 引擎节点上的组件编号集，q_{ij} 为 j 引擎节点上 i

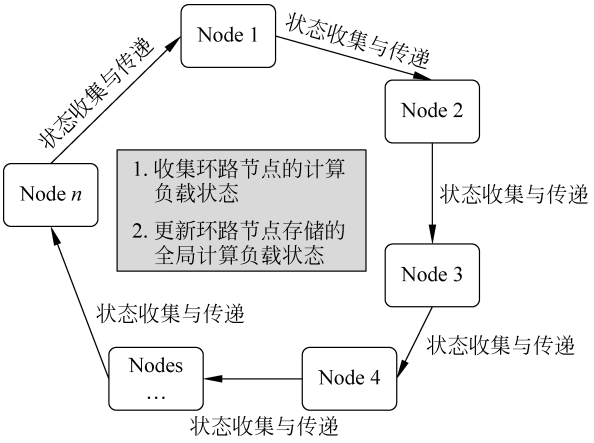


图 3-6　全局负载状态获取

组件的查询概率，即每个引擎线程按此概率选中组件，进行安全处理事件查询。

对于任何 j 满足 $\sum_i q_{ij}=1$。

在初始情况下所有 q_{ij} 为相同的值，即使得 $q_{ij}=\dfrac{1}{N_j}$。

S2，建立通信负载统计模型。

对每个组件与各个引擎节点间的通信负载进行统计，得到每个组件的通信负载集

$$\mathrm{TX}_{ij}=\{\mathrm{tx}_{ijk}\mid k\in[1,2,\cdots,N]\} \tag{3-3}$$

式(3-3)中 tx_{ijk} 表示单位墙钟时间内 j 引擎节点上 i 组件与 k 引擎节点间的通信次数，当 $j=k$ 表示当前组件与节点内的通信量。

S3，建立空载率统计模型。

在每个组件被选中并经过安全处理事件查询后，会出现有可安全处理事件或无可安全处理事件这两种情况，经过多次查询操作后，将得到每个组件被查询时处于无可安全处理事件状态的率，即每个组件的一个空载概率，将其定义为 kz_{ij}。

对于特定仿真系统，一个组件 i 的事件密度是固定的，每次查询后，如果有可安全处理事件，引擎会将其处理后清除，所以当查询概率 q_{ij} 增加时，其空载率 kz_{ij} 也会同样增加。

在给定的查询概率集为 Q 下，经过一定时间的统计后，将得到每个仿真组件的空载率集

$$\mathrm{KZ}=\{\mathrm{kz}_{ij}\mid i\in[1,2,\cdots,N_j],j\in[1,2,\cdots,N]\} \tag{3-4}$$

另外，定义每个引擎节点上的平均空载率为

$$\overline{\mathrm{kz}_j}=\frac{\sum_i \mathrm{kz}_{ij}}{N_j} \tag{3-5}$$

式(3-5)中 $i \in [1,2,\cdots,N_j]$。

定义整个仿真系统的平均空载率

$$\overline{\mathrm{KZ}} = \frac{\sum_j \overline{\mathrm{kz}_j}}{N} \tag{3-6}$$

式(3-6)中 $j \in [1,2,\cdots,N]$。

S4，节点内的查询概率调整。

首先判断节点内的空载率是否均衡，即是否满足对节点内的所有组件 $\mathrm{kz}_{ij} \approx \overline{\mathrm{kz}_j}$，若满足则执行 S5，若不满足则执行如下操作：

在节点内调整组件的查询概率，使得空载率趋于均衡，步骤如下：

S41，取 $k_{ij} = \dfrac{q_{ij}}{\mathrm{kz}_{ij}}$，可以得到 $K_j = \sum_i k_{ij}$，$i \in [1,2,\cdots,N_j]$；

S42，将每个组件新的查询概率调整为 $q'_{ij} = \dfrac{k_{ij}}{K_j}$；

S43，重复执行 S3 与 S4，直至使得所有组件的空载率接近于当前节点的平均空载率，即 $\mathrm{kz}_{ij} \approx \overline{\mathrm{kz}_j}$，然后执行 S5。

S5，节点间的组件迁移：将 $1-\overline{\mathrm{kz}_j}$ 理解为每个引擎节点 j 的当前计算负载。如图 3-6 所示，将所有参与引擎节点按照节点序号首尾相连组成一个环路。沿着该环路节点循环，即可收集各个节点的计算负载信息，并沿路更新到各个节点，使得各个节点获得较新的全局计算负载信息，计算可以得到整个仿真系统的平均计算负载 $1-\overline{\mathrm{KZ}}$。每个节点判断自身是否为最高计算负载节点且 $\overline{\mathrm{KZ}}-\overline{\mathrm{kz}_j}>0.1$，如是，则启动自主性的组件迁移过程。

记当前节点为 j，为最高计算负载节点。选择计算负载最低的节点为组件迁移的目的节点，记为 k。

在选择迁移组件时，同时实现通信负载均衡模型。选择当前节点中所有组件中通信差值 $\mathrm{tx}_{ijk}-\mathrm{tx}_{ijj}$ 最大的 $(\overline{\mathrm{KZ}}-\overline{\mathrm{kz}_j})N_j$（最小值取 1）个组件作为迁移组件，将其向 k 节点迁移。迁移后将所有迁移组件的通信负载清零，并重新开始新的通信负载统计。

重复执行 S3、S4 和 S5 实现动态的负载均衡。

3.3　CSMSL 并行仿真引擎中的组件模型时间管理

3.3.1　基本原理

模型之间的并行度是影响仿真模型执行效率的关键因素。在并行离散事件仿真应用中，时间管理模型能够给出允许模型自主地解算与推进时间的区间，能否充分挖掘模型的并行度取决于时间管理模型及其实现的优劣。

并行离散事件仿真的时间管理模型通常是基于前瞻量的最小时间戳下限计算方法(保守时间管理算法)以及计算全局虚拟时间的乐观时间管理算法[95]。保守时间管理算法给出的时间区间可以确保仿真模型在此区间内不再收到来自其他模型的事件,即能够保证仿真模型只处理安全的事件,不会收到落伍的事件,在该时间区间内模型可以自主并行地执行。而乐观时间管理算法则不考虑模型之间的耦合度,允许仿真模型处理不确定是否安全的事件,一旦收到滞后消息,将回滚模型至滞后消息前状态,并恢复已经处理的违反时间顺序的事件至未来事件队列中,然后再继续处理未来事件队列中的事件(包含滞后消息)。如果模型之间的耦合程度较低,那么乐观时间管理算法会有较高的效率,但是过于冒进的事件处理会导致大量模型回滚(甚至会是雪崩式回滚),最终造成效率的低下。

CSMSL 主要关注保守时间管理算法,因为在实际仿真应用中,像乐观时间管理要求的那样,保存模型的全状态以支持模型状态的回滚比较难、内存空间代价大。而且当仿真系统的黑盒模型(如模型是基于商用软件工具构建的,或者遗留模型)中存在不可逆转操作时,保存和回滚模型状态几乎不可能。

上文所述传统的保守时间管理算法都是基于前瞻量(lookahead)和最小时间戳下限值(Least Bound Time Stamp, LBTS)的。经典的保守时间管理算法,如 HLA 系统中计算一组 HLA 成员构成的联邦的 LBTS 的公式定义如下:

$$\mathrm{LBTS}=\min\{\mathrm{LBTS}(i)\},i=1,2,\cdots,n$$

$$\mathrm{LBTS}(i)=\begin{cases}\min\{T(j)+\mathrm{Lookahead}(j)\}, & \text{如果 } i \text{ 为时间受限成员,}\\ & j \text{ 为所有给 } i \text{ 发送事件的成员}\\ \infty, & \text{如果 } i \text{ 不为时间受限成员}\end{cases}$$

所有成员在其当前时间和联邦 LBTS 之间都是可以并行执行的,但是该时间管理算法是全局规约的,扩展性差,当联邦中存在数目较多的成员时,算法性能会大大降低,而 HLA/RTI 中时间推进请求与允许机制也导致整个仿真系统性能的降低。我们倾向于按照时间管理算法确定的模型并行执行区间,主动调度模型执行,消除请求与允许的协商机制的时间成本。实际上,对每个成员来讲,LBTS(i)已经足够作为其安全处理事件的区间,该区间肯定是不小于联邦 LBTS 的,但是仅定义到成员区间的时间算法粒度还太粗,成员的 Lookahead 定义为所有内部子模型的 Lookahead 的最小值,这导致内部子模型之间的并行性被忽略。另外,由于 HLA/RTI 的订购/发布(Subscribe/Publish)机制,数据在接收端成员内往往要经过一次过滤,从订购的数据类型中挑出真正需要的数据,因此发布数据的成员数量要大于有效数据来源成员的数量,采用传统的时间管理算法中一锅端的方式不利于充分挖掘模型间的并行性。

在多核环境中,研究人员大多基于多核环境中仿真引擎的执行架构,对时间管理算法进行优化。例如唐文杰提出了多中心(多个线程调度中心)的异步 LBTS 算法[94],在一个线程处理完所有安全事件之后,获取全局 LBTS,如果当前已经有新

的全局 LBTS，则更新本地的 LBTS，然后继续处理安全事件；如果没有更新的全局 LBTS，则请求获取全局 LBTS 计算权，然后发起 LBTS 的计算，并将计算之后的结果更新到全局 LBTS 中。这种异步的计算方法可以减少重复计算 LBTS 的量。

彭勇在 HLA 并行成员框架中提出了一种改进的集中式栅障同步算法[93]。典型的栅障算法要求在所有的逻辑进程完成时间同步请求之前，先请求时间同步的逻辑进程处于阻塞状态（不能处理事件）。他的方法是逻辑进程请求时间同步后，并不进入阻塞状态，而是继续处理事件，但是处理这些事件而产生的事件被保存在逻辑进程内部，而不马上发给其他逻辑进程，逻辑进程可能会回滚，但是被限制在该逻辑进程内部。一旦逻辑进程收到了同步完成消息后，将确定哪些事件可以安全发出，哪些必须回滚。本质上这是一种保守与乐观相混合的时间管理算法。

上述这些时间管理算法大都使用全局规约的方法，算法考虑的模型粒度较粗，往往并不考虑仿真系统的结构，特别是当在多核环境中，仿真模型之间的通信速度非常快、计算核比较多时，充分挖掘模型之间细粒度的并行性是急需解决的问题，当然时间管理算法也可以按需并行执行，以提高效率。例如，在许多 HLA/RTI 的软件实现中，成员模型获得的全局 LBTS 是各个成员 LBTS 的最小值，算法上也不考虑成员之间的交互关系，可直接通过计算所有成员的当前时间加上各自前瞻量的最小值获得。

当仿真系统的结构发生变化时，当前时间管理算法也不能捕捉这种由结构变化带来的并行性。例如，当舰载飞机离开舰船自主执行任务时，它与舰船之间的耦合依赖关系将被解除，这样舰载飞机模型的执行就可以与舰船模型的执行完全并行，但当前的时间管理算法没有考虑这种结构变化。

这里提出了可以挖掘模型之间内在并行性与捕捉结构变化产生动态并行性的高效时间管理模型，以保证模型的正确同步并提高模型执行的并行性。

3.3.2　时间管理模型

CSMSL 设计的时间管理模型中，每一仿真组件采用独立的 LBTS，设组件 i 的最小时间戳下限为 LBTS(i)，那么在组件 i 当前仿真时间与 LBTS(i)之间的事件即为可以安全处理的事件。

LBTS(i)的计算公式如：

$$\mathrm{LBTS}(i)=\min\{T(j)+\mathrm{LA}(j)\} \tag{3-7}$$

其中，j 是指向组件 i 有输出关系的所有组件，LA(j)是组件 j 的前瞻量，$T(j)$的取值满足如下条件：

$$T(j)=\begin{cases}T_{\mathrm{C}}(j), & \text{如果正在处理安全事件}\\ \mathrm{LBTS}_{\mathrm{last}}(j), & \text{如果无安全事件}\end{cases} \tag{3-8}$$

即当组件 j 在处理某一安全事件时，$T(j)$为当前事件时间 $T_{\mathrm{C}}(j)$；而当安全事件

处理完毕时，$T(j)$为最近一组安全事件所对应的 LBTS(j)。

采用该时间管理模型，各个组件的 LBTS(i)可以并行计算，所有组件的计算过程只与其有输出连接关系的组件有关，不存在时间管理总线，随着仿真组件数量的增长，时间管理计算不会出现瓶颈。另外，在对输出组件筛选时采用端口及连接关系的双层过滤方法，较之于传统 HLA 仿真中只采用端口过滤的方法，大大降低了需要通信的组件数量，并且为系统的变结构特性提供了基础。

下面对该时间管理模型的正确性进行证明。

为方便证明，定义如下变量：

T_a——发送给组件 i 的事件 a 的时间戳；

$T_b(i,j)$——组件 i 发送到组件 j 的事件 b 的时间戳；

$T_c(i)$，$T_c(j)$——组件 i 与组件 j 的当前时间；

LA(i)，LA(j)——组件 i 与组件 j 的前瞻量。

在处理完事件 a 之后，组件 i 调度事件 b 给组件 j。组件 i 连接到组件 j。因此有：

$$T_c(i) = T_a \tag{3-9}$$

$$T_b(i,j) \geqslant T_c(i) + \mathrm{LA}(i) \tag{3-10}$$

$$\begin{aligned} T_c(j) &\leqslant \mathrm{LBTS}(j) \\ &= \min\{T(m) + \mathrm{LA}(m)\} \\ &\leqslant T_c(i) + \mathrm{LA}(i) \end{aligned} \tag{3-11}$$

组件 i 在处理完事件 a 后推进当前时间到 $T_c(i)$，即式(3-9)。根据前瞻量的定义，组件 i 只能够调度时间戳不小于 $T_c(i)+\mathrm{LA}(i)$的事件，即式(3-10)。

因为系统中无瞬时事件(已经发送，但是尚未收到的事件)，且事件的调度受到式(3-9)与式(3-10)的约束，因此推断得出式(3-11)成立，所以有 $T_b(i,j)\geqslant T_c(j)$，即组件 j 不会收到滞后的消息。那么组件 j 会按照时间顺序处理事件，即组件 j 的事件处理遵循本地因果约束条件。事件 a 是发送给组件 i 的；同理，可以证明组件 i 也遵循本地因果约束条件。因此，所有的组件都遵循本地因果约束条件，表明所有的组件被正确地同步。

除了对该模型的正确性进行证明外，还需对其无死锁性进行证明。

保守时间算法会导致死锁，表现为一些模型始终无法推进其本地时间。

定理 3-1　如果仿真系统产生了死锁，必定有环存在。

证明思路：假设在死锁的仿真系统中没有环路存在。那么系统可以被抽象为一个有向无环图(Directed Acyclic Graph，DAG)，图中节点代表组件模型实例，即仿真组件，有向连接代表模型之间交互关系。仿真系统的 DAG 图可以被拓扑排序为线性序列，使得如果序列被排成一排，那么所有的连接都是从左边的顶点到右边的顶点。将位于第 n 个位置的节点记为组件 n。显然，如果上游的节点(模型)不是处于挂起等待状态，那么使用本时间管理模型下游的节点也不会产生死锁，因为

上游节点逐渐推进它们的仿真时间，这样下游节点的 LBTS 会不断变大，所以下游节点也可以逐渐推进它们的仿真时间。接下来严格地证明。

引理 3-2　如果仿真没有结束，任意一个仿真组件可以在某一个时间点推进仿真时间到足够远的时刻。

将上述引理转换为数学描述为：

引理 3-2′　给定任意 $M>0$，可以找到一个 $T_M>0$，使得在经过墙钟时间(Wall clock)T_M 后，仿真组件 n 的虚拟时间 $T(n)>M$。

证明：强数学归纳法是该引理证明的基础。

基础：当 $n=1$ 时，组件 1 没有事件发送者，因此 LBTS(1)＝＋∞。如果有尚未处理的事件，处理这些事件是安全、无锁的，而且如果仿真没有结束，组件 1 可以在有限的时间内推进仿真时间到足够远的时刻。如果没有事件存在，那么 $T(1)=$ LBTS(1)＝ ＋∞。因此引理 3-2′在 $n=1$ 时成立。

归纳：假设引理 3-2′在 $n\leqslant m$ 时都是成立的。

当 $n=m+1$ 时，任意给 $M(m+1)>0$，

$$\mathrm{LBTS}(m+1)=\min\{T(h)+\mathrm{LA}(h)\}$$

其中组件 h 连接到组件 $m+1$。

按照归纳法的假设，对任意 $\delta>0$，则 $M(m+1)+\delta>0$，任意的组件 k $(0<k\leqslant m)$存在一个 $T_M(k)$，使得在墙钟时间过了 $T_M(k)$后，$T(k)>M(m+1)+\delta$。

记 $T'_M=\max\{T_M(k)\mid 0<k\leqslant m\}$，所以在墙钟时间过了 T'_M之后，$T(a)>M(m+1)+\delta$，对于任意的组件 a $(0<a\leqslant m)$。假定组件 p 在组件 a $(0<a\leqslant m)$中具有最小的 $T(a)+\mathrm{LA}(a)$。那么：

$$\begin{aligned}\mathrm{LBTS}(m+1)&=\min\{T(a)+\mathrm{LA}(a)\}\\&=T(p)+\mathrm{LA}(p)\\&>M(m+1)+\delta\end{aligned}\tag{3-12}$$

因此按照式(3-8)与式(3-12)，组件 $m+1$ 可以处理任何时间戳小于 $M(m+1)+\delta$ 的事件。

$$\begin{aligned}T(m+1)&=\mathrm{LBTS}(m+1)\\&>M(m+1)+\delta\\&>M(m+1)\end{aligned}\tag{3-13}$$

因此对于 $n=m+1$，引理 3-2′成立。

通过使用强数学归纳法，可知引理 3-2 成立。因此对任何组件，如果有任何的未来事件存在，那么这些事件都将在有限的时间内被处理。这与死锁的假设是相冲突的。因此，死锁的仿真系统中必然存在环路。

引理 3-3　未在环路中的上游节点可以在有限的时间内推进仿真时间到足够远的时刻。

证明：这些节点与上游的其他节点一起构成一个 DAG 图，从引理 3-2 可以推

出该引理是成立的。

定理 3-4　使用本时间管理模型，环路中不存在死锁现象。

证明：假设环路中的模型有待处理的事件，而且最早时间戳的事件存在于组件 i 中。因为 LBTS 的限制，没有事件可以被安全地处理。所以根据式(3-8)，$T(k)=\mathrm{LBTS}(k)=\min\{T(a)+\mathrm{LA}(a)\}$，对环路中的任意组件 k 成立。

当死锁时，

$$\begin{aligned}\mathrm{LBTS}&=\min\{T(j)+\mathrm{LA}(j)\}_{j\to i}\\&=T(k)+\mathrm{LA}(k)\\&=\min\{T(I)+\mathrm{LA}(I)\}_{I\to k}+\mathrm{LA}(k)\\&=\cdots\\&=\mathrm{LBTS}(i)_{\mathrm{latest}}+\mathrm{LA}(m)+\cdots+\mathrm{LA}(k)\end{aligned}\tag{3-14}$$

因为上游节点在一定时间后都会拥有足够大的当前时间，所以死锁时，式(3-8)中的组件 j 都是属于环路的；又由于环上仅有有限个节点，限制组件 i 的节点最终被追溯到它自己。如果环路中任意模型组件具有非零的前瞻量，那么 LBTS(i)会不断增加，组件 i 中待处理的事件一定会在一定墙钟时间后被处理。因此环路中最早事件的时间戳逐渐增加。这与仿真系统死锁假设是冲突的。因此环路中不存在死锁可能性。另外，可以看出本时间管理模型与空消息法具有相同的约束：环路中必须有非零前瞻量的模型。

3.3.3　组件模型时间管理的软件实现方法

实现本时间管理模型的本质问题是求得各仿真组件的最小时间戳下限值 LBTS(i)，而 LBTS(i)指的是所有向组件 i 有输出的组件的最小事件时间戳极小值。通常的实现方法是，每次计算 LBTS(i)时需要对所有相关组件进行遍历的通信，由于分布式仿真的缘故，不同仿真组件往往分布在不同的引擎节点上，此时的通信需要采用网络通信的方式，过多的通信会造成效率低下，这也是传统离散事件仿真中时间管理成为仿真计算瓶颈的主要原因。这里，在模型层面上采用端口及连接关系的双层过滤方法，较之于传统 HLA 仿真中只采用端口过滤的方法，大大降低了需要通信的组件数量。另外，在对算法的实现上采用其他两个方法，进一步降低了通信次数。

方法一：

设组件 k 为任意一个向组件 i 有输出的组件，由于各个仿真组件的仿真时间 $T(k)$ 是一直递增的，则其最小事件时间戳 $T(k)+\mathrm{LA}(k)$ 也是递增的，即 $T_{\mathrm{old}}(k)+\mathrm{LA}_{\mathrm{old}}(k)\leqslant T(k)+\mathrm{LA}(k)$。那么如果 $T_{\mathrm{old}}(k)+\mathrm{LA}_{\mathrm{old}}(k)$ 大于当前最小的最小事件时间戳，那么 $T(k)+\mathrm{LA}(k)$ 也不可能成为 LBTS(i)，此时就不需要对 k 组件进行通信。在遍历组件时，从最有可能成为 LBTS(i)的组件开始遍历组件，

那么采用该最小事件时间戳的预过滤方法可以过滤到大部分通信。

方法二：

传统的最小事件时间戳通信采用拉的模式，即需要计算 LBTS(i)的组件向其他相关组件请求最小事件时间戳值，相关组件在收到请求后返回其最小事件时间戳值，若涉及网络通信需则采用阻塞通信的方式。这会产生三个问题：第一，阻塞通信的效率较低，当相关组件较多时，LBTS(i)计算非常耗时；第二，当相关组件本身推进较慢时，多次的请求可能会返回相同的最小事件时间戳值，造成通信负载的浪费；第三，当某相关组件与某一引擎节点上的多个组件都有输出关系时，多个组件的 LBTS(i)的计算可能会多次请求该相关组件的最小事件时间戳值，同样造成通信负载浪费。为解决这一问题，我们开创性地将最小事件时间戳通信的拉模式改为推模式。在每个引擎节点上存储各相关仿真组件目前最新的最小事件时间戳值，当任意组件的最小事件时间戳发生更新时，需要进行一次非阻塞通信，通知各相关组件所在引擎节点更新其存储的最小事件时间戳，而当某一组件需要获取其他组件的最小事件时间戳时，只需在其所属节点的本地内存中获取即可，无须进行网络通信。这种推模式完美解决了上述的 3 个问题。

计算当前组件的 LBTS(i)的步骤如下。

(1) 通过调用连接管理接口得到所有与当前组件有输出关系的输出组件集。

(2) 首先得到上次 LBTS(i)计算时最小 T+LA 所在的组件 k 对应的 $T(k)$+LA(k)作为临时 $\mathrm{LBTS}_{\mathrm{temp}}(i)$。如果组件 k 属于当前节点，则通过内存共享直接从组件 k 读取，若组件 k 属于其他节点，则从引擎节点的组件最小事件时间戳映射表中读取。

(3) 遍历输出组件集，假设当前组件为 j，如果 j 不等于 k 且旧的最小事件时间戳 $T_{\mathrm{old}}(\mathrm{j})+\mathrm{LA}_{\mathrm{old}}(\mathrm{j})$小于 $\mathrm{LBTS}_{\mathrm{temp}}(i)$，则获取组件 j 对应的最小事件时戳 $T(j)$+LA(j)，获取方法同(2)，若 $T(j)+\mathrm{LA}(j)$小于 $\mathrm{LBTS}_{\mathrm{temp}}(i)$，则将其作为新的 $\mathrm{LBTS}_{\mathrm{temp}}(i)$。

(4) 遍历输出组件集后将最终的 $\mathrm{LBTS}_{\mathrm{temp}}(i)$作为 LBTS($i$)。

可见，采用该实现方法后，整个 LBTS(i)的计算过程不需要进行网络通信。

3.4　CSMSL 并行仿真引擎编程框架示例

CSMSL 需要相应的运行支撑(软件)，为了加快仿真解算速度，避免长时间等待仿真结果，我们实现了一个基于并行仿真引擎的框架。其体系架构和执行生命周期过程模型都按并行计算的标准来设计，因此它可以利用多核计算系统并支持仿真系统结构的动态变化；可方便地集成各类异构的黑盒、白盒模型，构建可高效并行、具有良好开放性的变结构仿真系统。

支撑 CSMSL 模型运行的仿真引擎分为两个模块：引擎核心与用户界面，这样

的设计使仿真计算与用户操作分离。引擎核心运行在后台计算节点上，采用标准C++编程，支持多核服务器和高性能计算机，通过 Socket 通信的方式对外提供操作接口。用户界面采用 Java 编程，运行在客户端计算机，通过可视化的方式提供仿真语言编译、组件加载、仿真系统变结构、仿真引擎操作、仿真结果展示等等功能。上述这种设计有利于软件维护、用户使用及软件跨平台。

为了更直观地向读者展示 CSMSL 编程框架，下面给出一个防空火力单元仿真的示意性案例。

首先，用户需要进行模型架构设计及建模。在建模过程中，复合组件模型对原子组件模型或其他复合组件模型进行封装，形成了树状的模型结构，如图 3-7 所示。

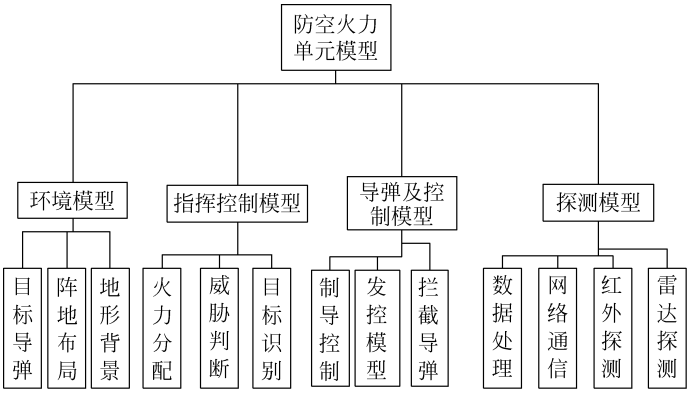


图 3-7　防空火力单元的树状模型结构

在复合组件模型中，用户需要对模型间的连接关系进行说明，图 3-8 中为防空火力单元的顶层复合组件模型，该顶层复合组件模型中的各模型均为复合组件模型，这里不再对这些复合组件模型内部的连接关系进行说明。

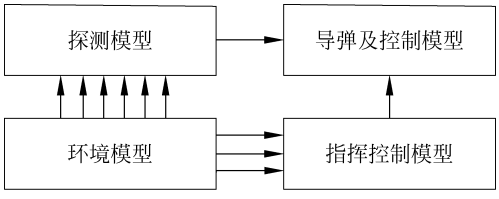


图 3-8　顶层复合组件模型间连接关系

完成仿真系统建模后，即生成针对该仿真系统的仿真语言文本，对其进行编译（见第 4 章）可生成元素组件、初始参数和扁平化连接关系文件（复合组件模型扁平化编译见 4.2.4 节）。图 3-9 所示为该系统扁平化的连接关系图。

最后，将元素组件、初始参数和扁平化连接关系载入仿真引擎并开始仿真计算，此时引擎需要根据负载均衡原则对扁平化后的组件实例进行系统划分和部署。

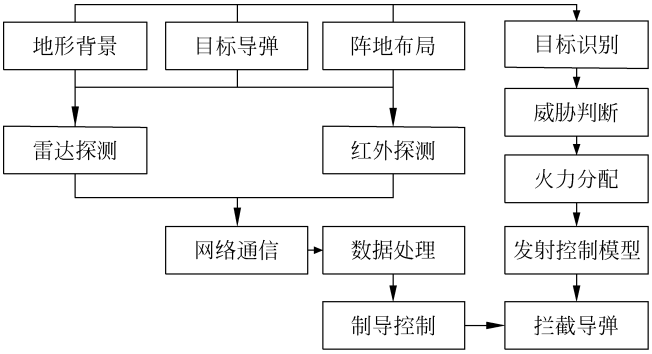


图 3-9　扁平化连接关系图

如图 3-10 所示，首先进行一次划分，需要对节点上的仿真计算量和节点间的通信量进行负载平衡，将不同组件划分到不同的计算节点上（见图 3-10 中的实线框）。其次，需要在每个节点内进行二次划分，将节点内的各个组件实例划分到不同引擎线程上（见图 3-10 中的虚线框），由于节点内的组件通信采用共享内存的方式，通信负载可以忽略，因此，二次划分不需要考虑通信负载，而只需对仿真计算量进行负载平衡。

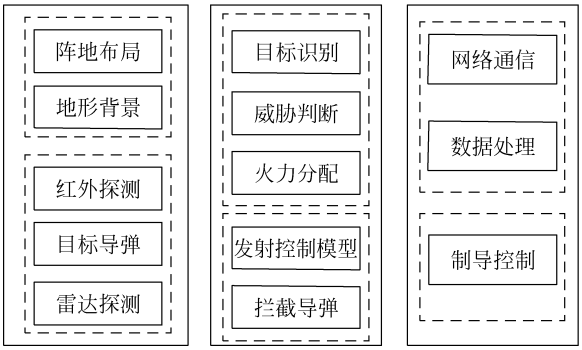


图 3-10　仿真组件的负载均衡分配（实线是计算机节点，虚线是仿真引擎线程）

这里提出的复杂系统建模仿真语言编程框架，为仿真用户设计了一套完整的仿真系统构建工具，包括仿真语言、编译器及仿真引擎。在该编程框架下，仿真用户可以专注于仿真模型的构建，而不必操心仿真计算并行化、仿真时间管理、信息订购发布及计算负载均衡等对用户仿真理论知识及计算机编程知识要求较高的内容，使得仿真研究的技术门槛大大降低，有着很好应用前景。