SPADES（并行agent离散事件仿真系统）

用户指南和参考手册

版本：0.9

作者：Prick Riley　　邮箱：pfr+@cs.cmu.edu

时间：2003年11月7日

翻译：时长娥　　邮箱：evelineshi@sohu.com

翻译：许　元　　邮箱：xychn15@yahoo.com.cn

翻译时间：2005年8月

排版上的约定：

- 所有的程序参数用斜体表示，如 *my program parameter* ；
- 所有的方法和类用等宽字体表示，如 `myMethod`, `MyClass` ；
- 所有精确的代码或事件用等宽字体再斜体表示，如 `My name is` *`yourname`*
- 特殊的SPADES术语用无衬线的字体表示，注意：在索引中这些术语都用规则字体表示

说明：

1.以上提到的几种情况均用英文原文表示，而不作翻译；
2.agent有多种翻译方式，代理，agent，实体等，这里翻译成agent较为适当，但为力不把这种理解强加于读者，所有出现agent的地方，冷然使用“agent”，不做具体翻译；
3.对于有关术语，为了防止翻译不准影响理解，在第一次出现时均注上手册中的英文表达方式；
4.因为时间关系，部分世界模型相关的内容没作分析，手册中的世界模型部分是与服务器的设计直接相关的，所以缺少这部分不会影响agent设计者对SPADES的了解和agent的设计。

Xu Yuan

Digitally signed by Xu Yuan
DN: CN = Xu Yuan, C = CN, O = Southeast University, OU = RoboCup
Date: 2005.10.04 12:59:40 +08'00'

目录

第一章　绪论

1.1　什么是 SPADES?

并行agent离散事件仿真系统(The System for Parallel Agent Discrete Event Simulation ,简称 SPADES)是基于agent（agent）的分布式仿真的中间件系统，其主要目标是人工智能，在人工智能的仿真中agent的思考是大量的计算。SPADES使得基于agent的仿真变得容易。

Agent是SPADES的主要特色，这里的agent我们只是简单指一个计算实体，它从模拟世界接收感觉，经过一定的计算然后返回将要执行的动作，SPADES明确地跟踪感知、思考、动作的固有反应时间，并在仿真中反映出来。

SPADES提供了一个抽象允许世界模型和agent的设计者忽略网络的通讯和分布式事件的分配，世界模型以仿真事件被依次执行为操作目的，agent只简单地接收感觉信息和发送动作。

SPADES的几个重要特性：

• agent以执行为基础，包括对建模（感知、思考、动作）反应时间的外在支持，对思考反映时间的建模通过跟踪一个感知被发出所经过的计算数量。

• agents可分布在若干个机器中，SPADES做了全部必要的论证，因此不管仿真使用了多少台计算机，世界模型和agent的设计者都不需要它们的不同。

• 仿真的结果不受网络延迟或者机器间负载的不同的影响，自然，这些因素会影响仿真的速度。

• agents的结构不受约束，SPADES不要求agents用某一种特定的编程语言编写，唯一的要求是agent的程序能够被通道读和写。

•跟许多人工智能的仿真环境不同SPADES中的AGENTS不需要同步，他们不在某一特定时间有一关联动作，事实上，在仿真中他们在不同时间的动作同样产生效果。

1.2　SPADES 提供了什么？

SPADES是一个仿真中间件，它不被约束于某一特定领域的仿真。为了产生一个完整的仿真，需要给SPADES加上两块：agent（计算实体），用来感知、思考和行动；世界模型，用于模拟真实世界，并负责agent之间的交互。

SPADES提供下列各项来辅助构造一个仿真：

• 提取一个基于事件的仿真，提供给世界模型一个仿真包含按时间被接收的事件，这叫作离散事件仿真。世界模型能创作仿真事件和当事件被实现时对其做任意的处理(比如，把它们的作用加在世界上)。

• 提取感觉、动作提供给agent，agent只需从一个通道读取数据（以世界模型规定的格式）然后通过另一通道发送动作。Agent除了可能需要限制计算量外，不需要做网络或外在的时间管理工作。

• 响应一个感知时，SPADES跟踪agents计算的数量，这么做是为了仿真这个事实：使用越多计算的agent响应的越慢。

• 所有的网络通讯被操纵，世界模型和agent都不需要创建套接字、知道网络地址或做其它任何的网络控制。

• 所有关于机器间分配控制所需的论证SPADES已做完。；在机器间分配一个仿真，并要正确（指不违反时间之间的因果关系）、有效率地使用可用的资源是一个棘手的问题。基于SPADES的仿真的结果不受所使用机器数目的变化、网络负载或机器负载的影响。世界模型和agents的设计者可以在很大程度上忽略分布的存在。

•提供多样的日志工具。错误记录、动作执行的轨迹、仿真状态的记录都能够由SPADES完成。

1.3　如何使用这个手册？

这个手册是 SPADES 的主要文件，它的目标是各种不同的使用者。这里是对于不同的使用者使用这个手册的推荐方法：

1）完整的仿真的使用者 如果你要使用一个完整的基于 SPADES 的仿真，也许你还需要 SPADES 的一些基本信息，第二章给了一个纵览，第三章讨论如何开始运行这个系统。agent 的数据库描述在 4.5 和 6.2 节，参数的描述在 6.1 节。

2）agent 设计者 如果你想要创作一个 agent，让它在一个已有的 SPADES 仿真种作用，你也需要从第二章开始得到一个概览，学习第三章以获取如何着手的信息，然后可以阅读 4.5, 4.6, 4.7 5.5, 6.2, 6.3, 6.4, an 和 6.7 节。

3）世界模型设计者 如果需要创建一个使用 SPADES 的新的系统，就需要首先创建一个新的世界模型。你可以首先阅读第二、三章以得到系统的全貌，第四章是信息的主要来源，需要的话可参考稍后的几章，快速的浏览一下会发现它们是有用的。

4） 仿真的研究人员 关于 SPADES 的更加简练的摘要在一篇技术论文随同实验结果一起被提供，你可以从 Riley [2003] and Riley and Riley [2003]开始。在这片文章里，你可以阅读第二章和 6.8 节

5）对改进 SPADES 感兴趣的程序员 你必须顺序阅读全部的文档，然后你需要阅读很多的源代码和注释，如果还 有困惑可以和开发者讨论联系，邮件地址是：spades-sim-devel@lists.sourceforge.net

第二章　系统结构

这一章描述所有基于 SPADES 的仿真的基本结构，任何使用 SPADES 的人都可以从这里开始。

2.1 结构组成

图 2.1 给出了整个 SPADES 系统结构的概览，其中包含了必须由系统使用者提供的组件(图中的阴影部分)，模拟引擎和通讯服务器作为 SPADES 的一部分提供，世界模型和 agent 由模拟一个特殊环境的使用者提供。

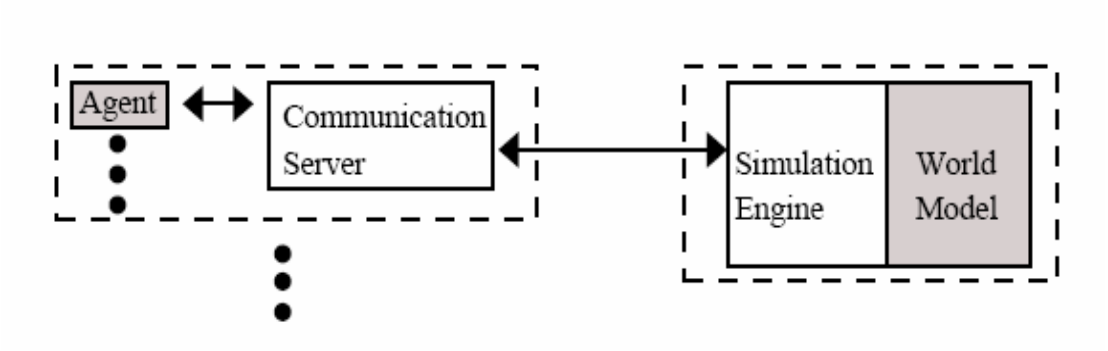

图 2.1

仿真引擎是离散事件模拟器的核心，所有待解决事件在这里排队，引擎调整网络通讯。在 agents 运行的每一台机器上都要运行通讯服务器。通讯服务器处理所有和 agent 的通讯(通过一个 UNIX 管道接口）并且跟踪 CPU 的用法来计算思考的反映时间（见 6.7.1 节）。模拟引擎和通讯服务器通过一个 TCP/IP 连接通讯。注意：如果需要模拟器和通讯服务器在统一台机器上运行，可以运行一个集成的通讯服务器，它与仿真引擎运行在同一个进程。

SPADES 的使用者为一个特殊的环境创建一个仿真模型，即世界模型。模拟引擎是一个库，世界模型必须连接到库上，以使模拟引擎和世界模型存在与同样的进程中。世界模型必须提供这样的功能比如，把世界状态推进到一个特定的时间、识别一个事件（即随着事件的发生响应改变世界的状态）。SPADE 提供一个 c＋＋类的集合，世界模型中的对象可继承这些类以与模拟引擎结合。

最后，agent 是计算实体，它们的交互被模拟。2.3 节更详细地描述了它们与系统地交互，但是 agent 与通讯服务器地通讯通过管道实现，因此 agent 能够使用任何编程语言和任意的结构，只要能够被管道读和写。

2.2　基于事件的仿真

SPADES 是连续和离散混合的事件模拟器。在一个连续的仿真中选择一个小的时间量子，仿真通过轮流处理每一个时间量子向前进；而在一个离散的仿真中仿真的状态由在离散时间里发生的一系列事件改变。

在与世界模型交互时 SPADES 连续的模拟器。模拟器命令世界模型前进若干时间量子。这对于一个潜在的连续过程比如物质交互尤其有用。

在与agent（可推及分配）交互时，SPADES是一个离散事件模拟器。假定agent不做任何事知道从世界接收到感知信息（但是注意，agent在一个特殊的时刻通过一个request time notify信息请求一个感知信息）。事件（比如感知信息和动作）构成了agent被仿真影响和影响仿真地基础。

世界模型主要的几件工作：第一，把仿真状态推进到下一个事件发生的时刻，这也正是模拟器要求的；第二，识别每一个事件，即把事件的作用效果加进世界状态；最后，产生要发送到agents的感觉以及创建基于agents的回复的事件。

SPADES保证事件的识别中遵循因果关系，比如没有因果相关地事件被乱序执行，这就

是说事件被识别的顺序可能不严格遵守时间次序。4.2节准确地描述了SPADES提供地保证以及agent的更详细的细节。

2.3 感知－思考－执行

SPADES把一个agent看成一个计算实体，它接收感知信息，做一些思考（比如处理），然后返回一些动作。agent被假定成在接收信息之外不做任何处理（但是注意agent能够请求它们自己的感知信息），因此从agent的角度看仿真的交互相当的简单：

1.等待接收感知信息。

2.决定一组动作，并把它们发送到通讯服务器。

3.发送一个done thinking信息来表示所有动作都已发出。

因为 SPADES 限制了一个 agent 只能在对感知信息答复时发送动作，系统提供给 agent 一个特殊的动作叫做请求时间通报（request time notify），一个时间通报的实质是一个空的感觉，agents 想对其他感觉一样对其应答以动作。比如，这允许 agent 在一个特定的时间发送动作，即使通常没有接收到感觉信息时。

感知－思考－动作循环的另一个重要概念是循环中每一步的延时，图 2.2 表示一个动作循环的时间线，考虑最顶端的一根线。时间点 A 表示感知信息产生的点，就像是机器人照相机的视频帧或股票市场价格的快照。点 A 和点 B 之间的时间表示传输和处理一个感知信息的时间，这个感知信息将被由 B 点开始的思考组件使用，比如这个时间可以表示一个机器人从照相机获得一帧并从中析取出信息的时间。在点 B 和点 C 之间，做一些计算以决定哪一个动作将被用来响应感知信息。在点 C 和点 D 之间，动作信息被发送到动作器，动作在 D 点产生影响，例如，这个延时可以表示把信息发送到动作器的时间。注意：我们并不是在说在感知组件、思考组件和动作组件中发生的计算有本质的不同。尽管如此，一个仿真系统有必要对正在模仿的世界作一个提取。因为仿真以程序的形式提供信息和接收动作，使用外在的延时允许仿真计算在真实世界中用来与程序形式互相转化的时间。注意，SPADES 不要求所有的感知和动作有同样的延时。

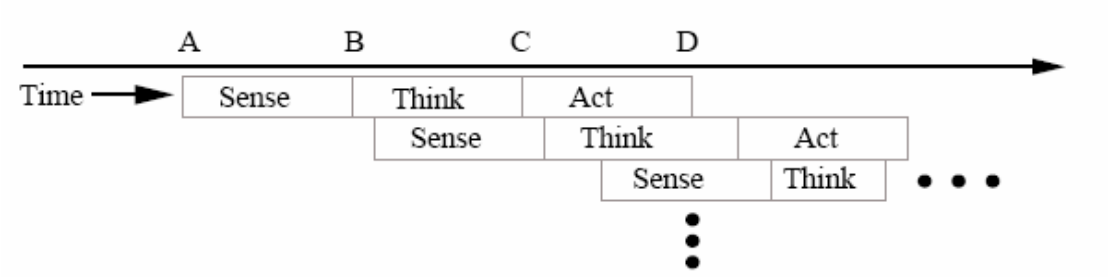

Figure 2.2: Example timeline for the sense-think-act loop of an agent

图 2.2

在很多 agent 中，感知、思考、动作组件可以像图 2.2 中描述的那样交迭，SPADES 明确地允许所有地交迭，只有一个例外，在我们的环境中，一个单独 agent 地思考循环从不交迭。这是真实世界中 agent 的合理模型，假定每一个 agent 使用一个单独的处理单元，当实际的 agent 不能而允许它在同一时间内处理几个感知信息是不合理的。

通讯服务器的一个主要工作是跟踪 agent 的思考时间来模拟思考延时。当发送一个感知信息给 agent 后，通讯服务器开始跟踪 agent 使用的 CPU 时间，这个时间由 Linux 的内核提供。当收到思考结束（done thinking）的信息时，通讯服务器计算用来产生这些动作的全部 CPU 时间，CPU 时间被转化成仿真时间（见 6.7.1 节）。所有的动作在思考结束阶段被加上同样的时间戳。

同样，如果使用瞬间计时器，当前由 LINUX 内核报告的时间片以 10ms 递增。由于中断的随机性和其它的系统行为，CPU 的使用数据并不是完全可重复。尽管如此，实验证明由此带来的影响很小。

使用 perfctr 计时器跟踪指令执行的数目可以减轻这个问题，但是当前使用 perfctr 计时器需要一个补充的内核。

第三章　开始

这一章是以 SPADES 开始的初级读物，讨论配置和安装步骤以及提供的一个世界模型和 agent 的例子的概览。

3.1 结构和安装

SPADES 使用GNU自动工具，因此可通过一下简单的几步完成：

```
./configure
make
make install
```

这一节包含了配置和安装的其它一些特征

3.1.1　配置参数

SPADES努力自动检测许多特性，如果失败，你就需要提供一些配置帮助，你也可以调整安装的参数，安装路径等，这一节将覆盖SPADES配置的特殊选项。关于自动工具常规配置的讨论，可以到这个网页上：`http://www.gnu.org/software/autoconf/manual/autoconf-2.57/autoconf.html`.

注意，所有以“使能（enable）”开始的选项都有一个“不使能（disable）”变量，”有（with）”选项有一个”没有(without)”变量。

允许调试（enable-debug，默认值关(off)）如果配置为开（on）则动作记录允许（注意，错误记录总是允许的）。推荐正在构造世界模型时使用这个功能，但是当模型构建好以后关闭此功能可以稍微减少 SPADES 的代码尺寸和加快执行速度。

使能例子代码（enable-sample-code 默认值 on）如果配置为开，世界模型和 agent 的例子被编译，这主要用来测试系统，对初学者推荐使用，当 SPADES 允许你自己的世界模型和 agent 时，这不是必须的。

允许例子监视器（enable-sample-code-monitor 默认值 on）决定世界模型和 agent 例子的监视器是否被编译，监视器是用 java 写的（其它都用 C++），所以在一些安装中这一部分不能被编译，不安装这个仍然可以运行 SPADES 和世界模型和 agent 的例程。但是将更难知道世界模型例子里正在发生什么。

指明 Perfctr 定时器的路径（with-perfctr=DIR） Perfctr 定时器需要 perfctr 系统，perfctr 系统是 LINUX 内核的一个补充，和一个存取信息的用户库。如果用户库不是标准的包含路径，可以用这个选项来显示路径在哪。如果内核补充的头在一个不平常的地方，你将需要用 `CPPFLAGS` 变量来帮助 `SPADES` 找到它们。

如果这个变量没有被指定，那么 `configure` 将试着探测 Perfctr 是否在，然后使能或使不能计时器。如果的确提供了这个选项，而在配置时找不到 Perfctr，`configure` 将退出。

没有 Perfctr 计时器，SPADES 仍然能够正确地运行；有其它地计时器可以使用（见 4.7.1 节）。但是它会影响再现性（见 4.11 节）。

定义 `fork` 的动态库地址(with-fork-dynlib-loc=PATHTOLIB) 为了？？动态库，调用 `fork` 和相关功能（见 `4.7.2` 节），`SPADES` 需要定位动态库，动态库定义了 `fork` 的功能。它尝试通过编译一个小程序和运行 `ldd` 和 `nm` 来实现这个功能。如果没有这些程序（或它们的输出是 `configure` 不能理解的形式），你必须准确地告诉 `SPADES` 哪一个动态库定义 `fork`。注意 `SPADES` 假定 `clone` 在同一个库中定义。

是否使用瞬间定时器（with-jiffies 默认值 YES）决定是否使用瞬间定时器，这个定时器使用 `/proc` 文件系统。如果因为一些原因这个不可用或者 `SPADES` 不能使用它，可以使它不能使用（如果真是这样，请在 `SPADES` 的网站上张贴窃听器报告/ 放映（bug report/feature）请求）。没有这个计时器，`SPADES` 也可以工作，但如果 perfctr 计时器和瞬间计时器都没被编译的话，将不能跟踪实际使用的计算。

Java 前缀（with-java-prefix=PFX）只有当编译世界例子的监视器时是相关的。给出一个 java 运行时间安装的地方的前缀。

Java 的标记（with-javac-flags=FLAGS）只有当编译世界例子的监视器时是相关的。提供给 java 编译器额外的标记。

3.1.2 安装的文件

这一部分给出 SPADES 安装的概览，但不给出一个完全的文件列表。所有的路径和安装是给出的前缀相关（默认是/usr/local ）。

bin/commserver 这是通讯服务器的可执行文件，因为通讯服务器不针对一个特定的仿真，所以所有的世界模型都可用同样的可执行文件。

bin/show tttimes.pl 这个 perl script 把一个记录 agent 思考时间的文件转化成人类易读的形式（见 4.7.1 节）。

bin/run exp.pl 这个 perl script 提供了一个为了测试而多次运行 SPADES 的方法。如果不作改变，它的用法可能很少。在这个手册中没有关于这个手稿的更进一步的文档。

include/spades/ 所有编译世界模型或 agent 需要的头文件都安装在这。

lib/ 世界模型必须链接上去的库，agent 设计者可选用的库放在这。关于 agent 接口库的信息，在 SourceForge 网页上阅读 agent 库的手册。

lib/spades/ SPADES 中心使用的库安装在这，知道这个可用来为 agent 的帧听库设置正确的值，虽然通常这是为你做好的。

share/spades/agent.conf 一个设置文件给出了一些合理的默认参数值，这些参数用来控制 SPADES 如何操纵 agent。这个包含在通讯服务器和模拟引擎中很有用（如果是在运行一个集成的通讯服务器的话），你很可能需要为你的仿真复制和调整它。

share/spades/commserver.conf 一个配置文件为通讯服务器提供合理的默认值，但是你很可能需要为你的仿真复制和调整它。注意这个文件包含了 agent.conf。

share/spades/agentdb.xsd xml 图解，描述了 agent 数据库的格式。

3.2 例子的世界模型和 agent

如果这是你第一次安装 SPADES 或者你想确认所有事情都像你假定的一样工作，试着运行例子的世界模型的 agent 是个好主意。

3.2.1 描述

例子的世界模型是“球世界”，例程的世界模型仿真的世界是一个二维空间的矩形，对边互连，比如说成环绕状。在这个世界里每一个 agent 是一个球，每一个 agent 接收到的每一个感知信息包含了仿真中所有 agent 地位置，每一个 agent 的唯一动作是请求一个特定的速度矢量。世界模型提供正确地加速度来达到这个速度。对于机器人在地毯的小的全方向的移动，动力学和运动特性如果不绝对正确也至少是合理的，但是世界模型中不包括冲突。注意，参数使用 2001 年 robocup 参赛的来自 Carnegie Mellon 的 CM Dragons 队所用的参数。世界模型以 1ms 的增量前进。

例程中有两种 agent，“流浪者”到处随机的移动，“猎人”通过把被请求的速度直接指向那个 agent 当前观察到的位置来追赶一个流浪者。注意，agent 并不试图在延时之前预知，延时指感知信息产生和动作被执行之间的时间，如果你注意 agent 的表演，你会注意到这个效果。

可以指定 agent 采用不同的计算时间。

3.2.2 运行

例程的世界模型没有被安装，所以你需要到编辑目录下去运行它。首先改变到sample world model目录，然后运行例程的世界模型，具体做法是：

```
./ballworld --file ballworld.conf
```

在版权通告和其它的启动信息之后，你会看到如下信息 ： Engine Status: PAUSED simtime=0 events=0 mean simtime/sec=0 你不能收到任何错误或警告信息，如果有你必须找出哪儿发生了错误。

这时候，世界模型和模拟引擎等待通讯服务器的连接，在本机上连通讯服务器，用下面这条指令（所有的在一行上）：commserver --file /usr/local/share/spades/commserver.conf --agent_db_fn ../sample_agent/agentdb.xml

注意：这要求你仍在例程的世界模型目录下，安装的二进制目录在你的路径中。

如果你需要把几台计算机连接到仿真中，你需要按下面的步骤进行。假定你想连接N他台计算机，运行例程的世界模型用这样的指令：./ballworld --file ballworld.conf --num_comm_servers_wanted *N*

如果运行仿真引擎的机器名字是*HOST,*那么每一个通讯服务器像这样启动（所有的在一行上）：commserver --file /usr/local/share/spades/commserver.conf --agent_db_fn ../sample_agent/agentdb.xml --engine_host *HOST*

如果你想只在一台机器上运行，你可以用以下指令运行集成的通讯服务器：./ballworld --file ballworld.conf --run_integrated_commserver

在以上哥哥情况下，你可以连接一个监视器到仿真中来看它如何运行，在编译路径下用这个指令： cd sample_world_monitor/monitor

./connect

这将把一个监视器连接到一个世界模型正在运行的当地主机上，如果想要连接一个远程机器的话，参照connect原本的工作方式，以及适当地访问java。

如果机器运行速度非常快，可能在你来得及连接一个监视器之前，仿真已经结束了，如果这个情况确实发生，试着增加仿真长度，或者在连接通讯服务器之前先连监视器。

有很多影响例程世界模型运行方式的参数，它们在6.1.4节中详细介绍。

3.2.3 日志文件

默认的，SPADES产生很多的日志文件，默认的，它们都在当前工作目录的Logfiles目录下。

agent*-stdout.log, agent*-stderr.log agent的标准的输出和错误日志见4.6.1节。

agent*-ttimes.log 这是agent记录思考时间的文件，见4.7.1节。

actions.log 这是记录仿真引擎的动作日志，见5.1节。

events text.log 这是一个人类可读的文件，包含了仿真中能识别的全部agent的列表。当前SPADES不能读进这个文件，所以它只能输出。参数*use_ text _ event _ log*控制了这个文件是否被创建，而*text_ event _ log _fn*控制文件地址和文件名。

monitor.log 通常这个文件由SPADES创建，但它的格式完全由世界模型决定（见4.8节）。对于例程世界模型来说，你可以检查例程世界模型监视器的这个文件，在编译路径下,用这条指令： cd sample_world_model/monitor

./playlog 播放文件用：../Logfiles/monitor.log 如果需要播放其它的文件，参考playlog原本，并作适当的改变。

第四章　创建一个SPADES仿真

这一章详细地介绍了使用SPADES创建一个新的仿真需要做的事，几个不是严格必需的特性将在第五章中讨论，许多技术方面的细节在第六章中给出，第六章比这一章更像一个参考手册。然后，SPADE 在sample world model和 sample agent目录下分别提供一个例程的世界模型和agent，这些将提供可构造的有价值的例子。

4.1 基本的仿真过程

这一节将从仿真的终端用户和世界模型的角度全面描述方针的过程。

4.1.1运行一个仿真

SPADES被设计成运行横越多个机器的仿真。会有一个中心机器用于协调和通讯，在中心机器上模拟器和世界模型作为单个进程运行。世界模型必须链接到模拟器库上以产生执行体，这个执行体必须首先运行。然后通讯服务器在每一个参与仿真的机器上运行。通过提供主引擎的参数给通讯服务器来确定主引擎，注意，对于在同一台机器上运行的通讯服务器，可以使用集成的通讯服务器（见5.4节）

通讯服务器启动和追踪agent，必须给每一个通讯服务器一个agent的资料库（agent database）以让它知道如何启动agent程序，当方针结束时（由世界模型控制）每一个通讯服务器监控agent的关闭，然后退出。

4.1.2世界模型的看法

首先，介绍一些概念，SPADES是基于时间仿真和连续仿真的结合。仿真的过程是先把世界模型推进到下一个事件的时刻，然后识别一个事件（比如在世界模型里产生它的影响）。

时间是由SimTime类型声明的SPADES的离散尺度。使用者必须创建主功能，SPADES只要求主功能两件事，一、分配一个世界模型类的对象，这个对象将是SPADES与其使用者的首要接口；二、必须调用SimulationEngineMain功能函数。注意，参数处理必须在调用SimulationEngineMain之前完成。

一旦在SimulationEngineMain函数里，世界模型对象的解析参数（parseParameters）方法被调用。注意，这个函数必须返回EngineParam类的对象。参数处理的推荐方法是使用EngineParam子类，调用getOptions函数。

一旦参数被解析，日志工具即被初始化（见5.1节）注意，这意味着在这之前发送到动作日志的任何信息都不会被记录。发送到错误日志的消息将仍会以标准错误形式显示，但不在动作日志中出现。仿真然后进入到主仿真循环。引擎可以在不同的仿真模式，有：

SM RunNormal　仿真的大多数时间处在这个模式。从世界模型的角度，调用simToTime方法把世界模型推进到一个指定的时间。然后，事件被识别。仿真做一些工作来完成这个任务，但是世界模型也有机会通过Event的realizeEventWorldModel方法来实现这个事件。注意对于事件的识别没有调用任何WorldModel方法。在将来事件将以SPADES提供的标准类型来分类。注意尽管保证事件的因果关系不被违反但事件不一定按严格的时间顺序被识别，4.2节将讨论这方面的细节

SM RunLimitedRate　除了仿真的速度被限制外，这个模式与正常的运行模式是一样的，5.6节详细讨论细节。

SM PausedInitial　有三个暂停模式，在暂停模式中仿真时间不再前进，也没有事件被识别。从服务器接收到信息后，WorldModel的pauseModeCallback函数被调用。如果仿真引擎在暂停模式逗留的时间太长达到了最大暂停模式时间，仿真将被关闭。这个暂停模式是仿真启动后进入的初始化模式。一旦仿真开始后就不能再返回这个模式。

SM PausedMonitor　当监视器要求仿真暂停时进入这个模式，其他同SM PausedInitial。

SM PausedWorldModel　当世界模型要求仿真暂停时进入这个模式，其他同SM

PausedInitial模式

SM Shutdown 表明仿真正处在关闭进程。 注意，世界模型可以通过调用changeSimulationMode函数和 SimEngine类控制仿真模式的改变。

4.2 **事件**

这一节更深一层定义了作为仿真的基本构造砖的事件，然后介绍世界设计者必须基于的SPADE提供的事件类。

4.2.1**定义**

事件是仿真的主要对象之一。正如在4.1节中描述的仿真的过程是先把世界模型推进到下一个事件的时刻，然后识别一个事件（比如在世界模型里产生它的影响）。

SPADES提供了一个类结构，创建你的仿真需要的子类都基于这个类结构。这一节将描述这些类和这些类如何在仿真中使用这些类。

有一个事件的特殊子类，叫固定agent事件（fixed agent events）固定agent事件的知识用于在agent之间达到更好的并行，它有一下的属性：

1. 他们不依赖于当前世界的状态；
2. 它们只影响一个单独的agent，通常通过给这个agent发一个消息；
3. 感知事件和时间通报事件都是固定agent事件；
4. 固定agent事件是唯一能引发agent开始一个思考循环的事件，但它们不一定激发一个思考循环。

把固定agent事件从其它事件中分离出来，SPADES提供的正确性保证可以陈述如下：

1.所有非固定agent事件按时间顺序被识别；

2.所有给agent发送感知信息的事件是固定agent事件；

3.一个特定的agent一批固定agent事件按时间顺序被识别。

6.8节提供了达到这个的算法的技术细节。

4.2.2**接口描述**

图4.1显示了SPADES事件以及它们的继承关系。SPADES的使用者创建的事件都必须是这些基本事件类型中的一个的子类。

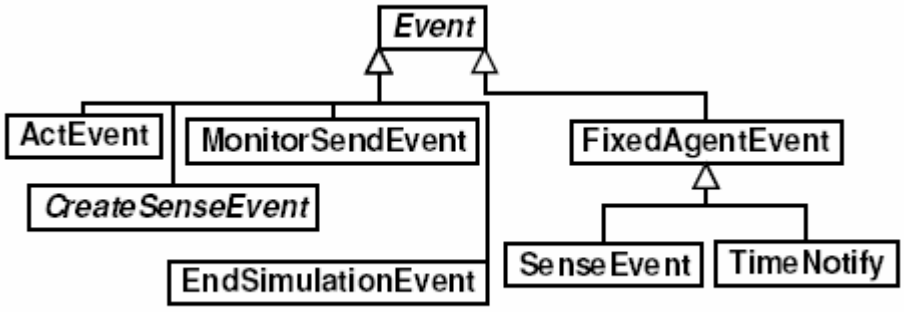

图4.1

下面描述这些类的函数和使用这些函数的方法。在讨论中一些细节将被忽略，要知道全部的信息请参照源代码。

事件 这是一个抽象基类，所有的事件都继承于它。Event维持事件的时间、提供子类要定义的虚拟函数。

重要的函数有：

- 构造器使用两个可选择的参数，事件时间和事件顺序常量。时间默认值时SIMTIME INVALID，但是使用这个时间的事件永远不能在仿真引擎中重新排序。顺序常量在getOrder函数中描述。

•返回事件的时间
•设置这个事件的事件。注意，永远不能改变正在排队的未决的事件的时间，因为这将引起错误。
•获取事件的顺序常量。所有的事件都有一个顺序常量来给有相同时间的事件排序。一般，不同类型的事件应该有不同的顺序常量。文件shared/sharedtypes.hpp定义了不同类型的一些常量。
•这是一个纯虚函数。如果两个事件有相同的时间和相同的主顺序变量（见getOrder），getSecondaryOrder函数的返回值被用来排列事件的次序。如果返回值也相同，那么SPADES将不能保证这些事件的次序。
•这是一个纯虚函数。事件的原文打印在各种日志和错误报告设备上使用。基类定义了ostreams 操作符<<来调用它。
•调用这个函数来识别事件。这是一个虚函数，所以不能重载（函数是虚函数的前提是固定agent事件可以重载它，其它的类都不可以），参见函数realizeEventWorldModel。返回事件能否被删除或者被保存在别处。
• 这个受保护的纯虚函数不为将被模拟引擎识别的事件工作，它在before realizeEventWorldModel被调用前由realizeEvent调用。不能重载，除非你改变了SPADES而不是用SPADES创建仿真。
•这个受保护的纯虚函数必须执行世界模型必需的工作来识别事件。这里是事件把它的影响加在世界模型上需要作的全部工作，返回是否对事件进行了操作。这个返回值主要用作错误检查以确保事件没有正被放进一个从不做事的队列。

动作事件 当接收到agent的动作消息时，这个类型的事件排序。WorldModel对象的parseAct函数创建这些事件（见4.3节）这不是一个抽象类，你可能从不想用它，但子类是抽象类，所以你可以重载realizeEventWorldMode函数。ActEvent仅仅比 Event增加了与事件相关的agent的存储。agent通过getAgent和 setAgent函数存取。

CreateSenseEvent 这是一个对创建感知事件有用的抽象基类(在4.6节看基本感知－思考－动作循环的描述和相关的事件)。虽然不是所有的感知事件都必须由CreateSenseEvent函数的实现创建，但强烈推荐大多数是这样。首先这样运许仿真引擎做额外的错误检测，为世界模型的设计者简化了感知的创建。CreateSenseEvent在基本Event类上增加了两件事，一是通过getAgent和setAgent函数实现agent标识符的储存，第二纯虚函数createSense由realizeEventSimEngin调用，createSense必须返回一个SenseEvent类型的事件，SenseEvent创建感知信息，然后这个感知信息被模拟引擎重新排队。注意CreateSenseEvent定义了getSecondaryOrder返回agent标识符，你很可能不需要重载这个函数。

EndSimulationEvent 这个事件的实现开始关闭仿真进程。推荐的结束仿真的方法是重新排序一个EndSimulationEvent类。

MonitorSendEvent 这个事件由仿真引擎内部使用来创建监视器日志文件。很可能世界模型不需要创建这类文件。

FixedAgentEvent固定agent事件是事件的主要子类。作为一个世界模型的设计者，因为它们不同的排序处理，你必须注意那些定义了FixedAgentEvent的子类。SPADES提供了FixedAgentEvent的两个子类。

1. SenseEvent（感知事件）

所有将要发送给agent的感知信息都必须有这个类型的事件发出。仿真引擎识别这个事件后产生一个将要发送给agent的信息。一个感知事件（SenseEvent）包含一个指定这个感知信息的思考部署的ThinkingType类的值，可能的值有：

TT Invalid 它不能在这里被使用，它的存在仅仅用来表明一个错误。

TT Regular 一个正常的感知信息启动一个思考循环。在agent类型中指定的计时模块被用来计算思考延时。

TT Untimed 除了计时模块的输出被忽略，思考延时被设定为零外，这与TT Regular一样，被称为不计时的感知。

TT Not 这是一个不思考的感知信息，也叫做一个通告。agent不能用动作来回复一个不思考的感知信息，在通告处理中使用的计算时间被加到下一个思考感知中。

SenseEvent的思考状态由函数getThinking and setThinking控制。

感知事件也给事件增加一个额外的时间，发送时间。每一个感知有一个发生时间和一个agent开始思考的时间；这个时间间隔叫做感知延迟。存储在基础Event中的时间是到达时间，发送时间是SenseEvent的一个受保护的元素，由getSendTime方法存取。

最后，SenseEvent维持将要发送给agent的数据，它被存储在受保护成员data中，data是DataArray类型的，DataArray将在4.9节中充分的描述。将要发送的数据可以是任意多的字节。

2.时间通报事件（TimeNotifyEvent）

每次agent使用request time notify信息请求一个时间通报后仿真引擎重新排序这类事件，不推荐使用世界模型来排序任何这类事件，因为这样会搞乱agent。使用感知信息和SenseEvent来代替

4.3 事件模型（World Model）

因为客户端只涉及agent，世界模型是与服务器（server）相关的部分，在这里略过。

4.4 仿真引擎接口（Simulation Engine Interface）

这一节描述仿真引擎类的方法，在这里最关心的是世界模型和相关的事件，所以略去。

4.5agent类型（ Agent Types）

仿真中的所有agent都有一个关联的类型。这个类型不费力地允许不同种类的agent在仿真中混合。agent数据库（agent database）存储了有关agent类型的全部信息。所有的agent类型有一个相关的名字来标识agent，所有的名字必须不包含空格，并且只包含字母、数字、“-”和“_”。

agent数据库文件格式的细节都在6.2节描述。

4.5.1 外部 agent（ External Agents）

外部agent以单独的进程启动，通过管道通讯。

对于一个外部agent，agent数据库允许我们指定：

•agent用来输入输出的文件描述器（见4.7.1）。

•使用的进程计时模块。

•agent工作路径。

•使用的包含参数的可执行代码。

4.5.2 集成的agent（ Integrated Agents）

一个集成的agent由一个动态库加载，它与他通讯服务器存在于同一个进程中。这就允许agent改变通讯服务器和不接受一些计时机制的使用。另一方面，它有比运行一个外部agent更快运行的好处。而且，如果运行一个集成的通讯服务器，agent可以存取世界模型的内在数据，减轻连载和重载信息的需要

对于一个集成的agent，agent数据库允许我们指定：

•包含agent的动态库路径。

•使用的进程计时模块。

•agent是否必须在一个集成的通讯服务器下执行

•agent启动时提供给它的参数

4.5.3 占位符agent（Placeholder Agents）

一个占位符agent是一个只有一个名字的agent类型。一个占位符agent不能由一个通讯服务器启动。它仅用于提供给模拟引擎以让它知道一个特殊的agent类型存在而不需知道如何启动它。注意，作为负责启动agent的服务器必须有一个类型的完整描述。

4.5.4 使用agent数据库（Working with the Agent Database）

世界模型的作者存取agent的数据库以启动agent。这一节描述了怎样存取agent数据库。

首先，agent数据库可通过SimEngine的getAgentTypeDB方法存取。这个方法返回一个指向agent类型数据库的指针，一个AgentTypeDB类型的对象。

AgentTypeDB 包含AgentTypeIterator and AgentTypeConstIterator类型，它们是存取agent数据库中的agent类型的主要方式。

AgentTypeDB下的方法对于agent类型相关的工作很有用。它们以两种类型出现，获取/返回规则的或不变的重述。

isIteratorValid 返回提供的描述是否正确（指向一个真实的agent类型）。

nullIterator 返回一个空的描述，比如是有缺陷的。

deref 取走一个描述其并返回相关的agent类型。你将不能废弃一个有缺陷的描述器（比如一个空的描述器）。Ta**getAgentType** 取走一个agent类型的名字，在数据库中找到它，返回一个agent类型的描述器。如果在数据库中没有哪个名字的agent类型，返回一个有缺陷的描述器

getBeginIterator 返回数据库中第一个agent类型的描述器。

AgentTypeIterator支持用++操作符来推进到下一个类型

4.6 agent接口（Agent Interface）

这一节从agent和世界模型的角度概要描述了agent的接口。准确的格式和其它的细节在6.4节讨论。

一个agent进程由通讯服务器使用agent数据库中的信息初始化。用来与通讯服务器通讯的管道像在6.4节中讨论的被初始化。

然后给agent一个初始化数据（initialization data）信息，如果在使用agent移动（5.3节），这就是一个移动的agent，数据中会包含移动的信息，如果没使用移动的agent或者这是agent的第一次执行，这个数据将是空的，将被忽略。

一旦初始化步骤完成，agent必须以一个初始化完成（initialization done）信息回复。这允许agent在启动时作任意数量的不计时的处理。

一旦agent完成它自己的初始化，它只需要等待从通讯服务器接收信息。能够从通讯服务器接收到的信息有很多种，但可以分为两类，一类开始思考进程，另一类则不。

感知和时间通报开始一个思考进程，在这个进程中agent响应以动作。所有的思考循环以一个完成思考（done thinking）信息结束，这个信息通报通讯服务器agent的思考已完成。

所有其它信息是不启动思考循环的（non-thinking）感知信息，agent不能响应以任何动作。注意用来处理这些信息的时间会被加到下一个思考循环。

SPADES涉及agent的两个主要动作。第一个是动作信息（act message），它的形式完全由世界模型解释。第二个是请求时间通报（request time notify），它要求一个在特定时间发送给agent的时间通报（time notify）信息。在思考循环的结束处，agent永远必须发送一个完成思考（done thinking）信息

这里必须提到有关agent接口的重要参数。通过跟踪agent的计算时间，agent可能落后于仿真。比如，按计划agent可能在时刻x接收到一个感知信息，但是因为上一个感知引起的思考，agent已经在时刻x+1。如果agent一旦在感知的到达之前的时间大于max_agentq_trailing_time,感知

被转化成一个通知，一个non-thinking感知。这通过将ThinkingType的类型改为TT Not实现。Max_timenot_trailing_time有同样的功能，但是对时间通报信息而不是感知信息作用。如果参数Send_agent_think_times是开（on），agent将接收到一个思考时间（think time）信息报告上一个思考循环用了多少时间，这是通报来的信息，不启动思考循环。
对于各种乱序和错误格式的信息错误，agent将发送错误信息。在6.4.1节有完整的列表。
可以建立各种时间限制来控制agent必须用多长时间思考。这个特别的可以用来预防一个故障agent妨碍其余的仿真，或者为此提供检测设备。4.7.3节讨论了这个过程。
SPADES为每个agent的标准输出和标准错误提供记录，但是 这个功能可以通过参数create_agent_logfiles关闭。参数agent_stdout_log_fpat和 agent_stderr_log_fpat控制这些文件的位置和名字。
因此agent可以使用标准输出开log或记录信息，通过核对所有agent的标准错误来确保仿真过程中没有发生错误是一个好主意。

4.6.2从集成agent的角度（Integrated Agent Perspective）

集成的agent已通讯服务器的交互与外部agent有许多相同（在4.6.1节中描述），他们的不同是：
•通讯由方法调用完成而不是通过管道。
•初始化时发送给集成agent的字符串可以在agent数据库中指定（通过initialize方法）。这允许类似于命令行参数的东西来指定外部agent。
• 集成的agent提供一个IntegratedAgent的子类的对象，调用IntegratedAgent的方法给agent发送信息。
•要给通讯服务器回发信息，集成的agent调用IntegratedAgentActions类型对象的方法，如果IntegratedAgent对象代表了agent。
•自然的，agent的标准输和标准错误也是通讯服务器的标准输和标准错误，因此zhnt大概需要打开自己的文件来做记录。
•4.6.1节中提及的对于思考的时间限制读不适用于集成的agent。
有关的类的完全描述在6.5节。

4.6.3世界模型的角度（World Model Perspective）

世界模型通过事件和一些函数的调用来与agent交互信息。图4.2从世界模型的角度显示了一个agent的感知、思考、动作循环。这个循环由一个 CreateSenseEvent类型的事件被识别开始。它重新排序一个SenseEvent类型的事件。介于CreateSenseEvent 和SenseEvent之间的时间是感知延迟。一旦识别SenseEvent，一个信息被发送给agent，思考循环开始。agent以一些动作作为答复，以使用的计算时间为基础，这些动作被指定一个时间。SenseEvent的时间与分配给返回的动作信息的仿真时间的区别是思考延迟。WorldModel对象的parseAct方法被调用来把一个动作信息转化为一个ActEvent类型的事件。这个ActEvent然后了解到必须完成循环。在动作信息和ActEvent被知道之间的时间叫动作延迟。

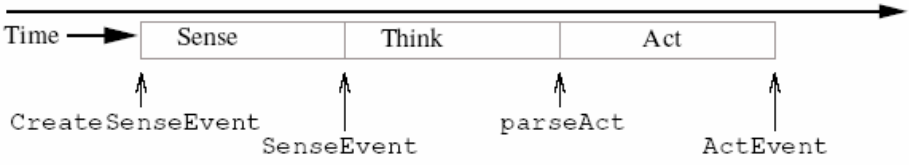

Figure 4.2: Time line showing the events and method calls relevant for the sense think act cycle of the agents

注意一个CreateSenseEvent并不是使SenseEvent处于排队状态的唯一方法。尽管如

此，推荐可能的时候都使用它。SPADES对CreateSenseEvent做额外的错误校核来避免很难用其他方法检测出的因果关系问题。

CreateSenseEvent能够被容易的用来创建有规律的重复发生的感知。对CreateSenseEvent的识别将引起createSense方法被分类以创建SenseEvent。然后你可以定义realizeEventWorldModel方法使一个新的CreateSenseEvent在将来的一个合适的时间排入队列

4.7 agent监测（Agent Monitoring）

这一节覆盖了SPADES通讯服务器负责的监控agent的所有方法。

4.7.1agent计时器（Agent Timers ）

SPADES提供了很多种方法来测量或计算agent的思考时间。注意世界模型也可以产生不计时的感知，当他们开始思考循环时，思考延时总为零。

不管用什么方法，SPADES都可以记录仿真使用的思考延时。参数*record_ think_ times*控制了是否计时。对每一个agent，时间存储在一个独立的文件中，参数*think_ times_ file_ pattern*指定了文件名。关于文件格式（是二进制而不是文本格式）的细节可在6.7.3节找到。

提供了一个叫show ttimes.pl的perl手稿可以阅读二进制，并输出一个更适合人类阅读的形式。它取走一个要阅读的文件的名字，随意地每一行显示用了多少思考时间。文件中的数值从左到右然后从上到下排列。

一些计时器能够被集成的agent（由动态库加载的agent）使用，另一些则不可以，这将在下面描述。

agent可以在agent类型数据库中指定，或者使用*default _process_ timer*参数。在这两种情况下，解析一个字符串来得到计时器类型和参数，正确的值有：

•固定的时间（‘fixed *time*’）

这是固定时间计时器。每个思考请求被给与相同的时间*time* 这可用于一个集成的agent。

•重放文件形式（‘replay *filepattern*’）

这是重放计时器。*Filepattern*参数指定了要阅读的文件。就像在*think_times_file_pattern*,字符串‘%N’由agent的号码替换。6.7.3节这个文件格式，但是仅仅为了重放SPADES之前记录的时间，没有必要理解这个。

在重放时，来自文件的变量*replay_ think_ buffer_ size*同时被阅读。改变这个参数会有一个有效率的提高，虽然可能对性能有许多小的影响。

这可用于一个集成的agent。

• ‘jiffies *kinst*’

这是瞬间计时器。一个瞬间是由Linux内核报告的预定在处理机上的一定数量的时间。整型参数*kinst*转化为每个仿真步的千条指令数，细节见6.7.1节。这个不能用于集成的agent。

• ‘perfctr instr *kinst*’

这是perfctr计时器。通过使用由perfctr系统提供的特性，agent使用的处理器指令数被追踪，计数。整型参数*kinst*显示了每个仿真步的千条指令数，细节见6.7.2节。这个不能用于集成的agent。

•默认值

这是默认值计时器。无论使用哪一个计时器，都使用参数*Default_ process_ timer* 指定。如果*Default_ process_ timer*的值是‘default’则使用‘fixed 0’如果默认的相关的计时器可以被集成agent接受，那它就可以用于集成agent。

4.7.2 agent过程跟踪（Agent Process Tracking）

当外部agent进程产生别的进程时SPADES也试图跟踪以能够完全的给agent计时，就像4.7.1节讨论的。这一节描述了这个这个是怎么实现的，以及相关的参数。尽管如此，必须注意这

个图解既不完美也不坚固。他的原意是使一个agent设计者的工作更简单，并且试图避免意外的错误。这样agent设计者还是可能愚弄系统。也要注意这仅仅应用于外部agent，而不应用于集成的。

SPADES的跟踪工作使用Linux的LD_PRELOAD来加载一个动态库，他截断了动态库的调用。所有动态库均调用fork, vfork，库抓住clone来通报服务器。注意，调用pthread create也执行使用clone，所以他也被跟踪。参数*agent_ intercept_ library*控制了预载哪个库。大体上，SPADES会自动设置这个，所以你不需要担心这个。

注意，这意味着任何不使用动态库的agent调用的新的分支进程（比如一个可执行的agent是静态连接的）将不会被准确跟踪。通常，SPADES不做任何事来试图效验agent是动态链接的，而且要试图效核所有的fork调用是动态进行的时非常困难的。

分支的通报通过SysV IPC Message Queue的配置实现。一个含意是每个使用者每台机器智能运行一个通讯服务器，另一个是如果一个通讯服务器不正常退出，下一次你会得到一个这样的错误：The IPC message queue seemed to already exist.如果你使用参数 *ipc_ force_ remove,*信息队列将会在通讯服务器启动前被移走。你可以通过使能参数*ipc_ message_ reception*来使得IPC信息不被接收

4.7.3agent的校核（Checking on Agents）

在每一个信息处理循环的末尾，通讯服务器检查agent。这主要是用来捕捉像agent碰撞这样的错误，当然也允许处理一些家务管理。步骤是：

•确认作为agent一部分的所有进程仍然存在。如果它们不存在，这不是一个错误。只是将那些进程从追踪进程中移走。尽管如此，如果所有agent进程都没有了，agent将以状态ALR ProcessVanished.关闭。

•agent思考已用的时间可以被计算。如果它比*max_ secs_for_ agent_ think*大，agent将以状态ALR ThinkTooLongWallClock关闭。推荐你设置参数*max_ secs_for_ agent_ think*完全高于预计的agent思考时间。理想的这个校核只抓住无限循环的agent或发送done thinking信息失败的agent。例如，在我当前的例程的世界模型和agent的安装中，我预期agent的最大思考时间是50ms，但我设置参数*max_ secs_for_ agent_ think*为10s。

• 基于计算的wall clock思考时间决定是否做更深一层的核对。如果*agent_ check_ use_ randomness*是关而且wall clock思考时间大于 *agent_ check_ threshold_ sec*就做更深一层的校核。*Agent_ check_ use_ randomness*是开就用参数agent check gumbel dist A 和 agent check gumbel dist B和输入作为wall clock时间来计算一个从Gumbel分配的开端。在那个概率下，做更深一层的核对。

•过程计时器（见6.7节）用来作为计算一个agent在仿真中已经使用的时间的参考。如果这个时间大于*max_ simtime_for_ agent_ think*，agent以状态ALR ThinkTooLongSim被关闭。

•如果agent用来思考的仿真时间增加了，一个时间更新信息（time update）被发送给仿真引擎。这个对世界模型和agent必须是完全透明的，这个时间更新不能对仿真的结果有任何影响，但能影响非因果关系的相关事件的效率的调整。

4.8监视器（Monitor）

监视器的接口允许外部程序连接到仿真，得到有关仿真世界的周期性的更新。监视器也能回发现信息给引擎和世界模型来执行各种功能像暂停和取消暂停。

仿真引擎只接受*accept_ monitor_ connections*为真的监视器了；连接。监视器必须以TCP套接子连接到端口*monitor_ port*。在每一个连接之后，引擎调用世界模型（WorldModel）的getMonitorHeaderInfo方法。这个初始化信息或头信息被发送给监视器。SPADES不限制信息的格式。

在时刻零和每一个*monitor_ interval*仿真步之后，一个MonitorSendEvent被重新排序。这个事件的识别引起方法getMonitorInfo 和WorldModel被调用，返回的信息被发送到所有的监视器。SPADES不限制信息的格式。
世界模型也能通过SimEngine的sendExtraMonitorInfo方法发送额外信息，这个可用于任何不同步的更新。
SPADES也能创建一个监视器记录文件来仿真中将会发送给一个连接的监视器的所有信息。如果使用监视器记录，所有监视器信息（包括头信息）都被写进*monitor_ log_fn.*
监视器能发送命令来暂停、正常运行、以限制的速率运行（见5.6节）、从仿真断开、关闭仿真以及向世界模型发送人意的字节。6.6节描述了监视器接口的细节。

4.9数据排列（Data Array）

DataArray是一个有用的类，它包含了计算字节排列的参考，并存储字节的数目。一个字节阵列的简单拷贝由在一个DataArray对象中任意多的参考的压缩支持。当这些参考的最后一个被删除，数据阵列即被自动删除。在同样的信息需要发送给很多agent的场合，必须提供重要的存储。这一节将着重解释这些重要方法的一部分，请阅读源代码以得到精确的细节。

• DataArray(const char* buffer, unsigned length)
这个构造器拷贝给定缓冲区的数据。如果你想DataArray取代（take over）存储器，见takeData。

• DataArray(std::stringstream& str), DataArray(std::streambuf& strbuf),
DataArray(std::string str)
所有这些构造器拷贝指定对象中的数据。

• DataArray(std::ostrstream& ostr)
这个构造器取代包含在ostrstream中的存储器在这个调用后，你不能调用ostrstream的freeze(0)或对ostrstream有任何操作。

• void copyData(const char* data, unsigned length)
这个方法用给出的数据的拷贝替换当前存储的任何数据• void takeData(char* data, unsigned length)
这个方法不拷贝数据，但是为接替传进的存储器的责任。在这个调用之后你不能对data中的数据作任何操作。

• const char* getData() const
对于存取实际的字节阵列没有用，但你不能存储这个值。如果你需要保留一个参考，可简单使用拷贝构造器DataArray

• unsigned getSize() const
这个返回由这个对象控制的阵列数据的字节数。

4.10达到同步（Achieving Parallelism）

当一个仿真运行在若干机器上时。要达到好的加速是一个棘手的问题。SPADES使用的算法的技术细节在6.8节描述，但这一节将会描述其中一部分，并给数据模型的设计者就如何达到好的并行加速提出建议
对这里描述的各种不同设置的影响做一个彻底的实验分析这里还没有完成，所以这里的建议是基于算法的知识和基于仿真运行SPADES的实验。
SPADES通过两个机制实现同步。第一个是识别每一个agent可以下次重新排序一个动作的时间。这就允许不需要必须等待agent的回复来识别事件。这考虑了最小动作延迟（minimum action latency）来确定最小agent时间（minimum agent time）。minimum agent time是一

个agent能够重新排序一个新的动作的最早时间。WorldModel的getMinActionLatency方法给出了最小动作延迟。

大体上，最小动作延迟越长，越能达到更好的并行因为同一时刻能够识别的事件越多。这意味着必须避免对任何特殊种类动作使用非常小的动作延迟必须被避免，因为最小动作时间是仿真中所有动作的最小时间。

另一个机制包含了识别非因果相关的事件，这样他们能够不安顺序被识别。这里感兴趣的事件是感知（sensations）和时间通报（ time notifies）因为sensation 和 time notify信息是固定的，一旦这些事件被识别就被实现，在这个事件之前不会有任何信息发送给agent。这个决定围绕最小感知延（minimum sensation latency），它是介于一个事件被识别的时间和一个agent的一个新的感知信息能够被重新排序之间的最小时间。最小感知时间是一个新的感知能够为一个agent激发的最早时间。在最小感知时间之前的agent的感知或时间通报信息可以被识别。

最小感知延迟越长则达到的并行越好。因此你必须避免任何特殊情况具有短延时的感知，因为由方法getMinSenseLatency返回的最小感知时间是激发事件的所有类型感知的最小值。

如果agent是并排的这样感知（比如思考循环）在同一个仿真时间被激发，这样就很容易达到并行。因此在你能够创建像这样的情形的场合（比如世界模型设计者），你就必须这么做。注意，尽管如此，你绝不可牺牲真实，因为产生非同步动作的能力是SPADES的实力之一。例如，对于场地上的一组机器人，你不可能期望同步的视觉更新，而且也不推荐这么做。注意，SPADES便于我们获得在一个指定的时间利用同步所能达到的任何好处。在由SPADES操作的实验中，感知被激发的时间有一个小的随机成分。这有利于确保在平均上有并行，agent分布和感知时间不好的情况不会延续很久。

另一个关于并行的有趣的点是机器间负载平衡。在先前的实验中，经常是在一台机器上agent最大数目减少后才能达到加速的并行。例如，有十二个agent在四台机器上和五台机器上运行时看不到额外的并行加速（因为每种情况下都有一台机器有三个agent）。尽管如此，在五台机器上和六台机器上运行就可看到并行加速的提高，因为后者每台机器上最大agent书减到了e二。

agent的移动可用来帮助执行负载平衡。尽管如此当前用以作移动决定的算法并不非常的富有经验。这个希望在将来有所改变。

4.11随机性和再现性（Randomness and Reproducibility）

另一个在很多仿真中想得到的重要特性是再现性。这里再现性将意味着通过在相同的启动配置下重新运行能得到同样的仿真状态。

第一， SPADES提供的精确排序的保证在4.2.1节讨论。因为一个固定agent事件不能以任何方式影响世界状态，初步的保证是所有非固定agent时间按时间的顺序被识别。

对于所有真实的重现，你可能想要一个更强壮的保证，而不仅仅是一个时间顺序，因为在同一时间可能会有许多的事件。特别的，你可能想要事件识别的特殊顺序的保证。为了达到这个，你必须保证你适当的定义了顺序恒量和Event类的getSecondaryOrder方法以及它的子类。那就是，你必须确保没有两个不同的事件有相同的顺序恒量，

getSecondaryOrder有相同的值

其次，WorldModel类的parseAct方法不能以任何方式影响世界模型的状态。parseAct的调用不受时间的控制，所以不允许任何状态改变。

如果你在仿真中使用伪随机性，事情会稍微复杂一点。在动作的解析（比如获取动作延迟）中使用随机性有一些自然的原因。尽管如此，为了得到重现性，你不能在parseAct方法和任何固定agent事件的识别中使用伪随机性。如果你确实想使用随机的动作延迟，一个可能

是在当前时间加上最小动作延迟为解析动作放置一个空白的固定事件。那个事件的识别能使真的动作事件排在以后的一个随机时间。

又，agentConnect方法不能使用伪随机因为连接的顺序不可控。

SPADES提供一些功能来帮助处理伪随机。SPADES 操作了随机数字激发器的开始，这样世界模型不需要做这个。参数*random –seed*允许你指定使用什么随机激发。又，如果参数*print –random- seed- to- stdout*为真，那么使用的随机激发（由*random –seed*指定或由/dev/urandom读出）被打印到标准输出上。这允许你在需要时可以重新创建仿真。最后，如果你在使用agent移动，你可能需要使用参数*use- randomness* 来控制是否移动选择是随机的。

提供完全的可重现的结果的最困难的部分是跟踪agent思考时间。在瞬间计时器中，由内核报告的ＣＰＵ时间基于其它不可预知的系统动作而变化。为了稍微减轻这些问题，你可以使用固定的时间计时器（见4.7.1节）来关闭跟踪，而且总是返回一个固定的思考延时值。作为选择，你可以使用perfctr计时器来得到准确的指令水平的准确计时。注意，尽管如此，当前要求一个内核包。而且思考时间可以由重放计时器记录和回放。（见4.7.1节）

这是为达到完美的重现性有另一个重要注意点。把simToTime方法写成以下这样很自然：

```
for (SimTime t = time_curr;
       t < time_desired;
         t++)
{
per step code
}
finish stepping code
```

也即是simToTime方法通过持有*finish stepping code*节的代码可能能够分期清偿一些单步向前的代价。特别当使用浮点计算时，会导致失去完美的重现性。simToTime方法并不被保证在每一次运行中都以完全一样的方式被调用。改变浮点数低位的顺序最终能够改变全面的结果。因此，如果你想要完美的重现性，*finish stepping code*必须不包含任何影响世界模型状态的东西。技术论文Riley 和Riley [2003]更深一层地讨论了重现性的这些论点，并提供了有关SPADES重现性的一些实验证据。

第 5 章 辅助功能

这一章包含第 4 章中没有讲到的 SPADES 的两个功能。这在建立 SPADES 仿真和理解系统的基本功能不是重要的，但是，如果你想更多地使用 SPADES，这些功能会对你有用。

5.1 动作和错误日志

SPADES提供错误，报警和动作日志工具。错误是指不该发生的内部错误，但是，并不是所有的错误都是SPADES的Bug。报警不是指系统设计的错误，而是指仿真不能正常运行的异常情况。动作日志是基于分层次的揭发思想[Riley et al., 2001]，用来跟踪调试系统动作。

建议你使用这些工具实现自己的日志功能。

5.1.1基本使用

这些类定义在Logger.hpp中。基类是Lgger并遵循单独开发的模式。这就意味着Logger只有一个实例，并且很容易地使用实例的成员函数。

一些#define声明用来提供更简单的Logger工具，并用标准的C++文件流写Log文件。下面的#define声明表示使用操作符<<发送信息给logger。结束log表达，使用ende操作符。注意，它们不是真正的流类型，所以不能进行复杂的输入输出（如STL）。

- errorlog 用在关键的不该发生错误的情况下。所有的信息是以标准的错误输出，在动作日志文件中是0级（如过使用的话）。
- warninglog(x) 使用情况在发生异常情况，除了系统中不可挽回的错误和Bug。X值代表日志的等级；我目前所有警报都使用10级，但你可以自己分配。所有的信息是以标准的错误输出，在动作日志文件中是0级（如过使用的话）。
- actionlog(x) 记录程序执行的信息。这里不要担心写数据的影响，因为一个编译标记可以用来移除所有的actionlog代码。X值代表日志的等级。分层揭发的思想的缺点是怎样确定正确的日志等级。一般来说，越高等级意味着越细节的信息，logger提供一种只打印某一等级日志的方法。所有的信息输出到action log文件中。默认情况下，logger输出仿真时间和dash的次数，相当于10级所区分开的。SPADES使用下列规定：
  - 越低等级的是越粗略的信息
  - 等级是10的倍数，以防使用这之间的等级
  - 200级以上的log使用在内循环里会很快产生大量的输出。这只能使用在有目的的调试中
  - 100级就能提供足够的信息来理解仿真的一般控制
  - 所有等级都是非负的

这里有一些正确信息的例子，和它们在SPADES中怎么输出：

```
errorlog << "This is an error condition" << ende;
warninglog(10) << "I'm feeling a little sick today" << ende;
actionlog(150) << "I started running at time: " << time << ende;
actionlog(50) << "I just realized an event for you" << ende;
```

注意在log的结尾使用ende。？

这些例子将以标准错误格式输出：

```
EngineError(1234): This is an error condition
EngineWarning[10](1234): I'm feeling a little sick today.
```

注意警报信息给出了警报的等级（10）。圆括号中是出错的仿真周期。

actionlog例子的输出：

```
1234 EngineError(1234): This is an error condition
1234 EngineWarning[10](1234): I'm feeling a little sick today.
1234 ---------------I started running at time 12/5/2002 12:12p
1234 -----I just realized an event for you
```

注意‘-’代表日志的等级。

5.1.2参数

如果定义了NO_ACTION_LOG（例如使用-DNO_ACTION_LOG标记来编译），所有的actionlog声明都被禁用，并从编译中移出。

这里有一些影响logger工作的参数。

- action_log_fn 动作日志的文件名
- action_log_level 哪些命令日志被写入文件。所有小于等于这个值的等级都将写入日志文件位置。高于这个等级的将被忽略。等级0时，只输出错误。

5.1.3高级使用

上面的部分没有给出一个完整的例子。Logger事实上比上面的特殊例子更一般。这个部分并不给出一个完整的描述，但给出一些工作的特点。你可以自己看代码得到细节。

Logger使用TagFunction作为每个日志语言的头，包含错误、警报和动作日志。头包含（在上面的例子中）仿真时间，‘-’作为动作的头，”EngineError”和”EngineWaring”作为标记。

错误日志和警报日志使用相同的工具；错误就是0级的警报。

如果你想改变哪里输出日志，你可以使用setLoggingStreams函数来设置logger的C++文件流输出。

你必须在结束程序的时候注意一些东西。Logger::removeInstance用来移除那个单独的实例。并且，文件直到明确调用才打开，所以，动作日志在初始化程序的时候将输出标准输出代替动作日志文件。

5.2读取参数

这个部分讲述了读取参数的借口，像在程序中的代码见到的一样。6.1节描述了配置文件和命令行中参数的格式。

所有的参数类都是类ParamReader的子类，类ParamReader提供基本的运行功能。重要的读取参数函数有：

ParamReader 构造函数有一个最大的版本值。所有的配置文件都需要一个包含文件版本的版本行。如果文件的版本高于最大版本值，将输出错误。

getOptions 这个函数进行参数和文件的分析。

addAll2Maps 这个虚函数将所有的参数添加到数据流中。每个子类中，你需要自己定义自己版本的addAll2Maps来调用add2Maps。重要的一点是每个子参数类调用父类中的addAll2Maps，如ParentParam:: addAll2Maps.

setDefaultValues 这个虚函数用来设置默认值。这用来初始化给定参数没有设置的参数。重要的一点是每个子参数类调用父类中的setDefaultValues，如ParentParam:: setDefaultValues.

postReadProcessing 这个函数在参数值读取之后使用，也许需要调用几次。重要的一点是每个子参数类调用父类中的postReadProcessing，如ParentParam:: postReadProcessing.

Add2Maps 这个重载函数用来给各个参数设值。请看代码的细节，但基本地，当读取的参数在maps中没有找到时，你的类将被通报。

为了创建你自己的参数类，你需要继承ParamReader或者子类，重定义上面列出的虚函数。如果你继承了除了ParamReader的其他类，确保你在使用assAll2Maps,setDefaultValues和postReadProcessing函数的时候按照上面的说明使用。

你也许希望你的参数类也遵循单独设计模式，因为这种类型只有一个实体。见例子EngineParam。

为了给参数类增加参数，你需要做4到5件事情：

1. 增加参数类中数值存储的长度。目前SPADES的规定是参数是小写字母，参数之间用下划线隔开。
2. 增加参数的通道函数。规范是在参数名之前加get，每个词的第一个字母大写，没有下划线。
3. 在函数addAll2Maps中适当调用add2Maps。规范是参数的外部名称和内部名称一样。
4. 在setDefaultValues中增加默认值设置。
5. 可选地，你可以在postReadProcessing中增加代码，来进一步处理参数值。

5.3移动Agent

移动Agent就是在不同的communication servers之间移动agent，来试图得到更好的平衡和改进仿真的整体效果。仿真的速度经常由最慢的agent的速度限制，所以改变平衡有改进仿真效果的很大潜力。

移动Agent是由SPADES支持的，但实现机制并不高深。支持移动Agent使重要的负担给agents自己，所以在有些情况下不值得用这个功能。

移动由migration参数控制。如果migration打开，仿真引擎跟踪agent感觉反应时间。一旦接受到agent发出的min_responses_before_migration，这个agent就是migration的候补。如果agent反应时间大于所有agent的时间的两倍，这个agent将被移动。

当agent被移动之后，首先移动请求信息发送给agent。Agent返回移动数据信息，包含agent当前状态的反映信息。Agent进程将被关闭，在另外一台机器上开始agent进程。初始化的数据就是结束的agent发送的移动数据。仿真接下来正常进行。

5.4完整的communication server

仿真引擎提供一个完整的communication server工具。这就是说communication server与仿真引擎和世界模型运行在一个进程。因此，信息不需要串行化发送到端口。

communication server与仿真引擎运行在一台机器上可以减少通讯的开销。在一些不典型的实验下，速度能提高5%。

使用整个communication server是由参数integrated commserver控制的。所有的agent和世界模型不可分布的是communication server是否使用的根据。

5.5agent退出管理

当communication server希望一个agent进程退出，必须通过以下步骤

1. 如果communication server发出退出命令，但是agent没有请求或者agent部分出现一些问题，发出退出信息。
2. SIGTERM发送给agent进程。
3. 周期性地检查agent进程是否退出。
4. 如果agent进程退出了，子进程必须立即退出，所以死的进程必须在一定时间后退出。
5. 如果在sec_for_agent_shutdown秒后线程还没有退出，将发送一个SIGKILL强制杀死进程。这是必须做的，警报将写入警报日志中（见5.1节）。

注意，如果你的agent在仿真时间里写数据或者完成密集的任务，你可以把secs_for_agent默认的时间延长到2.0秒。

5.6限制速率的运行模

SPADES支持限制的仿真速度模式。这是很有用的，例如，为了获取人机接口或监视的“实际时间”。注意SPADES并不保证获取速率，只是保证仿真不超过给出的值。

通过设置SM_RunLimitiedRate来改变仿真限制速度。这个可以通过world model或者monitor的命令来实现（见6.6节）。

在限速模式下，参数limited_rate_default_st_per_sec控制仿真速度。SPADES目标是进一步以limited_rate_default_st_per_sec限制速度。注意，仿真引擎每秒输出大量的仿真时间，所以你可以跟踪完成的期望表现。

这种算法是很直接的。调用每秒期望的仿真周期数R。使用一个基本的wall clock（Bw）和仿真（Bs）。在意识到事件之前，期望事件t的wall clock是这样计算的：

$$\frac{t - B_s}{R} + B_w \qquad (5.1)$$

如果时间没有到，仿真引擎sleep（select系统调用）直到事件的期望时间。

这个算法有一些注意点。首先，wall clock是唯一事件实现的。第二，如果实现事件的进程密集地计算，有可能SPADES落后，哪怕有足够的可用资源。最后，正确的agent事件不用同样的方式控制，如果意识到下一个正常事件没有到达就可以使用同样的前向算法（见6.8节）。

仿真周期改变到限制速率模式，为基本的仿真和wall clock完成”rebasing”。另外，每limited_rate_rebase_interval秒完成rebasing。注意，如果机器或网络降低速度，仿真将不确定地试图赶上。

第 6 章 技术细节

这一章描述 SPADES 各个方面的种种技术细节。手册中其他部分有太多细节，这章是一个总结。如果你只是想一般地理解 SPADES 怎样工作，这一章可以略过不读。

6.1 参数

这一部分是理解 SPADES 的参数。一般说来，只给出参数的主要描述。需要更详细描述的参数在其他地方有详细介绍。看参数的列表找出这些信息的地方。

所有的参数以一两种方法给出（param 是参数名，value 是参数值）：

- '—param value'作为命令行
- 在配置文件中，有一行：'param: value'

  配置文件可以分别通过命令行'--file filename'读取或者在配置文件里内嵌'file:filename'。最大的内嵌配置文件数目是 16 个。file 命令的路径是配置文件目录的相对路径（不是当前工作路径）。

  另外，每个配置文件必须分配一个版本号。版本参数确保读取的配置文件至少是最新的（见 5.2 节部分的参数接口）。配置版本行必须没有注释和空格，格式像这样：'version:num'，其中 num 是文件的版本号。

  至于参数的顺序，所有的配置文件以其他命令行之前的—file 命令行区分的。为了嵌套配置文件，完整的子文件在 file 命令头中，然后是剩下的原始文件。

  有八中类型的参数：

  String 任意长度的字符串。如果没有特别的值，设为空字符串。

  File Path 这仍旧是一个字符串，但不像 String 一样设置一个绝对的路径。在命令行中，这是相对当前工作目录的相对路径。在配置文件中，是配置文件目录的相对路径。有两种例外的：如果是以'%'开始，这个值是不能得到的（这是因为一些参数是用'%X'格式来扩展值）。另一种情况是以'&'开头，'&'市的剩下的值都不可得到。这是允许的，例如，一个特别的配置文件的路径是相对于当前工作目录的相对路径。

  Integer 一个整型值。一般需要赋一个值。

  Real 一个 real 值，保存为一个 double 型浮点数。一般需要赋一个值。

  Boolean 一个 on/off 值。这不需要一个值。没有值的时候，参数是 on。值只能设为 on 和 off.

  Vector(Integer) 一个整型矢量容器。最好和（随意的）最大值都可以设置。如果这些都包含将输出一个警报。这个手册中，范围用[ ]括起来。配置文件中的值以空格区分开，例如， '2 12 47'.

  Vector(real) 一个 real 矢量容器，于 Vector(Integer)同样的属性。

  Vector(String) 一个字符串矢量容器。完整地存储在配置文件中，在命令行中，完整的字符串矢量容器不能用--开始（因为这看起来像下一个参数的开始）。

  注意，Vector(Integer)、Vector(real)和 Vector(String)不能用在 SPADES 的所有参数中（简单的 world model 使用其中的一些），但是为了方便 world model 的设计者。

6.1.1 共享参数

这些参数是 communication server 和 simulation engine 都能理解的。

名称	类型、描述	默认值
version	布尔型。如果指定，将输出版本号并且 communication server 和 simulation engine 将退出。	off
logfile_dir	文件路径。这个目录用在其他一些参数上替换'%D'。这样可以很轻松地改变所有日志文件的目录。	Logfiles
create_logfile_dir	布尔型。如果 logfile_dir 不存在并且这个参数指定了，就创建日志目录。只创建一层目录（像 mkdir,不像 mkdir –p）。	on
action_log_fn	文件路径。动作日志的文件名。字符串'%D'被 logfile_dir 代替。	% D/action.log
action_log_level	整型（>=0）。小于等于这个值的日志将被记录。	0
engine_port	整型。Simulation engine 监听 communication server 的端口。	12000
use_randomness	布尔型。是否使用随机打破平局。现在只使用在 agents 的移动。	on
random_seed	整型。任意的负值意味着读取一个随机结果。任意一个正值指定随机结果。	-1
print_random_seed_to_stdout	整型。是否采用标准输出输出随机结果。这对再现仿真很有用。	Off
internal_tcp_packet_size	整型。SPADES 手动收集数据以避免向一个端口写很多而量少的数据。这个参数指定在发送之前必须收集多少字节的数据。任何非正值表示关闭这个收集过程。	1024
agent_db_fn	文件路径。Agent 数据库的目录。	agentdb.list
status_update_intercal	整型。多少时间（秒）更新输出仿真状态。任何非正值表示关闭更新。	5
trace_on_error	布尔型。发生一些错误，像超时和意外错误，输出一些有用的日志信息。强烈建议打开这个功能，因为这些信息对于跟踪 BUG 很有用。	On
create_agent_logfiles	布尔型。是否为每个 agent 创建标准错误和标准输出的日志文件。	On
sec_for_agent_shutdown	Real(>=0)结束信号发出后，多少秒关闭 agent 进程。如果在指定的时间内没有退出，将会被强制地杀掉。	2.0
agent_packet_size	整型。SPADES 手动收集数据以避免向一个 agent 写很多而量少的数据。这个参数指定在发送之前必须收集多少字节的数据。任何非正值表示关闭这个收集过程。	1024
default_agent_input_fd	整型（>=3）。Agent 接收输入时作为管道用的默认文件描述符（agent 类型可以不考虑这个）。这个值必须大于等于 3，因为文件描述	3

	符 0，1 和 2 分别代表标准输出、标准输入和标准错误。	
default_agent_output_fd	整型（>=3）。Agent 输出动作时作为管道用的默认文件描述符（agent 类型可以不考虑这个）。这个值必须大于等于 3，因为文件描述符 0，1 和 2 分别代表标准输出、标准输入和标准错误。	4
load_send_interval	Real(>=0)多少秒将机器的装载值发送给仿真引擎。这是一个共享参数，因为可能使用一个完整的 communication server.	2.0
max_unnatural_lost_agents	整型。如果 agents 不正常退出（也就是，没有一个给出明确的退出或者明确要求退出），communication server 退出。任何负值关闭这个功能。这是整个 communication server 的共享参数。	10
send_agent_think_times	布尔型。是否发送 think time messages 给 agent。	On
send_agent_send_time	布尔型。是否为感觉或者信息发送 send time。Send time 是感觉产生的信息。	On
send_agent_arrive_time	布尔型。是否为感觉或者信息发送 arrive time。Arrive time 是 agent 接收到感觉的时间。	On
default_process_timer	字符串型。当 default timer 指定给 agent 时指定定时器。如果值是 default timer,使用”fixed 0”.	On
record_think_times	布尔型。是否记录 agent 的反应时间到 think_time_file_pattern 参数指定的文件。	Off
think_time_file_pattern	文件路径。指定记录 agent 反应时间的文件。字符串’%A’用 agent 的号码代替，’%D’指 agent 文件名。	%D/agent%A-ttimes.log
replay_think_buffer_size	整型。当使用 replay timer 的时候指定一次读取值的数目。现在没有理由改变这个参数。	64
agent_intercept_library	文件路径。给 agent 指定先装载的库（见 4.7.2 节）。字符串’%ID’指库文件包的目录（默认是/usr/local/lib/spades）	%ID/libspadesint.so
enable_pic_message_reception	布尔型。是否监听 Sys IPC 信息，用来通报 communication server 进程分叉。	On
ipc_forve_remove	布尔型。如果指定，IPC 信息在 communication sever 开始之前移除。在 communication server 异常崩溃或退出的时候这是很有用的	Off
max_secs_for_agent_think	Real.指定在发送序必须和反应一个思考信息之间的最大时间（由 wall clock 测量）。如果值小于 0，wall clock 就不检查。	10
max_simtime_for_agent_think	整型。指定在发送序必须和反应一个思考信息之间的最大时间（由当前 agent 的计时器测	1000

	量）。如果小于 0，就没有最大值。	
agent_check_use_randomness	布尔型。指定是否使用随机确定检查一个 agent（见 agent_check_gumbel_dist_A 和 agent_check_gunbel_dist_B 参数）（见 agent_check_threshold_sec 参数）	On
agent_check_threshold_sec	整型（>=0）如果 agent_check_use_randomness 关闭，一旦 agent 思考时间达到这么长（由 wall clock 测量），完成一个更深层次的检查。	1
agent_check_gumbel_dist_A	Real。如果 agent_check_use_randomness 开，就随机地检查。这个参数和 agent_check_gumbel_dist_B 是 gumbel 区分哪个指定随机检查的可能性。	1.5
agent_check_gumbel_dist_B	Real（>=0）。见参数 agent_check_gumbel_dist_A。	1
agent_stdout_log_fpat	文件路径。这个参数描述了 agent 标准输出的文件路径。字符串'%A'指队员号码，'%D'由 logfile_dir 指定。	%D/agent%A-stdout.log
agent_stderr_log_fpat	文件路径。这个参数描述 agent 的错误输出路径。字符串'%A'指队员号码，'%D'由 logfile_dir 指定。	%D/agent%A-sterr.log

6.12 通讯服务器

这些参数是通讯服务器所理解的（额外的公共参数见 6.1.1 节）。

名称	类型、描述	默认值
engine_host	字符串。运行仿真引擎的机器名称。	localhost
max_connect_reply_wait	整型（>=0）。等待仿真引擎建立连接的正确响应的时间（秒）。	5
wait_sec	整型（>=0）。传送到 select 的系统调用时间（秒）（与 wait_usec 联合使用）。	5
wait_usec	整型（>=0）。传送到 select 的系统调用时间（微秒）（与 wait_sec 联合使用）。	0
no_message_timeout	整型（>=0）。如果通讯服务器没有从仿真引擎接收到信息的时间超过这么多秒，通讯服务器就退出。	20

6.1.3 仿真引擎

名称	类型、描述	默认值
port_bind_retries	整型（>=0）。在这个次数里试图建立端口监听通讯服务器的连接。这特别有用在一排里运行多个仿真，因为 TCP 端口在系统等级里并不是一直立即完全关闭。	3
port_bind_sleep_sec	整型（>=0）。在试图建立端口监听通讯服务器连接之间的睡眠时间（秒）。	2

monitor_port	整型（>=0）。Monitor 连接服务器的端口。	12001
accept_monitor_connections	布尔型。是否监听 monitor_port 来建立 monitor 连接。	On
monitor_intercal	整型（>=0）。发送给 monitor 信息的间隔	33
use_monitor_log	布尔型。控制是否使用 monitor_log_fn 建立一个发送给 monitor 的所有信息的日志。	On
monitor_log_fn	文件路径。将发送给 monitor 的数据写入的文件。字符串'%D'指 logfile_dir。	%D/monitor.log
text_event_log_fn	文件路径。记录每个事件的文本文件（例如，人类可读的）。字符串'%D'指 logfile_dir。见 use_text_event_log。	%D/event_text.log
use_text_event_log	布尔型。是否记录每个事件的文本文件（例如，人类可读的）。这并不推荐，因为一般这样的文件会迅速增大，但这在调试的时候很有用。见 text_event_log_fn	Off
wait_sec	整型（>=0）。发送给 select 系统调用的时间（秒）（与 wait_usec 联合使用）。	3
wait_usec	整型（>=0）。发送给 select 系统调用的时间（微秒）（与 wait_sec 联合使用）。	0
pause_mode_wait_sec	整型（>=0）。像 wait_sec，但用在仿真暂停的时候。	1
pause_mode_wait_usec	整型（>=0）。像 wait_usec，但用在仿真暂停的时候。	0
max_pause_mode_seconds	Real.仿真引擎退出前暂停模式的最大秒数。任何负数关闭这个功能，但这并不推荐，因为关闭这个功能仿真引擎进程会不确定地等待。	90
timeout_for_event	整型。最大等待事件推进时间（秒）。也就是，如果在这段时间里没有事件发生（除暂停模式下），仿真引擎退出。任何负值关闭这个功能，但这并不推荐，因为关闭这个功能仿真引擎进程会不确定地等待。	60
use_migration	布尔型。在仿真过程中是否使用 agent 转移。	Off
min_responses_before_migration	整型（>=0）。考虑转移 agent 一台机器的最小反应数目。	10
max_agentq_trailing_time	整型。由于 agent 的思考时间，有可能会晚收到感觉信息。例如，如果感觉在 x 时刻发出，而 agent 已经是 x+10 时刻，	50

	这就落后了 10 个单位时间。如果 agent 落后的时间单位比这个参数值大，sensation 就转变成 inform（除思考感觉）。负值关闭这个特性。	
max_timenot_trailing_time	整型。像 max_agentq_trailing_time,但不是为 time notify.	50
run_integrated_commserver	布尔型。是否运行通讯服务器和仿真引擎在一个进程里。这可以稍微提高同一台机器上的通讯服务器的性能。	Off
sec_for_socket_shutdown	整型（>=0）。当仿真引擎关闭的时候，等待一些时间直到所有的端口写缓冲器清空。	5
limited_rate_default_st_per_sec	Real(>=0)。在限速模式下，仿真引擎试图维持 wall clock 时间和仿真时间前进的响应。目标是每个秒都运行这么多仿真步数。	1000.0
limited_rate_rebase_interval	Real。在限速模式下，引擎周期性地重设 wall clock 时间和仿真时间前进的响应点。这个参数决定多久（秒）完成这个。这个值小于等于 0 关闭这个特性。	2.0

6.1.4 世界模型例子

这些参数是世界模型例子能理解的（其它公共的和引擎的参数见 6.1.1 节和 6.1.3）。

名称	类型、描述	默认值
size	Vector(Real)[2](>0)。指定世界的 X 和 Y 的大小（米）。	200 200
min_sense_latency	整型（>=0）。见 max_sense_latency。	10
max_sense_latency	整型（>=0）。同 min_sense_latency 一起，指定一个范围给每个感觉的反应时间。	20
min_sense_interval	整型（>=0）。见 max_sense_interval。	50
max_sense_interval	整型（>=0）。与 min_sense_interval,指定一个范围给每个感觉的仿真时间。	70
min_action_latency	整型（>=0）。见 max_action_latency。	90
max_action_latency	整型（>=0）。与 min_action_latency 一起，指定一个范围给每个动作的反应时间。	100
num_agents	整型（>=0）。Agents 开始的号码。	1
sim_time_per_second	Real(>=0)。每秒运行多少仿真步。这用来说明其它参数。	100
stiction	Real(>=0)。这个参数影响球的物理运动。指定一个最大的加速度（仿真场地的最大摩擦力）。单位是 m^2/s。	3
speed_max	Real(>=0)。指定球的最大速度（m/s）。	3
accel_rest	Real(>=0)。指定球静止的时候最大加速度（m^2/s）。加速度随着速度的增加衰	5

	减。	
simulation_length	整型。仿真运行的仿真步数。	10000
num_comm_servers_wanted	整型。取消仿真暂停之前等待通讯服务器的个数。	1
agent_speed	字符串。这个传送给 agents 来指定 agents 的衰减。有效值是'fast','medium','slow'和'soccer'。最后的速度通过 SPADES 和 SoccerServer 的实验（http://sserver.sf.net）。这个值在 medium 和 slow 之间。	Fast
agent_types	Vector(String)[1-∞]。这个字符串直待牛关agent开始的类型名称。他们按顺序，重复使用只要能得到num_agents。	type0 type1
random_act_latency	布尔型。是在动作反应时间中否使用随机值。使用随机值的重要信息见4.11节。	True
random_agent_placement	布尔型。初始化agent位置的时候是否使用随机值。使用随机值的重要信息见4.11节。	True
use_random_untimed_sensations	布尔型。如果这个功能打开，一些感觉被标为没有时间的感觉。这只是测试没有时间感觉机制的时候使用。	False

6.2 智能体数据库

智能体数据库描述智能体的类型（见 4.5 节讨论）。

数据库是按照 XML 格式读入的（参数 agent_db_fn 指定了文件的路径）。SPADES 发行版和安装版里提供一个简要的 agentdb.xsd。这一节详细描述格式的细节。

所有的元素必须在 namespace 中 http://spades-sim.sourceforge.net/agentdbxml.html。在 sample_agent 数据库指定本地名称 adb。

顶层的元素必须是 agentdb。它必须包含版本号 version。这个特性内容必须是指定 SPADES 版本的一个十进制数。这是希望考虑到往后兼容性。

所有的属性名称只能包含数字、文字和_，不能有空格。

下面的子元素可以以任何顺序出现任何次数。

include 没有属性，没有子元素，文本内容。指定一个智能体数据库的完整路径（存在的）。在执行这个文件的其它内容之间读入那个文件。

agent_type_external 表示智能体作为一个外部进程开始。这个元素需要 name 属性来指定类型名称。

子元素：

inputfd （可选的）指定文件描述符，agent 用来从通讯服务器中读取信息。如果这个省略或者是-1，就使用默认的 default_agent_input_fd。

outputfd （可选的）指定文件描述符，agent 用来向通讯服务器中发送信息。如果这个省略或者是-1，就使用默认的 default_agent_output_fd。

timer （可选的）同 4.7.1 节中讨论的指定一个计时器类型描述符。如果省略，就使用默认值 default。

workingdir （可选的）指定 agent 的工作目录。在运行可执行码之前，当前目录转到这

个目录。所有的相对目录都是从 agent datebase 位置开始的。

commandline （必须的）为 agent 指定一个完整的命令行。注意这个子元素的格式和分析规则在以后的版本可能要改变。分析规则如下：

- 值必须在括号（‘{}’）中。
- 空格将命令行中各个参数隔开。
- 如果你的参数中需要一个空格，在参数两端使用双引号。双引号将被移除掉。
- 如果你的参数中需要双引号，在每个的旁边使用双引号。它们将被减少到一个双引号。
- 没有其它的字符是特殊的。注意一个细节，这就意味着反斜杠（’\’）没有指定特殊意义，shell 不能重定向（使用’>’和’<’）。

这里给出了一些命令行分析的例子。第一，给出命令行，然后参数指定给 agent，用逗号隔开，用单引号括起来。

命令行	分析后和隔开的参数
`foo -a b --code \\`	`’foo’, ’-a’, ’b’, ’--code’, ’\\’`
`bar -a "Some Name"`	`’bar’, ’-a’, ’Some Name’`
`baz -a "Some""Name"`	`’bar’, ’-a’, ’Some"Name’`
`bif -a """SomeName"""`	`’bif’, ’-a’, ’"Some Name"’`

agent_type_integrated 描述 agent 可以装载的动态库和存在同一进程中通讯服务器。这个元素需要一个属性 name，指定类型名称。

子元素有：

lin_path （必须的）定义 agent 时，指定连接动态库的（可能相对的）路径。

init_arg （可选的）如果指定，传送给 agent 的 IntegratedAgent 类中 initialize 函数。

timer（可选的）同 4.7.1 描述的指定计时器类型的描述字符串。如果忽略，就使用 default 值。注意整个 agent 不能接受有的计时器类型（见 4.7.1）。

require_integrated_commserver （可选的）布尔型，指定 agent 是否需要在一个通讯服务器上运行。默认值是 false。

agent_type_placeholder 这个元素需要名称指定类型的名称。没有子元素和内容。这种类型的 agent 不能实际启动。这在指定仿真引擎（不需要实际启动 agent）很有用，并指定通讯服务器的完整类型描述。

6.3 前缀长度的 I/O 格式

为了避免任意信息长度和格式的限制，SPADES 使用前缀长度的 I/O 格式。首先发送信息的长度前缀 I/O 格式。注意 SPADES 可能发送增量式的数据，所以整个信息不能立即可用。因此，读取器必须有一些读取缓冲。类 ReadBuffFD 提供这些功能，并且对 agent 来说，将来要合并到 SPADES 的 agent 库中。

每条信息，发送四个字节（按网络字节顺序）代表信息的长度（不包含表示长度的四个字节）。然后再发送数据本身。

6.4 外部的 agent 输入/输出

这一部分给出 agent 和外部的 agent 输入/输出的详细信息。这里的输入指 agent 从通讯服务器接收的信息，输出指 agent 发送给通讯服务器的信息。

首先，所有的通讯是通过管道的。当 agent 启动的时候，与通讯服务器的输入输出必须使用文件描述指定 agent 的类型（或者默认值 default_agent_input_fd，

default_agent_output_fd）。

所有通讯服务器和 agent 通讯使用前缀长度 I/O 格式（见 6.3 节）。

6.4.1 agent 输入格式

这部分描述 agent 从通讯服务器接收信息的格式。信息的第一个字节信息的类型，剩下的字节是信息的参数和数据。第一个字节后面没有空格。

通过这一章节，下面是可用的：

- time 指仿真时间（整型）。这之后一般跟着一个空格。
- data 指任意字符串数据。注意因为使用前缀长度格式（见 6.3 节），数据的格式和长度没有限制。实际上可以发送任意的二进制数据。
- token 是没有空格的字符串文本。

发送给 agent 的信息有：

- “Stime time data’
  这是发送给 agent 的感觉信息。开始一个思考循环。
  第一个时间是感觉信息产生的时间（也是发送的时间），第二个时间是感觉到达 agent 的时间（也就是到达时间）。如果参数 send_agent_send_time 是 off，发送时间就是－1。如果 send_agent_arrive_time 是 off，到达时间就是－1。data 是世界模型产生的任意数据流。Agent 可以返回动作信息 act message，必须以 done thinking message 结束。

- ‘Itime time data’
  这是一个通知信息 inform message。意义和感觉信息一样，但不开始一个思考循环。Agent 不能回复信息。
  注意通知由世界模型明确地产生，或者“旧”的感觉信息可以转化为通知。见 4.6 节详细信息。

- ‘Ttime’
  这是时间通报 time notify，空的感觉信息。不开始一个思考循环。时间是要求时间通报的时间，不管时间通报发送的时刻。Agent 可以回复动作信息 act message，并且必须以完成思考信息 done thinking message 结束。

- ‘Otime’
  这是不能开始思考循环的时间通报。Agent 不能回复信息。

- ‘X’
  这是告诉 agent 退出的信息。将没有更多的信息发送给 agent，也不需要回复。注意关闭进程在 5.5 节讨论。

- ‘Ddata’
  启动之后，这个初始化信息发送给 agent。正常启动情况下，data 是空的。移动 agent 的情况下（见 5.3 节），数据是旧 agent 进程发送给新 agent 进程的数据。初始化完成信息 initialization done 必须在一旦初始化数据完成和启动完成之后发送。

- ‘M’

这是告诉 agent 将要被移动的信息。Agent 必须回复一个移动数据信息。关于 agent 移动的信息见 5.3 节。

- ‘Ktime’
  这是通报 agent 上个思考周期用了多长时间。这仅仅在 send_agent_think_times 是 on 的情况下。

- ‘Etoken’
  这是发送给 agent 的错误信息。token 指定错误的类型。
  no_token_on_line　agent 发送空信息给通讯服务器。
  act_when_not_thinking　不是思考周期的时候 agent 发送动作信息。
  rtn_when_not_thinking 不是思考周期的时候 agent 发送时间请求信息。
  bad_time_in_rtn 时间请求的格式不对。
  done_thinking_when_not_thinking 不是思考周期的时候 agent 发送思考完成信息。
  mig_data_when_not_mig agent 发送移动数据信息，但通讯服务器没有发送移动请求。
  bad_token 信息的第一个字符不能指定有效的信息。
  init_done_when_not_init 不是初始化模式下 agent 发送初始化完成信息。

6.4.2 agent 输出格式

这个章节介绍 agent 发送给通讯服务器的信息格式。

通过这以章节，下面是可用的：

- time 指仿真时间（整型）。必须跟着一个空格。
- data 之任意数据流。注意因为使用前缀长度格式（见 6.3 节），数据的长度和格式没有限制。实际上，可以发送任意二进制数据。
  格式首先是基于文本的。第一个字节指出信息的类型，剩余的字节是信息的参数和数据。第一个字节之后不能有空格。
- ‘Adata’
  agent 发送的动作。Data 是任意格式，取决于世界模型。注意因为使用前缀长度格式（见 6.3 节），数据的长度和格式没有限制。
- ‘Rtime’
  只是一个时间通报请求。time 是要求通报的时刻。见 4.6 节细节。
- ‘D’
  完成思考信息用来结束每个思考周期。所有的动作和时间通报请求都必须在这个信息之前发送。
- ‘Mdata’
  这是 agent 移动的数据。data 是发送给启动的 agent 的初始化信息。见 5.3 节。
- ‘X’
  这是退出信息。Agent 向通讯服务器通报退出。这个信息不管 agent 是否在思考都可以发送。如果 agent 在思考周期，退出信息功能和完成思考信息一样。将没有更多的信息从这个 agent 发送或得到。
- ‘I’
  这是 agent 在启动之后必须发送的初始化完成信息。

6.5 完整的 agent 输入/输出

这节描述创建完整的 agent 的过程。

一个完整的 agent 是 SPADES 装载的一个动态库。你可以像创建其它动态库一样创建库，所以过程请参考你的系统文档。注意 SPADES 使用 libltdl[http://www.gnu.org/software/libtool/libtool.html]，并且必须使用 libtool 很好地结合 SPADES。

你的库必须提供两个东西。第一，IntegratedAgent（后面将描述其接口）的子类和创建 agent 的入口指针。

IntegratedAgent 是一个虚基类。函数一般对应于扩展的 agent 能接收的信息，所以请参见 6.4.1 节详细信息。

函数有：

initialize agent 启动的时候调用这个函数。附带一个字符串是初始化的参数，指定 agent 的类型（列在 agetnt database 中）。

receiveTimeNotify agent 接收到时间通报。ThinkingType 参数指定是不是正在思考（也就是能不能发送动作）。

receiveSense agent 接收感觉信息。ThinkingType 参数决定是感觉信息 sense 还是通知信息 inform。

receiveMirationRequest 要求 agent 开始移动进程。

receiveExit 发送给 agent 退出信息。在这条信息之后通讯服务器将删除这部分内存。

receiveError 通知 agent 发生错误。

receiveThinkTime agent 接收 think time message 信息。

getAction 指定对象的类型 IntegrateAgentActions，用来和通讯服务器通讯。

setActionCallbacks 你不需要调用这个函数。SPADES 用这个函数来设定 IntegrateAgentActions 调回对象（见下面）。

类 IntegrateAgentActions 是 agent 用来和通讯服务器通讯用的。IntegratedAgent 的函数 getAction 给 agent 一个 IntegrateAgentActions 对象来调用函数。扩展的 agent 可以发送相应的信息，见 6.4.2 节的详细信息。

IntegrateAgentActions 的函数有：

act 与数据结合的动作信息。

requestTimeNotify 这是一个时间通报申请 request time notify 信息。参数 time 是要求通报的时间。

doneThinking 结束每个思考周期的完成思考信息 done thinking。

exit agent 要求退出的时候调用。

initDone 这是初始化完成信息 initialization done，用来响应 agent 初始化信息。

migrationData 这是转移数据信息。用来响应移动请求信息。

创建 agent 的入口指针必须有一个函数是下面的名称和署名：

spades::IntegratedAgent*createAgent（void）；

spades::IntegratedAgent*libname_LTX_createAgent（void）；

libname 是你没有扩展名的库的名称。例如，如果你创建一个库叫 myagent.so，libname 就是 myagent。在其它情况下，这个函数返回一个动态分配对象，是 IntegratedAgent 的子类型。

注意如果你用 C++编译器编译，你必须有下面的声明

```
extern “C”
{
    your declaration here
}
```

避免搞乱函数名。

6.6 监视器 monitor 接口

4.8 节提供了监视器函数一般函数。这一节详细介绍接口。

Monitor 接收信息的格式完全取决于 world model；SPADES 不增加任何东西。

为了从 monitor 接收数据，SPADES 使用前缀长度数据格式（见 6.3 节）。第一个字符决定信息的类型。

P 仿真暂停。（也就是模式切换到 SM_PausedMonitor）。

R 仿真进入运行模式（也就是模式切换到 SM_RunNormal）。

L 仿真进入限速模式（也就是模式切换到 SM_RunLinitedRate）。见 5.6 节。

D 通知引擎 monitor 断开连接。将没有更多的信息发送给 monitor。

X 代表结束仿真。

W 指这个数据必须通过世界模型处理。首字母’W’移除，剩下的数据发送给 WorldModel 的 parseMoniotorMessage 函数，

如果发送了无效的首字母，将输出错误，但这在其它方面是忽略的。

6.7 agent 思考时间

为了决定 agent 进程的计算量，SPADES 依赖 Linux 内核提供的工具。总的来说，有两步过程。

1. 询问内核使用了多少计算量。
2. 将计算时间转换成仿真时间并申请动作。

6.7.1 跟踪瞬间计时器

瞬间计时器 jiffies timer 使用标准 Linux 内核工具。

在文件’/proc/pid/stat’中，Linux 内核提供“瞬间”进程使用 CPU 的次数。一个瞬间是相当粗糙的时间单位，一般是 10ms。内核实际上报告四种不通的瞬间次数：

utime 用户模式使用的瞬间次数。

stime 系统模式使用的瞬间次数。

cutime 用户模式下进程和子进程使用的瞬间次数。
cstime 系统模式下进程和子进程使用的瞬间次数。

utime 和 stime 不同之处在于加在一起得到 agent 思考的总瞬间次数。cutime 和 cstime 更新后我不能决定详细情况，所以 SPADES 现在没有使用它们。

一旦瞬间的次数确定，将转换成仿真时间。将进行下面的计算：

- 令 J 等于瞬间的次数。
- 令 B 等于及其的主频，从'/proc/cpuinfo'中读取。
- 令 K 等于瞬间计时器提供的参数值（见 4.7.1 节），代表多少千条指令转换成 1 个仿真步。

仿真步的数目就是（转换成最近的整数）

$$\frac{10JB}{K} \qquad (6.1)$$

如你所见，这个线性转换的参数是用户可控的参数。然而，不同的机器使用相同的仿真，需要一些基准来决定 CPU 时间和仿真时间参数的等价设置。

6.7.2 跟踪 Perfctr 计时器

Linux 运行次数驱动器[http://sourceforge.net/projects/perfctr]允许 SPADES 跟踪 agent 进程使用指令的次数。这允许更精确的跟踪思考时间和再现结果。

然而，现在，这需要修补后的内核。SPADES 当前工作在 2.5.1 版本上。2.6 系列改变了用户等级，SPADES 现在不能工作在上面。

当然，SPADES 使用 ptrace 机制正确地跟踪进程。使用 ptrace 会有 agent 的表现会有微妙的效果（详见 ptrace 文档）。

这个计时器有一个指示每个周期的多少千条指令的参数。

6.7.3 记录的文件格式

像 4.7.1 节描述的，SPADES 提供 agent 思考时间的记录的能力。这一节描述记录的文件格式。

文件是二进制格式来存储空格和分析时间。所有的数字安网络字节顺序存储。

文件的前 4 个字节说明每个思考周期记录的字节数。当前，SPADES 之支持每个思考周期 2 个字节的记录，所以这主要是为了向上兼容。这之后，每两个字节表示 agent 的一个思考反应时间。

注意 SPADES 安装了名为 show_ttimes.pl 的 perl 脚本以人类可读的格式显示思考时间文件的内容。详见 4.7.1 节。

6.8 算法

在技术文章中[Riley,2003,Riley and Riley,2003]也描述了时间顺序和实现算法。这里大部分从那些文章中拷贝过来的。

这一节描述 SPADES 仿真算法。算法是离散事件仿真的基本算法的修改。

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    while (no event will be received for time less than next_event.time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        realize (next_event)
        next_event = pending_event_queue.head
```

表 6.1 基本离散事件仿真的内部循环

表 6.1 是基本离散事件仿真的内部循环。一个主要的区别是调用 advanceWorldTime。这支持连续的仿真和世界模型。这个函数改进了离散事件仿真世界的时间。实现了事件改变世界，并能显著地引起其它事件，并排到队列 pending_events_queue 中。

为了保证所有的事件都是偶然的顺序执行的，仿真环境必须决定是否接收所带时间标签小于下个不定事件的未来事件。并行仿真器的time-managenment功能已经充分研究，有几种使用的途径[Chandy and Misra, 1981, Bryant,1977, Mattern, 1993, Chandy and Misra, 1979, Chandy and Sherman, 1989, Lubachevsky, 1989,Steinmann, 1991, Nicol, 1993, Riley et al., 2000]。这些方法太复杂是因为分布式仿真要管理分布式环境中每个进程私有事件列表，并且这个列表独立于其它进程（时间管理算法的限制）。为了简单实现，我们选择另一种著名的centralized event list方法。这种方法，master进程管理单个复合的事件列表，master进程负责为其它所有进程安排事件和管理事件。任何进程安排事件必须通报master进程（中心事件列表的管理者）来安排事件。这种方法非常简单和容易实现。Master 进程知道所有时刻将要发生的事件，并且立即决定将要发生的事件。中心事件列表方法的缺点是，每个进程必须通知中心时刻表，中心时刻表必须完成先前的事件再准备完成更多的事件。设计 agent 使用 sense-think-act 范例可以减轻这个缺点，因为每个 agent 对一个感觉事件返回一个动作，来通知时刻表进程完成。这种方法的一个明显缺点是效率和可测量，因为单个进程统一了所有 agent 的活动。单个的统一点会成为瓶颈并降低整个仿真速度。为了达到我们的目的，agent 的总数目很小，我们就不认为中心时刻表是主要的瓶颈。

众所周知，任何保守的并行离散仿真系统需要一个非零 lookahead 特性，来达到好的并行效果[Ferscha and Tripathi,1996]。简单情况下，lookahead 值低于任何进程 A 产生时间和其它进程 B 实现这个事件之间的仿真时间。大的 lookahead 值有更好的表现。我们正在讨论 SAPDES 中 lookahead 值的算法。我们的第一个版本将包含一些基本的想法，然后在详细描述 SPADES 算法。

首先必须给出 agent 正常 think-sense-act 周期引起的事件。这个周期已在 6.2 节介绍了。第一步，一个事件放入队列产生感觉。代表性地，事件的实现就是读取世界中的状态并将一些转换后的信息发送给 agent。这个信息被压缩到一个感觉事件并压入事件队列。SPADES 需要在创建感觉事件和感觉事件之间有一个最小的感觉延时，这是由世界模型指定的。当感觉事件实现了，这个信息将发送给 agent 开始思考进程。注意感觉事件的实现不需要读取当前世界模型的状态，因为在创建感觉事件的时候已经调整了信息。在收到感觉之后，通讯服务器开始为 agent 的计算计时。当通讯服务器接收到所有 agent 的动作，agent 产生这些动作的计算时间转换成仿真时间。所有的动作和思考延时将发送给仿真引擎（见 6.2 节发送动作）。

接收到之后，仿真引擎加入动作延时（由询问世界模型决定）并将动作放入未做动作队列。与最小感觉延时一样，SPADES 在动作发送时间和动作事件时间之间需要一个最小动作延时。动作时间的实现实际引起 agent 的动作影响世界。

注意单个 agent 在一个进程里一次可以有多个 sense-think-act 循环，见 6.1 节说明。例如，一旦 agent 发送动作（发送动作见 6.2 节），它可以接收下一个感觉尽管动作还没有实际影响世界（动作事件见 6.2 节）。SPADES 唯一禁止的重叠是两个思考阶段的重叠。

注意所有的动作在离散的事件发生效果。因此 SPADES 没有外在的支持并行动作交互的模型支持。

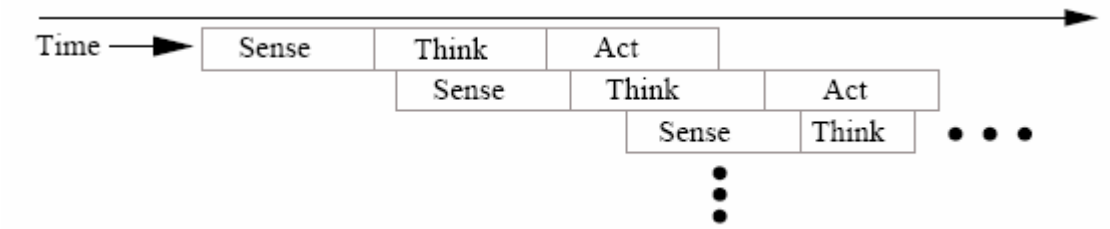

图 6.1 sense-think-act 循环重叠说明

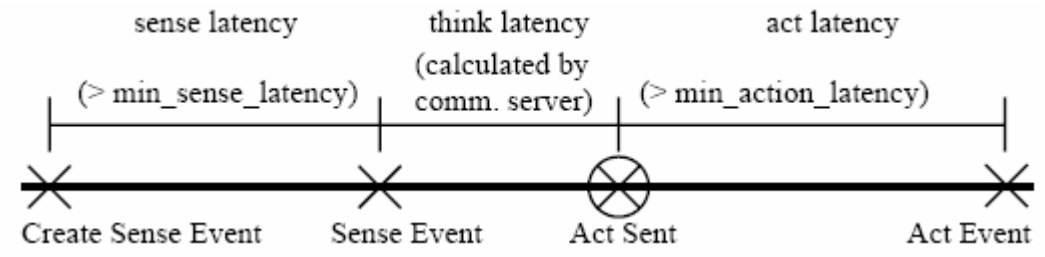

图 6.2 agentsense-think-act 循环。“Act Sent”用圆标出因为其它标志代表队列的事件，“Act Sent”只是通讯服务器发给引擎的一个信息而不是事件队列中的事件。

例如，两个仿真机器人开始向前移动。世界模型的任务是确认这些动作的交互（例如碰撞）和适当的反应。同样地，agent 之间的通讯作为另一个动作管理。世界模型提供按设计要求的通讯限制。

感觉和动作延时提供一个 lookahead 值，agent 就可以并行思考。当 agent1 实现了感觉事件，它不能在最小动作延时内产生任何事件。因此安全（至少只考虑 agent1 的时候）实现所有事件而不干扰事件顺序。

我们称“最小 agent 时间”的数量决定了所有 agent 的最大安全时间。最小 agent 时间是 agent 能产生影响其它 agent 或者世界的事件的最早时间。这同在仿真使用的时间标签底限（LBTS）概念相似。计算最小 agent 时间见表 6.2。agent 的状态要么是“思考”，也就意味着感觉发送给了 agent 并且没有接收到回复；要么是“等待”，也就是 agent 等待仿真引擎。除了初始化，agent 状态一般都是思考或等待。Agent 的当前时间是与 agent 的最后通讯时间（发送感觉或者收到动作）。从通讯服务器收到信息不能引起最小 agent 时间的减少。

```
calculateMinAgentTime()
    ∀i ∈ set_of_all_agents
        if (agent_i.status = Waiting) agent_time_i = ∞
        else agent_time_i = agent_i.currenttime + min_action_latency
    return min_i agent_time_i
```

表 6.2 决定 agent 影响仿真的最小时间代码

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    min_agent_time = calculateMinAgentTime()
    while (next_event.time < min_agent_time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        realizeEvent (next_event)
        next_event = pending_event_queue.head
        min_agent_time = calculateMinAgentTime()
```

表 6.3 并行 agent 分布事件仿真的严格时间标签限制代码

然而，实现一个时间可能会引起增加或减少。因此，最小 agent 时间必须在每个事件实现之后计算。然而，这个算法必须调整增加，这样整个 agent 就不用每次都扫描了。

基于最小 agent 时间的计算，我们现在可以描述一个简单的并行 agent 离散事件仿真器，见表 6.3。值 min_agent_time 用来决定是否在队列中下个事件之前出现更多的事件。

这个算法修正了结果（所有的事件按照时间标签顺序实现）并且实现并行，但并没有达到最大可能的并行数。图 6.3 说明了两个 agent 的例子。当 agent1 实现感觉事件，最小 agent 时间到 A。这允许产生 agent2 的感觉事件并询问。但是，直到收到 agent 的反应之前 agent2 的感觉事件不发生。然而，如上面讨论的，感觉事件实现的效果并不依赖于世界的当前状态。如果 agent2 当前正在等待，没有理由同时实现感觉和允许其它 agent 思考。

但是，允许按顺序实现事件，agent1 发送事件的时间可以早于 agent2 感觉事件时间。各种按顺序实现方法都是可以接受的（见例子说明）。实际上，我们需要检验事件顺序有没有偶然性。关键是 agent 收到的感觉与其它 agent 收到的感觉是偶然独立的。为了正确的保证，我们定义事件的一个子类“fixed agent events”，它有一下特性：

1. 它们不依赖于当前世界的状态。
2. 它们只通过发送给 agent 信息影响单个 agent。
3. 感觉事件和时间通报时间都是确定的 agent 事件。
4. 确定的 agent 事件是唯一引起 agent 开始思考周期，但它们并不需要开始思考周期。

SPADES 提供下列正确保证：

1. 所有的不确定 agent 事件都按时间顺序实现。
2. 所有发送给 agent 的感觉事件都是确定的 agent 事件。

3. 特定的 agent 确定事件按时间顺序实现。

为了完成这个，要包含一些新的概念。第一个注意点是“最小感觉时间”。这是一个新的感觉（也就是确定的事件）除了时间通报产生并压入队列的最早时间。SPADES 当前实现需要世界模型提供创建感觉事件和发送感觉事件之间的最小时间（见图 6.2），所以最小感觉时间就是当前仿真时间加上那个时间。

时间通报是有特权的事件。它们有特殊操作，因为它们之影响要求时间通报的 agent 。 SPADES 同样允许在未来任意短的时间里要求时间通报。

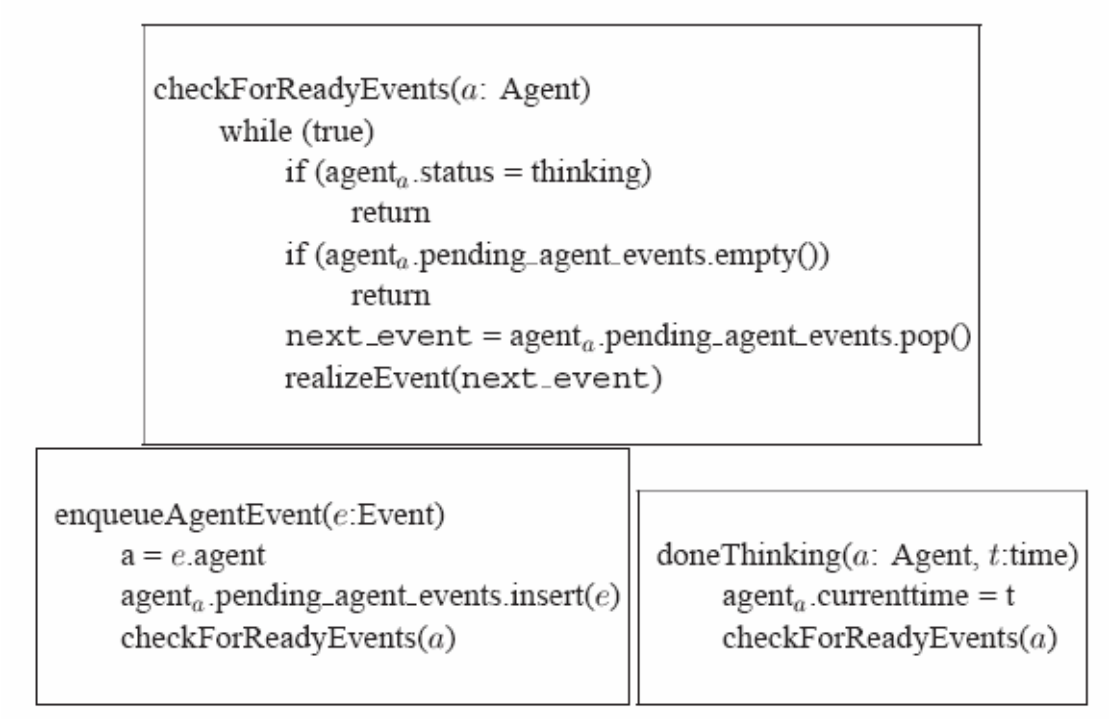

图 6.4：维持由 agent 确定的 agent 事件队列代码

这就意味着当 agent 思考的时候，仿真引擎不能发送任何更多的确定事件给 agent，这样 agent 就不可能违反正常情况 3。但是，如果 agent 在等待（也就是没有思考），队列中第一个确定的 agent 事件可以在最小感觉事件之前发送。

为确保可能事件的顺序，必须管理每个 agent 的确定 agent 事件队列。第一，enqueueAgentEvent 将一个确定 agent 事件压入队列。Agent 完成思考周期时调用 doneThinking 函数。这两个函数都调用一个第三方函数 checkForReadyEvents。这些函数的伪码见表 6.4。注意在 checkForReadyEvents 中，事件的实现可以引起 agent 的状态从等待转到思考。

使用这些函数，我们在表 6.5 中描述了 SPADES 主循环。这是由表 6.3 的算法修改的。两个关键的改变是：在第一个循环中，确定的 agent 事件不实现，改在压入 agent 队列的时候实现。第二个循环中（“每个”循环）扫描事件队列的头并在小于感觉时间的时间内将所有的确定 agent 时间移到 agent 队列中。注意在这两个方面，把事件移到 agent 队列中引起事件的实现（见表 6.4）。

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    min_agent_time = calculateMinAgentTime()
    while (next_event.time < min_agent_time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        if (next_event is a fixed agent event)
            enqueueAgentEvent(next_event)
        else
            realizeEvent (next_event)
        next_event = pending_event_queue.head
        min_agent_time = calculateMinAgentTime()
    min_sense_time = current_time + min_sense_latency
    foreach e (pending_event_queue) /* in time order */
        if (e.time > min_sense_time)
            break
        if (e is a fixed agent event)
            pending_event_queue.remove(e)
            enqueueAgentEvent(e)
```

表 6.5：SPADES 使用的高效的并行 agent 离散事件仿真器代码