SDT：一个面向场景规约的运行时测试工具*)

雷 斌 王林章 李宣东 郑国梁

（南京大学计算机软件新技术国家重点实验室 南京 210093）

摘 要 利用设计模型信息，提高测试自动化程度是测试领域的重要课题。UML 顺序图是广泛使用的场景规约语言。本文研究了面向场景规约的运行时测试方法，并应用该方法实现了一个基于 UML 顺序图场景规约的测试工具 SDT；它从 Ration Rose 的规约文件中提取顺序图信息，生成表示预期行为属性的事件有向无环图，对代码进行插装，并利用随机测试用例执行代码，最后将反向工程得到的运行时轨迹与有向无环图进行比较，对实现和设计的一致性进行自动化验证。

关键词 UML 顺序图，测试场景，有向无环图，运行时轨迹

SDT: A Scenario Specification-Oriented Runtime Testing Tool

LEI Bin WANG Lin-Zhang LI Xuan-Dong ZHENG Guo-Liang

(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093)

Abstract Utilization of design model to improve testing automation is an important subject in software testing research. UML Sequence Diagram is widely used for specifying software scenario. In this paper, we develope a scenario specification-oriented runtime testing method, and built a prototype tool SDT. It can parse the sequence diagram information from Rational Rose specification file and generate directed acyclic graph (DAG) which stands for expected behavior property, and interpolate java code implementation of the design model, then use random test cases to run the code, and generate runtime execution trace (RET) from trace file, at last compare DAG with RET to verify the conformance between design and implementation.

Keywords UML sequence diagram, Test scenario, Directed acyclic graph, Runtime execution trace

1 引言

软件测试通过设计和执行用例、分析结果等活动来获取软件系统的信息，发现错误，验证系统是否符合需求，并为软件生命周期的其它环节提供信息。随着软件复杂度的提高，以人工为主的测试耗费越来越大，且容易引入错误。因此，迫切需要利用已有的软件设计信息，提高测试的自动化程度。

UML[1]是一种定义良好、功能强大的可视化建模语言，对面向对象软件开发全生存期提供支持，在学术界和工业界都得到了广泛应用。而如何将基于 UML 的需求和设计规约，用于软件测试领域，提高效率，减轻测试的工作量，成为面向对象软件测试领域新的挑战。

UML 顺序图用于对系统单元之间的交互行为进行建模，强调消息的时间顺序[2]。对系统的一个有限片断来说，顺序图提供了实现的抽象视图，它通常是过多或过少细节的良好折中，可用于在不同抽象级别和粒度建立软件系统的模型[3,4]。从测试的角度来看，它是代码设计的依据，也是生成测试用例的信息来源和集成测试的依据。利用成为基线的顺序图生成测试用例，在代码成为基线后便可以开始测试工作，便于合理组织测试资源，且对顺序图进行分析时也能发现其本身的缺陷，及时排除，以防缺陷随着开发过程的进展被放大[5]。另外，正在逐渐被中小型项目广泛采纳的敏捷软件开发(Agile Software Development)也倾向于在项目早期建立测试规约[3]，因此我们希望能够研究出顺序图驱动的测试方法，实现部分自动化而不增加用户额外的工作量，且不需要用户掌握形式化方法领域的专门知识，这样的测试方法容易被已广泛使用 UML 的工业界采用[4,5]。

本文研究了面向场景规约的运行时测试方法，并应用该方法实现了一个基于 UML 顺序图场景规约的测试工具 SDT。文章组织如下：第 2 部分介绍 UML 顺序图的语法和语义；第 3 部分介绍顺序图驱动的运行时测试方法；第 4 部分介绍该方法的支撑工具 SDT 的实现和运行情况；最后是相关工作和总结。

2 UML 顺序图简介

作为 UML 的 2 种交互图之一，顺序图将某一用例的特定的场景中的交互，按时间顺序组织为二维图。纵向是时间轴，时间沿竖线向下延伸。横向轴代表在协作中各独立对象的类元角色[7]。通常只关注时间的前后关系，而不关注时间的长短。对象生命线表示一个对象起特定作用的期限，生命线间的箭头表示对象间的消息通信。每个消息旁标注消息名和一些控制信息。顺序图上的模型元素所表示的都是系统的

*)本文的研究工作受到国家自然科学基金(批准号 60203009，60233020)，江苏省自然科学基金(批准号 BK2003408)和国家 973 项目(批准号 2002CB312001)的资助。雷 斌 硕士生，主要研究软件工程、软件测试。王林章 博士生，主要研究软件工程、软件测试、模型驱动的测试。袁洁松 硕士生，主要研究软件工程、软件测试。李宣东 博士，教授，博士生导师，主要研究软件工程、形式化方法、模型检验。郑国梁 教授，博士生导师，主要研究软件工程、软件开发环境。

本质信息，需要在从设计到实现的过程中保持，因而软件实现中行为的本质方面都会在顺序图设计模型中表示。可以从设计模型中获取行为信息，用于确认软件实现是否与设计一致[2,7]。

本部分介绍一个顺序图，描述客户通过 ATM 验证存款额并存款的用例片断[5]，如图 1。为了从顺序图中自动获取行为信息，本文给出顺序图的形式化定义。

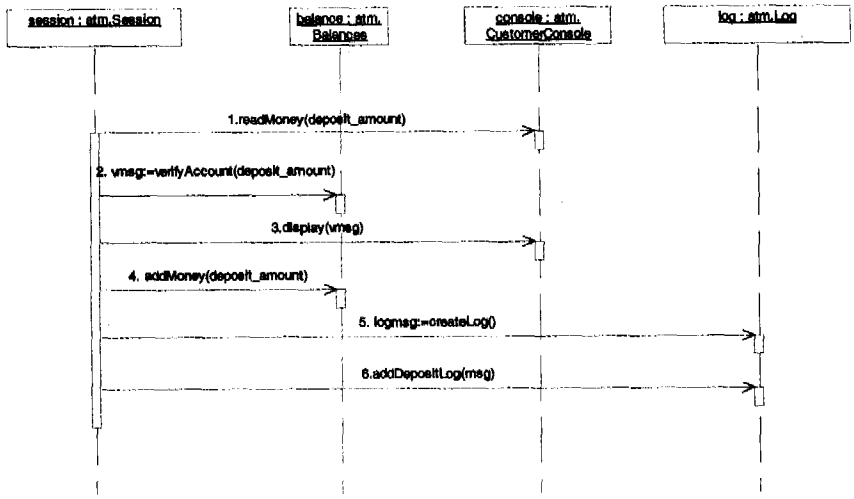

图 1　ATM 存钱场景的顺序图

2.1　顺序图的形式化定义

定义 1（顺序图，UML Sequence Diagram）　UML 中的顺序图 SD 可以表示为一个六元组：$SD=\langle O, M, E, \rightarrow, fem, feo\rangle$。其中：

• $O=\{O_1, O_2, \cdots, O_m\}$ 是有穷的对象集合。$O_1, O_2, \cdots, O_m$ 都是顺序图中的对象；

• $M=\{m_1, m_2, m_3, \cdots, m_n\}$ 是顺序图上描述的有穷消息集合；

• $E=\{e_1, e_2, e_3, \cdots, e_n\}$ 是 O 中所有对象可能产生的事件集合，包括发送消息和接收消息；

• $\rightarrow$ 是一个表示 M 和 E 中元素之间的时间先后的偏序关系；

• fem 是从 M 到 E 的一个函数关系，$\forall m_j \in M, \exists e_{i1}, e_{i2} \in E$ 使得 $e_{i1}=fem(m_j, s)$ 且 $e_{i2}=fem(m_j, r)$ 分别表示消息 m_j 所对应的发送消息事件和接受消息事件；

• feo 是从 E 到 O 的一个函数关系，$feo(e) \in O$ 表示事件 e 所对应的对象。对象 O_i 上所有的事件集合用 E_i 来表示，$E_i=\{e \mid \forall e \in E \wedge feo(e)=O_i\}$。

本文目的是验证顺序图表示的系统预期行为与软件在运行时行为的一致性。由于顺序图描述事件顺序时的语义不精确性，使得可能从图上观察到错误的作者并未定义的事件先后关系，因此很难直接从顺序图上得到合理的事件序列。例如，图 1 中同一发送者 session 发送给 console 和 balance 的消息，可能因为延迟或者通信内部实现原因，消息 4 先于消息 3 被接收到。为了明确上述顺序图定义中的事件顺序，给出定义如下：

定义 2　顺序图中的两个事件满足 $e_1, e_2 \in E, e_1 \rightarrow e_2$ 当且仅当满足下面三种情形之一：

• $\exists m_i \in M$，满足 $e_1=fem(m_j, s)$ 且 $e_2=fem(m_j, r)$；

• $\exists m_i, m_j \in M$，满足 $m_i \rightarrow m_j$，$e_1=fem(m_i, s)$ 或 $e_1=fem(m_i, r)$，且 $e_2=fem(m_j, s)$；

• $\exists m_i, m_j \in M$，满足 $e_1=fem(m_i, r)$，$e_2=fem(m_j, r)$，$m_i \rightarrow m_j$，$fel(e_1)=feo(e_2)$ 且 $feo(fem(m_i, s))=feo(fem(m_j, s))$。

因此，可以确定顺序图事件之间可能存在的时间先后关系的集合。对于任意两个事件 e_i 和 e_j，如果 $e_i \rightarrow e_j$，则可以从 e_i 画一条有向边到 e_j，当 E 中所有事件都在这个有向图上出现后，就完成了顺序图事件的有向无环图（DAG，Directed Acyclic Graph）。顺序图事件间的先后顺序确定后，就可以确定特定场景下刻画系统行为的事件序列 。而所有满足 DAG 中先后关系的事件序列，对应顺序图上的一条执行路径，表示系统的一个场景。为了保持目标系统行为的一致性，其运行轨迹也应该满足 DAG。DAG 的形式化定义如下：

定义 3（顺序图事件的有向无环图，Directed Acyclic Graph of UML Sequence Diagram Events）　对于一个顺序图 $SD=\langle O, M, E, \rightarrow, fem, feo\rangle$，SD 的一个 DAG 是以 SD 上事件为节点构成的有向图。可以表示为一个 2 元组：$DAG=\langle E, \rightarrow\rangle$。其中：

• $E=\{e_1, e_2, e_3, \cdots, e_n\}$ 是所有事件的集合，包括发送消息和接收消息的事件。其中 $e_i=(id, etype, sender, receiver, method, in\_degree, next1, next2)$，其中 id 是该事件节点在 DAG 中的唯一编号，$etype$ 是事件类型，可以为发送或接收消息，$sender$ 和 $receiver$ 分别表示消息的发送和接收者，$method$ 为消息所对应的方法名，in_degree 为节点的入度，$next1$ 和 $next2$ 用于存储该事件节点在顺序图上纵向和横向的直接后继；

• $\rightarrow$ 是一个表示 E 中事件之间的二元关系，$e_i \rightarrow e_j$ 表示在事件节点 e_i 到 e_j 之间有一条有向边，即 e_i 的 next1 或 next2 指向 e_j；

• 若存在有向边序列 $e_{i1} \rightarrow e_{i2} \rightarrow \cdots \rightarrow e_m$，则称节点 e_{i1} 到 e_m 之间存在一条路径（$e_i \rightarrow e_j$，$e_j \rightarrow e_k$ 传递得到的 $e_i \rightarrow e_k$ 不作为路径中的边），路经上有向边的数目称为路径的长度。如果存在一条路径，其长度≥2，且起点和终点为同一节点，称该路径是一个环路；

• 顺序图的 DAG 中任一路径都不是环路。

对于一个顺序图 SD，其 DAG 描述的事件顺序，是从顺序图到实现应该保持的系统本质信息。而对应的正确的软件实现，运行所得到的事件序列，都必须满足 DAG 所规定的拓扑

排序关系。相反，错误实现可能导致交互中发生行为偏差，偏离软件规约规定的功能。造成偏离的原因可能有很多种：消息名的编码错误、错误的参数，参与者缺少功能，错误的方法调用，错误输出，异常未处理等等。软件行为的偏离导致实现和需求的不一致，即顺序图被错误地实现了，那么至少有一条实际运行时的行为与 DAG 产生了不一致。为了发现不一致，定位错误的来源，可以采用随机测试的方法，根据随机输入多次执行程序，将软件的运行时的动态行为与 DAG 做比较，验证是否一致。定义软件运行时的执行轨迹如下。

定义 4（软件的运行时执行轨迹，Runtime Executing Trace of the program） 程序的运行时执行轨迹 ret 是一个实际发生的方法调用和执行的事件序列 $re_1, re_2, re_3, \cdots, re_n$，$re_i = \langle (mtype, o\_s, o\_t, method\rangle$ 其中 *mtype* 可为 *mc* 或 *me*，分别表示方法调用和方法执行，o_t 为方法调用的目标对象名，o_s 为发出方法调用请求的对象，*method* 是当前对象调用的或执行的方法。

软件的运行时执行轨迹记录了软件在运行时的动态行为，验证它们是否正确实现了顺序图描述的系统的预期行为是本文的主要工作。由于本文研究的消息传递方式是方法调用，软件的预期行为需要满足的属性表现为方法调用和方法执行的 DAG。而面向对象软件系统通过方法调用传递消息，其运行时执行轨迹也表现为一个方法调用和方法执行的序列。因此要对表示实际行为的事件序列进行验证，观察是否与表示预期行为属性的 DAG 相符，如果相符，则说明顺序图所表示的行为被正确实现，下面给出顺序图被正确实现的定义。

定义 5（顺序图被正确实现） 对于一个顺序图 SD 及其相应的待测程序实现 IUT（Implementation Under Test），SD 的预期行为属性为 DAG，随机生成的测试用例集合为 *tc*，当用 $\forall tc_i \in tc$ 执行 IUT 时，在运行时的实际执行轨迹为 ret_i，当 ret_i 中包含了某一连续片断与 DAG 中的事件先后顺序完全吻合，称 SD 被该 IUT 正确实现。

2.2 顺序图的可测试性要求

我们使用 Rational Rose 进行需求、设计的建模，而 UML 模型语义本身的不精确性和工程师设计思维的多样性使得模型的可测试性很难得到保证。因此，为了能直接应用顺序图作为测试资源，模型必须满足以下可测试性要求：1. 假定顺序图描述的交互与用例图描述的规约是一致的。模型本身的验证是通过非形式化的复审和形式化的模型检验方法进行的，已超出本文的研究范围。而保证顺序图与用例之间的一致性，使得测试更有针对性，可以有效地剔除噪音数据。2. 本文只研究针对检错进行的动态交互行为的测试，为了明确解决问题，假定消息都表示本地过程调用。3. 至少有一个执行者（actor）和一个对象（object）。每一个消息发送，都能在某对象找到对应的方法。为了可以与运行时轨迹进行比较，必须完整地填写类名和方法名，并提供 getName() 方法，使得分析程序可以获得对象所属的类名。4. 因为同步消息使得事件数目大为增多，所以要求顺序图规约所描述的场景中只发送异步消息，需要同步的消息用 1 条返回消息来模拟。5. 因可以通过路径选择的展开化简顺序图，不考虑条件分支、循环和并行执行。满足以上条件的顺序图，称之为“可测试的顺序图”，可以在基于顺序图的动态测试方法中，用于描述系统场景的规约。

基于上述定，下面介绍基于顺序图的运行时测试方法。

3 面向场景规约的运行时测试方法

为了验证顺序图描述的软件系统预期行为是否被正确实现，将顺序图及其相应代码实现作为方法的输入，在顺序图成为基线时就开始测试设计工作，与代码实现并行展开。

首先，对顺序图进行分析，产生表示预期行为属性的 DAG。当代码成为基线时开始测试的执行工作，将选定的顺序图中所关心的方法在源码中进行插装，使得软件在运行时能记录下这些方法的调用和执行序列。然后，用随机测试用例执行插装过的代码，并将输出的方法调用和执行序列记录到一个轨迹文件中，利用反向工程方法从输出信息记录文件重构并可视化软件的运行时执行轨迹。最后，将分别表示软件系统预期行为属性和实际行为的 DAG 和运行时执行轨迹（RET）进行比较，从而确认实现的行为与设计的行为的一致性。

3.1 从顺序图生成 DAG

为了验证顺序图中的交互行为是否被正确实现，首先需要得到作为系统预期行为属性的 DAG。我们从顺序图中获取消息事件之间的先后顺序关系，在有前驱-后继关系的节点之间画一条有向边，形成事件的有向无环图。实际的顺序图上的消息可能包括条件发送或重复发送，在增加了顺序图的表达能力的同时，增加了模型分析的复杂性。但是，通过判定/循环覆盖原则可以遍历所有可能的消息序列，因此在顺序图的可测试性要求中添加了不发送重复或条件消息的要求，将处理流程简单化，又不影响方法的正确性。根据顺序图中的消息列表，为每个消息新建 2 个事件对象，根据定义 1 中定义的 3 种后继，为每一个节点寻找其纵向、横向后继。最后设定每一个节点入度，参见算法 1。

算法 1 DAG 生成算法

```
//算法名：GenerateDAG(由 CDAGGenerator 类实现)
//输入：SequenceDiagram(顺序图)；输出：DAG 对象(有向无环图)
//算法开始
Declare rerDAG as DAG;
For each Message in Sequence Diagram's Mesage list
Begin
      创建 Message 的 Send，Rceeive 事件；
      将其的入度设为 0
      将它们添渐加到 retDAG 的 event list;
End for
For each event in retDAG's event list
Begin
      Switch Type of the event
            Case Send:
            后一个事件节点为其横向后继，其后面第二个事件节点
            为其纵向后继；
            Case Receive:
              如果有同一信道的后继接受消息事件，则为其纵向后
              继
              向后遍历，该事件所属对象的下一个发送事件为其纵
              向后继
      end Switch
End for
For each event in retDAG's event list
    For each subsequent node of event
    Begin
          Subsequent node 入度＝入度＋1
    End for
Begin
End for
Return retDAG
```

3.2 随机测试用例

测试用例的生成一直是测试领域的核心课题。好的测试用例没有统一的标准，需要根据被测试的系统及测试目的来综合评定[8]。顺序图描述的信息，是系统行为的本质信息，也包含作为测试模型的顺序图中对生成测试用例有用的信息，需要在从设计到实现的过程中保持。如果能用一定量的用例

执行程序，并证明顺序图上的所有测试需求(用 DAG 中的事件顺序来描述)都在实现系统中得以保持，就可以说测试用例是充分的。采用了让用户输入变量的类型、值域的方法，自动生成随机测试用例，提高了自动化程度。

3.3 顺序图制导的代码插装

代码插装是获取系统运行时信息的有效方法，可以将测试感兴趣的运行结果和运行轨迹记录下来，供后续分析之用。由于系统运行轨迹需要的信息是实现系统的内部信息，无法从运行结果直接得出，因此需要对代码进行插装。本文采用 Java 实现系统，消息发送和接收对应着运行时方法的调用和执行，我们根据从顺序图中提取出的类，以及相关方法的信息，在 java 源代码中执行插装，插入探测语句使得目标系统在运行时输出执行轨迹。为了方便反向工程生成运行时轨迹，还添加了一些附加信息，如构造方法、方法名返回等功能，这些信息不是系统必须保持的本质属性，但对于测试流程来说是不可缺少的。反向工程生成的运行时轨迹(RET)由程序执行生成的运行时轨迹文件自动导出，被用于与 DAG 的比较。

本文的插装算法详细参见算法 2。本文的方法同样适用于其它面向对象语言的实现。

算法 2 Java 代码插装算法

```
//Algorithm:InterpolateCode
//Input:类列表 Clist,方法列表 Mlist,代码列表 JList;输出:插装过的代码;
//Algorithm Begin
Declare cpJList as copy of JList,proneCode as String,LogFile as File;
For each JavaCodeFile in cpJList
Begin
    While (JavaCodeFile 未到文件尾)
        在 JavaCodeFile 中扫描新的一行
        If 发现一个函数中 m 且 m 在 MList 中 Then
            If m 是函数调用 Then   //找到函数调用插装点
                proneCode={
                        mid++;
                        cname=this. getClass(). getName();
                        boolean inMList = checkCName(cname,CList);
                        if(inMList){
                            mtype=mid;
                            trace=mid. toString()+cname+m. getMethodName();
                            LogFile. writeBytes(trace);
                        }
                        else
                            mtype=-1;
                }
                在 JavaCodeFile 的前一行插入 proneCode
            Else If m 是函数定义 Then   //找到函数定义插装点
                proneCode={
                        i=Integer. parseInt(mtype);
                        if(i ! =-1){
                            cname=this. getClass(). getName();
                            trace=i. toString()+cname+m. getMethodName();
                            LogFile. writeBytes(trace);
                        }
                }
                在该方法的第一行插入 proneCode
            Else
            End if
        End if
    End while
End for
Return cpJList;
```

3.4 运行时执行轨迹和 DAG 的比较

为了验证代码和设计之间的一致性，将所有的运行时轨迹和 DAG 进行比较，RET 与 DAG 一致，表示顺序图体现的本质信息得以保持，如果发现不一致，可以通过偏离发生的位置定位系统出错信息，供迭代开发的使用。将运行时轨迹反向工程作为顺序图，然后验证与设计的顺序图是否一致，但这种方法非常复杂，我们采取了折中的方法，将顺序图生成 DAG，然后与 RET 比较一致性。根据定义 5 对一致性的形式化定义，先将程序生成的轨迹文件反向生成 RET，再对应转换为 DAG 中事件的列表，然后采用改进的深度优先遍历算法，试图寻找一个完整的片断与 DAG 完全匹配。

从系统实现中反向得到 RET，因为插装信息是根据顺序图实体得来的，故可以为 RET 中的事件节点找到其在 DAG 中的对应节点，将 DAG 和 RET 的节点 ID 列表进行匹配，就可以验证是否正确实现了顺序图。根据定义 4，如果能在 RET 中找出一个连续片断，其中所有 DAG 的节点在 RET 中出现且仅出现一次，并满足 DAG 中的事件关系，就称该 RET 满足顺序图的预期行为属性，否则称预期行为发生偏离。本文采用从零入度节点逐个删除的方法，在遍历 RET 的过程中，将 DAG 的复制中的节点删除，同时更改 RET 中其后即节点的入度，如果发现 DAG 中剩余节点为 0，则可以返回一个精确匹配，(参见算法 3)。

算法 3 DAG 与 RET 一致性比较算法

```
//算法名:CompareARetWithDAG
//功能:将反向工程生成的 RET 与 DAG 的拓扑排序关系进行比较
输入:DAG,ret 上事件的起始和终止位置 rbegin、rend
输出:true or false,ret 中匹配 DAG 的事件序列的头和尾 dbegin、dend
备份 DAG 的入度表
rbegin,rend 分别取为 ret 的头事件和尾事件节点
print 指向 ret 的第一个事件//从 ret 的第一个事件开始匹配
dbegin=point//记下起始位置
while dbegin 不是 ret 尾事件//新的一次匹配开始
    DAG 节点访问标记清空
    while point 不是 ret 的尾事件//找到起始事件
            在 DAG 中查找当前事件所对应的节点 node
            if(node. indegree! =0)//如果当前节点的入度为 0,即为当前的开始事件
                point++//否则继续找 ret 扣下一个在 DAG 中入度为 0 的事件
                loop
            else
                break;
            endif
        end while
    while point 不是 ret 的尾事件
        在 DAG 中查找 point 的事件所对应的节点 node
        置 node 节点访问标记
        if node. indegree=0 then
                node 的横向后继和纵向后继(如果存在)的 indegree 各减!
                point++
                loop
        else
            exit
        end if
    end while
    if DAG 节点都被访问一次 then
        dend=point//记录成功匹配 DAG 的 ret 上的事件序列的尾
        return true
    else
        dbegin++
        point=dbegin
        重置 DAG 入度表
        Loop
    endif
endwhile
return false//没有匹配成功
end of function
```

4 运行时测试工具 SDT 的实现

为了支撑上述运行时测试方法，我们设计了一个基于 UML 顺序图模型的自动测试工具 SDT。SDT 首先利用 Rose 扩展接口(REI)[10]从 Rational Rose 的规约文件中提取作为场景规约的 UML 顺序图，生成 DAG；根据顺序图的分析，对 java 代码进行插装；根据用户设定生成随机测试用例；执行程序并由运行轨迹反向工程生成 RET；通过验证 RET

是否同 DAG 一致验证程序是否正确实现了顺序图规约的场景。

由于 SDT 中实体、操作等基本元素具有鲜明的对象特性，我们采用面向对象方法开发 SDT，并用 UML 对其本身进行了分析和建模。下面介绍 SDT 的设计实现过程。

4.1 分析模型

本节介绍了在分析阶段建立的模型包括工具的用例图、类图。

验证顺序图和代码之间的一致性，是 SDT 的最终目标，而达到这个目标需要经过导入顺序图、生成 DAG、生成 RET、比较 RET 和 DAG 四个步骤，其中生成 RET 的部分又包括插装代码、生成随机测试用例、自动执行、重构 RET 四个部分；导入活动图的过程需要测试工程师提供 Rational Rose 的规约文件名，并且需要利用 Rose 扩展接口。用例图(图 2)如下。

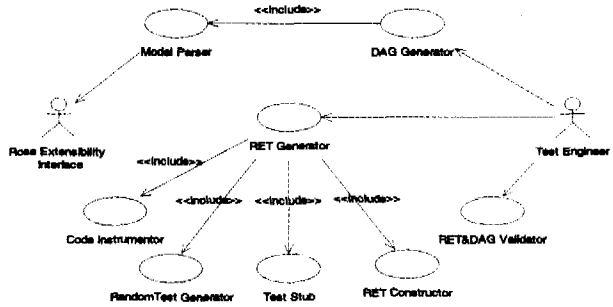

图 2 用例图

根据用例，细化出测试工具的执行流程，用活动图表示，如图 3 所示。

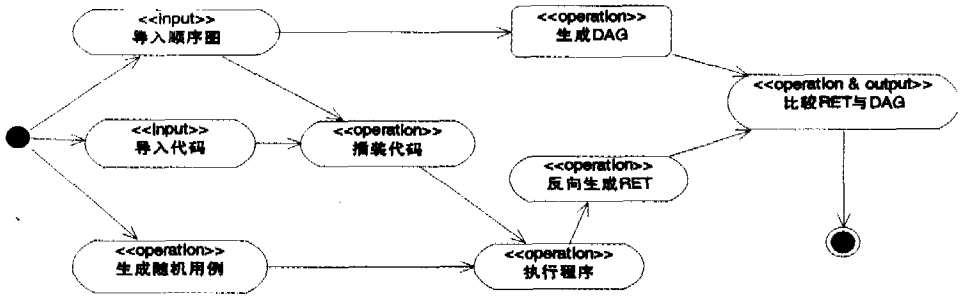

图 3 活动图

根据用例图和活动图，可以初步确定几个分析类(Analysis classes)。其中 Sequence Diagram 类，DAG 类，Test Case 类，IUT Code 类和 RET 类是实体类，负责储存数据；Model Parser 类，DAG Generator 类，Code Instrumentor 类，Stub 类，RET Constructor 类，Test Case Generator 类和 Validator 类，是控制类，负责处理数据。

4.2 设计模型

本节介绍设计阶段的工作，包括系统类图和实体类的存储结构。我们使用 VC. net 实现了工具的核心模型和用户界面；使用了 C++ STL 库(Standard Template Library)中的容器类[17]实现了实体类的存储；使用文[11]提供的模型和显示模块，所有的模型实体都是 CLeafItem 对象，所有实体对应的图形对象为 CShape 抽象类的子类对象。

4.2.1 类图 系统主类图如图 4 所示：CSDParser 类将 UML 顺序图从 Rose 文件中提取出来，存入 CSequenceDiagram 类中；CDAGGenerator 类根据顺序图中的事件的时间顺序生成 DAG，存入 CSDEventDAG 类中；CSDRETGenerator 中的 InterPolateCode 方法根据顺序图实体中的信息将 CJavaCode 所管理的代码插装，GenerateRET 方法根据随机测试用例执行插装过的代码，生成程序运行轨迹信息，LoadRetFromFile 方法从文件中读取信息到 CSDRET 类；CSDTraceComparor 类中的 Compare 方法将 RET 与 DAG 相比较，返回测试结果。

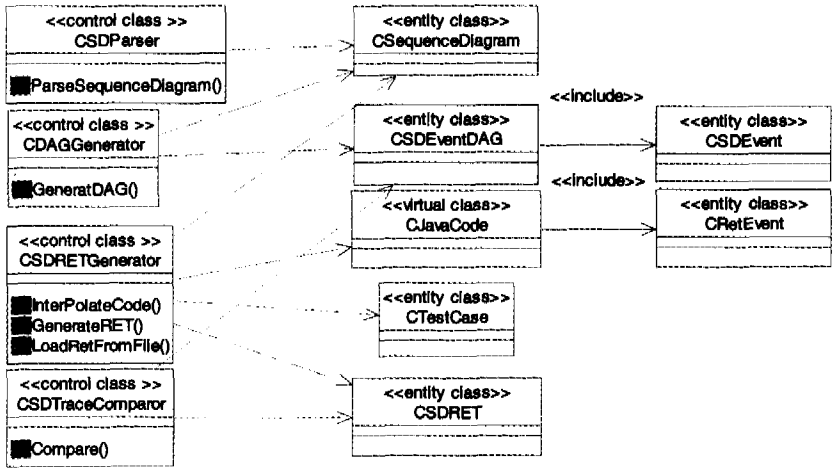

图 4 主类图

4.2.2 实体类的存储结构 顺序图(CSequenceDiagram)中包括对象、消息，而这两种实体都是 CLeafItem 对象。采用 STL 中的 map 容器来存储这些节点，map 中关键字字段为元素的唯一 ID(从 REI 中提取)，第二个元素为对象指针。

而对象和消息的ID分别存入一个vector容器中，每天加一个实体到顺序图中时，先增加ID到vector，然后再以这个ID关键字，将对象指针加入map中。在检索、添加、删除等操作中，可以充分利用map容器的高效定位功能。

事件(CSDEvent)为预期事件，从顺序图中抽取；包括发送消息(方法调用)、接受消息(方法执行)。每个事件可能有2个后继，如果为发送消息，则接受事件为其横向后继；同一对象相邻发送事件中的后者是前者的纵向后继，或者同一信道(相同的发送者和接受者)相邻接受事件中的后者是前者的纵向后继。每一个事件节点有入度，即被作为后继的次数，可能为0(起始节点)，1(有纵向或横向前驱)，或者2(有纵向和横向前驱各一个)。

有向无环图(CSDEventDAG)为CSDEvent对象的集合，并通过产生DAG的算法保证了其中没有环的出现。CSDEventDAG存储了系统的预期行为属性。

运行时事件(CRetEvent)为实际运行轨迹反向之后的信息，包括类别(方法调用或方法执行)，调用者，执行者，方法名。反向成功后可以在DAG中找到对应的CSDEvent对象的ID。运行时事件轨迹(CSDRET)是CRetEvent对象所对应ID的列表，存在vector容器中，占用较少的资源且极大提高与DAG比较的效率。

4.3 测试工具SDT的实例研究

本文以ATM取款机的代码实现为例演示工具SDT的运行，输入顺序图为图1定义的顺序图。具体场景：ATM存钱动作，验证输入金额的数目，如果为奇数则允许存入并修改相关纪录，生成日志信息。工具中实现了模型分析、DAG生成、随机测试用例生成、代码插装、测试自动执行、RET重构、运行时行为验证等模块。实验中，首先导入顺序图，用DAG生成功能产生系统预期行为属性；然后将相关代码实现(共5个java源程序)与项目相关联，调用插装模块进行代码植入(左上角Code Manager显示代码已经被插装，图标改为带红P标记)；再添加随机测试用例，生成了20个从0到10000的随机数，作为输入调入执行模块；反向生成的RET自动显示在树状结构中，左上角是其中一个RET的回显；最后调用比较模块，根据比较结果更新相关RET的图标，见左侧树状结构。测试的结果是：所有的奇数输入生成的RET与DAG匹配，而偶数生成的RET在验证过程中发生不一致。

运行结果如图5所示。

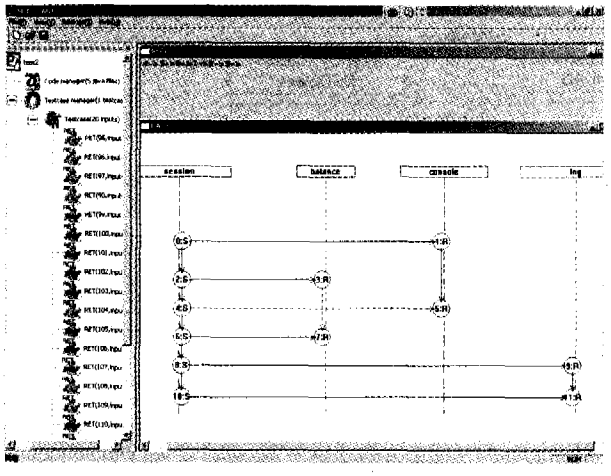

图5 SDT运行界面

相关工作和总结 近年来基于UML模型的测试工具的研究取得不少成果，有很多试图依赖于系统设计信息(如状态机)或者引入新的形式化方法实现在测试中对模型的利用，但是这些方法提高了专业化要求和实施难度。SCENTOR[12]利用顺序图生成JUnit的测试用例，对J2EE平台的项目进行测试生成和验证。AGEDIS[13]是比较成功的测试研究项目之一，它采用UML和AML(ADEDIS Modelling Language，项目内建语言)作为建模语言，生成测试用例，执行并分析结果。SeDiTeC[9]将顺序图进行分解处理，并将得到的顺序图作为测试数据的规约，生成用例并制导执行，但需要将运行结果反向为顺序图后与源顺序图比较，反向生成的过程可能存在歧义，顺序图之间的比较又可能隐藏同构模型间的一致情况，导致分析结果不精确。文[14]初步探索了直接使用交互图模型生成测试，也给出了基于交互序列进行插装以确定测试覆盖，但只研究静态检查方法，没有能够给出生成用于动态测试的方法；文[15,16]系列地介绍了一个利用交互图进行测试的方法，但由于用了范畴划分方法以及加权方法，需要很多用户交互操作。文[18]介绍了从协作图生成测试的方法，遍历各条路径获取路径条件、参数变量和预期方法序列，生成用例。上述方法基本上是覆盖了顺序图上的消息序列，本文的方法达到了更细粒度的覆盖，即事件覆盖，这是面向对象软件系统的动态行为测试中最充分的测试集，能够测试系统行为的所有可能场景，是测试用例细节过多或过少的一种折中选择；且完全基于UML，直接利用顺序图模型，避免了重构测试模型或模型转换的开销，在生成测试用例时采用的随机方法从输入域中选择数据，在很大程度上实现了自动化，减低了测试成本。

本文提出了面向UML顺序图场景规约的动态行为测试方法，在顺序图成为基线后，与系统实现并行进行测试设计，基于UML顺序图生成表示软件预期行为属性的事件DAG；当代码成为基线时开始测试的执行，首先将顺序图中对象的方法的调用和执行在源码中进行插装，使得软件在运行时能记录下实际的方法调用和执行序列，并将这个方法执行序列记录到一个轨迹文件中，从而能够利用反向工程方法从轨迹文件重构软件的运行时执行轨迹(RET)，并可视化表示；最后比较表示系统预期行为属性的DAG和系统实际行为的运行时执行轨迹，从而确认实现的行为与顺序图描述的行为的一致性。

这种模型驱动的软件测试方法在四个方面比传统测试方法有明显优势：一是在测试设计过程中复用软件开发工件，直接使用描述软件动态交互行为的顺序图作为测试模型，避免了重构测试模型的开销，并使用随机方法生成测试用例，提高了自动化程度，减少了整体测试的时间；二是在顺序图成为基线后就可以开始测试设计工作，可以在代码实现前或者同步生成测试用例，有利于合理使用测试资源；三是当软件需求发生变化时，用顺序图来捕获变化的需求，只需从新的顺序图重新生成测试用例，其余的处理与原来一样，这样能够灵活处理不断变化的需求，并将变更影响到的测试加入到原有的测试中；四是在测试设计过程中灵活运用正向工程与反向工程相结合的方法，仅将在预期执行轨迹中出现的方法调用和执行信息制导在源码中进行插装，使得方法集中于动态行为相关的本质信息，减少了对无关信息跟踪的开销。

尽管UML的应用相当广泛，但相应的测试实用方法和支持工具还不多见。本文的方法基于UML模型生成数量较

(下转第166页)

为了说明实验仿真的结果，定义加速比指标 λ 如下：

$$\lambda=(1-\frac{TH+TH_{\max}}{TS})$$

其中：$TS=\sum_{i=1}^{n}Ts_i$ 是全部任务均由软件实现所用的执行时间；$TH_{\max}=\max_{i=1}^{n}\{Th_i\cdot x_i\}$ 是问题 ρ' 解中由硬件实现的任务，它们所需要的最长的硬件执行时间，假设硬件实现的任务区不是并行执行的；$TH=\sum_{i=1}^{n}Th_i\cdot x_i$ 是问题 ρ' 解中由硬件实现的全部任务执行所需要的时间之和。

算法分别取任务数 $n=30,50,70$ 三组进行比较，仿真结果如图 3 所示，仿真结果表明任务数目的多少对加速比影响不大，影响加速比的关键因素就是硬件的有效面积。

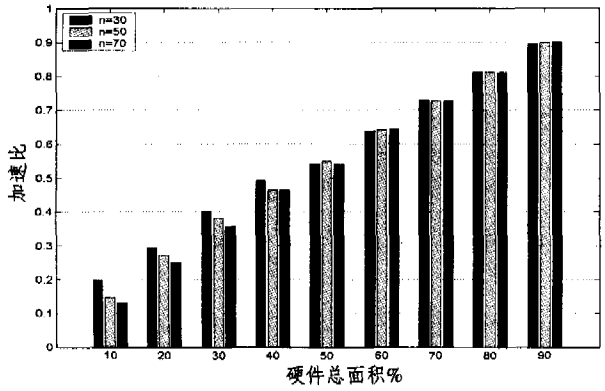

图 3　算法仿真结果比较

小结　对于嵌入式系统设计而言，软硬件的划分对系统性能有着重要的影响。本文给出了在高频访问路径的基础上，采用贪心算法中经典的 0-1 背包模型进行求解软硬件划分问题是一种新的思路，在忽略软硬件任务间通信开销的基础上，可以在时间复杂度为 $O(n\log n)$ 的情况下获得问题的近似最优解。将通信开销纳入求解的空间内是下一步工作需要考虑的内容。

参 考 文 献

1　Ernst R, Henkel J, Benner T. Hardware-Software Cosynthesis for Micro-controllers. IEEE Design and Test of Computer, 1993, 10(4): 64～75

2　Harkin J, McGinnity T M, Maguire L P. Partitioning methodology for dynamically reconfigurable embedded systems. IEE Proceedings Computers and Digital Techniques, 2000, 147(6): 391～396

3　Niemann R, Marwedel P. Hardware/software partitioning using integer programming. In: Proc. of IEEE/ACM European Design Automation Conf. (EDAC), 1996. 473～479

4　Gupta R, Micheli G D. Hardware-software cosynthesis for digital systems. IEEE Design and Test of Computers, 1993, 10(3): 29～41

5　Gupta R K, Coelho C, Micheli G D. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In: Proc. of 29th ACM, IEEE Design Automation Conf. 1992. 225～230

6　Vahid F, Gajski D D, Gong J. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In: Proc. of IEEE/ACM European Design Automation Conf. (EDAC), 1994. 214～219

7　Vahid F, Gajski D D. Clustering for improved system-level functional partitioning. In: Proc. 8th IEEE/ACM Int. Symp. System Synthesis, 1995. 28～33

8　Jinwoo S, Dong-In K, Crago S P. A communication scheduling algorithm for multi-FPGA systems. In: Proc. IEEE Symp. Field-Programmable Custom Computing Machines, 2000. 299～300

9　Melski D. Interprocedural path profiling and the interprocedural express-lane transformation: [PhD thesis]. University of Wisconsin, 2002

10　Madsen J, Grode J, Knudsen P V, Petersen M E, Haxthausen A. LYCOS: The Lyngby co-synthesis system. Design Automation for Embedded Systems, 1997, I: 195～235

11　Pisinger D. Algorithms for knapsack problems: [Ph. D. Thesis]. University of Copenhagen, 1995

12　Edwards S A, Lavagno L, Lee E A, Sangiovanni-Vincentelli A. Design of Embedded Systems: Formal Models Validation, and Synthesis. Proceedings of the IEEE, 1997, 85(3): 366～390

13　Knudsen P V, Madsen J. PACE: A dynamic programming algorithm for hardware/software partitioning. In: Proc. of 4th IEEE/ACM Int. Workshop Hardware/Software Codesign, 1996. 85～92

(上接第 152 页)

少覆盖程度高的测试用例，能够部分实现自动化而不过多增加用户额外的工作量，这样的测试方法容易被已经使用 UML 的工业界采用，但可测试性要求较多，下一步的工作是使方法和工具能处理利用更多的模型信息，往实用化方向发展。

参 考 文 献

1　UML Specification 1.5. Available at: http://www.uml.org Jan 2005

2　Booch G, Rumbaugh J, Jacobson I. The Unified Modeling Language User Guide. Addison-Wesley, 2001

3　Booch G, Rumbaugh J, Jacobson I. The Unified Software Development Process. Addison-Wesley, 2001

4　Kruchten P. The Rational Unified Process -An Introduction, 2nd edition. Addison-Wesley, Reading, MA, 2000

5　Binder R V. Testing Object-Oriented System: Models, Patterns, and Tools. Addison-Wesley, 2000

6　Beck K. Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, MA, 1999

7　Booch G, Rumbaugh J, Jacobson I. The Unified Modeling Language Reference Manual. Addison-Wesley, 2001

8　Bertolino A. Software Testing research and practice. In: 10th Intl. Workshop on Abstract State Machines, 2003

9　Fraikin F, Leonhardt T. Sequence Diagram Based Test Automation. In: Proc. of the 17th IEEE Intl. Conf. on Automated Software Engineering, Edinburgh, Sep. 2002

10　Rational Rose@, Using the Rose Extensibility Interface. http://www.cs.rhul.ac.uk/CompSci/Computers/rational/pdf/rose_REI_guide/Rose_REI_guide.pdf, Jan. 2005

11　Wang Lin- Zhang, Li Xuan-Dong, Zhen Guo-Liang. Generating Test Cases from UML Activity Diagram Based on Gray-Box Method. In: 11th Asia-Pacific Software Engineering Conf. (APSEC'04)

12　Wittevrongel J. The Scentor Web Site. http://sern.ucalgary.ca/~milos/etesting/scentor, Jan. 2005

13　The AGEDIS project. http://www.agedis.de, Jan. 2005

14　Abdurazik A, Offutt J. Using UML Collaboration Diagrams for Static Checking and Test Generation. In: Proc. 3rd Intl. Conf. on the Unified Modeling Language (UML'00), York, UK, Oct. 2000. 383～395

15　Basanieri F, Bertolino A. A Practical Approach to UML-based Derivation of Integration Tests. In: Proc. of QWE2000, Bruxelles, Nov. 3T

16　Basanieri F, Bertolino A, Marchetti E. CoWTeSTt: A Cost Weighted Test Strategy. In: Proc. of ESCOM-SCOPE 2001, London, England, April 2001

17　Josuttis N M. C++标准程序库. 武汉：华中科技大学出版社，2002

18　王林章，李宣东，郑国梁. 基于 UML 协作图生成测试用例. 电子学报，2004，32(8)