一种基于认知模型检测的 Web 服务组合验证方法

骆翔宇[1),2)] 谭 征[3)] 苏开乐[4),5)] 吴立军[6)]

1)(华侨大学计算机科学与技术学院 福建 厦门 361021)

2)(清华大学软件学院 北京 100084)

3)(中核兰州铀浓缩有限公司 兰州 730065)

4)(北京大学教育部高可信软件重点实验室 北京 100871)

5)(浙江师范大学数理信息学院 浙江 金华 321004)

6)(电子科技大学计算机科学与工程学院 成都 410073)

摘 要 近几年 Web 服务组合的形式化验证逐渐成为研究热点. 模型检测作为形式化验证的一种主流技术, 可以克服传统软件测试用例生成不完备的不足, 同时具有验证自动化的优点. 该文提出并实现了一种 Web 服务组合的认知模型检测方法, 将 Web 服务组合建模为多主体系统, 在分析 BPEL 语言控制流程基础上, 提出 BPEL 活动的形式化模型, 给出活动执行语义. 进而以迁移七元组为中间形式, 开发从 BPEL 流程到迁移七元组集合以及从这些迁移七元组到 MCTK(一种我们开发的多主体系统模型检测工具)输入语言的自动转换算法, 最终通过 MCTK 进行验证. 实验结果表明开发的算法不仅可以有效验证 Web 服务组合的时态逻辑规范, 而且可以验证多主体系统特有的认知逻辑规范及其时态组合.

关键词 模型检测; Web 服务组合; BPEL; 认知逻辑; 多主体系统

中图法分类号 TP301 **DOI** 号: 10.3724/SP.J.1016.2011.01041

A Verification Approach for Web Service Compositions Based on Epistemic Model Checking

LUO Xiang-Yu[1),2)] TAN Zheng[3)] SU Kai-Le[4),5)] WU Li-Jun[6)]

1)(*College of Computer Science & Technology, Huaqiao University, Xiamen, Fujian* 361021)

2)(*School of Software, Tsinghua University, Beijing* 100084)

3)(*China National Nuclear Corporation, No 504 Plant, Lanzhou* 730065)

4)(*Key laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing* 100871)

5)(*College of Mathematics Physics and Information Engineering, Zhejiang Normal University, Jinhua, Zhejiang* 321004)

6)(*School of Computer Science and Engineering, University of Electronic Science and Technology, Chengdu* 410073)

Abstract In recent years, the formal verification of Web service compositions is gradually becoming a hot research area. As a mainstream technique for formal verification, model checking is able to not only overcome the shortcoming of traditional software test techniques that they cannot generate the complete set of test cases, but also automatizes the verification process. In this paper, the authors propose and implement an epistemic model checking approach for the verification of Web service compositions by modeling them as multi-agent systems. After analyzing the BPEL control flow, the authors first propose a formal model for the BPEL activities and give the semantics of the implementation of the BPEL activities. By taking the so-call transition seven-tuple as

收稿日期: 2010-11-20; 最终修改稿收到日期: 2011-05-18. 本课题得到国家"九七三"重点基础研究发展规划项目基金(2010CB328103)、国家杰出青年科学基金(60725207)、国家自然科学基金(60763004、61073033)、中国博士后科学基金(20090450389)、华侨大学中央高校基本科研业务费项目(JB-GJ1001)和华侨大学高层次人才科研启动费项目(11BS108)资助. 骆翔宇, 男, 1974 年生, 博士后, 副教授, 主要研究方向包括模型检测、时态逻辑、认知逻辑、知识推理、多主体系统、安全协议验证等. E-mail: shiangyuluo@gmail.com. 谭 征, 男, 1982 年生, 硕士, 主要研究方向为模型检测、Web 服务. 苏开乐, 男, 1964 年生, 博士, 教授, 博士生导师, 主要研究领域包括模型检测、知识推理、非单调推理、多主体系统、模态逻辑、时态逻辑、概率推理、安全协议验证、逻辑程序设计等. 吴立军, 男, 1965 年生, 副教授, 主要研究方向为人工智能与网络安全.

the intermediate form, the authors further develop two automatic transform algorithms, one converts the BPEL process to the set of seven-tuples and the other converts this set of seven-tuples to the input language of MCTK (a model checker for multi-agent systems developed by the authors), such that can finally implement the verification by MCTK. The experimental results show that the proposed algorithms can effectively verify not only the temporal logical specifications, but also the temporal epistemic logical specifications, which are proper to multi-agent systems.

Keywords model checking; Web service compositions; BPEL; epistemic logic; multi-agent systems

1 引 言

随着面向服务的体系架构(SOA)的出现,许多软件资源与应用都封装成为服务,服务提供以功能为单位的调用标准,对服务的调用者展现统一的调用接口,而屏蔽服务的实现细节.可以说 SOA 将是80%的开发项目的基础,并且成为主流的软件工程实践方法之一.不过,当用户需要以某种定制的次序或规则调用多个功能或服务时,Web 服务标准本身就显得无能为力了,而这恰恰是 Web 服务业务流程执行语言 WS-BPEL(简称 BPEL)所关注的焦点.BPEL 将 SOA 系统中的孤立服务按照预订的规则进行调度与协调,从而提供有价值的流程服务.BPEL 的这一特点使得其在 SOA 架构中具有固有的优势,被众多厂商所采用.

然而,由于服务及其协同的动态性,开放多变的互联网运行环境,以及松耦合的服务开发模式所导致的开发和运行过程不确定性,使得服务的正确性、可靠性、安全性、可用性、时效性等可信性质难以得到保证.如果单纯采用传统的软件测试方法,许多这样的可信性质是无法表示和测试的.另外,传统的测试方法也难以保证生成的测试用例集是完备的.因此,自从 BPEL 标准发布后,研究者们在 BPEL 的形式化建模和分析领域做了不少研究工作.Fu 等人开发了针对 Web 服务的验证工具WSAT(Web Service Analysis Tool),将 BPEL 转化为形式模型 Guarded 自动机,接着将 Guarded 自动机转化为模型检测工具 SPIN 的输入语言,验证线性时态规范[1-2];Kazhamiakin 对通信模型、控制流、数据流和时间属性进行形式化建模和分析,并且开发了 BPEL 验证工具 WS-VERIFY 对上述属性进行验证[3];Foster 等人使用进程代数的方法把 BPEL 建模成 FSP,使用模型检测工具 LTSA 对其验证[4-5];Walton 把 Web 服务看作为多主体系统,定义了一种名为轻量级的协议语言表示 Web 服务内部主体的交互,并将这种语言转化为 SPIN 的输入语言对通信属性进行了验证[6];Nakajima 使用扩展的有限状态机(EFA)对 BPEL 进行建模,将 EFA 转换为 SPIN 的 Promela 语言,自动验证了 Web 服务组合的属性[7];Mongiello 等人将每一 Web 服务建模为有限状态机,然后再将所有的有限状态机转换为 NuSMV 输入语言,对其属性进行自动验证[8];Ouyang等人开发了一个自动分析 BPEL 流程的工具,先用工具 BPEL2PNML 将 BPEL 转化为 Petri 网,然后将工具 BPEL2PNML 的输出通过工具 WofBPEL 对活动的不可达及数据传递进行静态分析[9-10].

从上述研究进展看,目前 Web 服务组合的形式化验证方法的主流思想是针对被验证属性选择自动验证工具(通常是模型检测工具),提出该工具输入语言的形式模型,然后通过 BPEL 到形式模型以及形式模型到输入语言的转换,实现属性的自动验证.本文也沿用这种思路,但是我们侧重于对 BPEL 进行多主体系统建模,原由如下:在 W3C 工作组起草的 Web 服务体系结构文件中提出,Web 服务是一个抽象的概念,必须由具体的主体执行,该主体是一个具体的能够发送或者接收消息的软件或者硬件.而多主体系统(Multi-agent Systems, MAS)是指由多个相互交互的主体组成的系统,形成多个主体合作的问题求解网络.因此很自然地,我们可以把 Web 服务组合抽象为多主体系统,从而应用多主体系统模型检测工具对大多数基于传统模型检测工具(如 SPIN 和 NuSMV 等)的方法不能验证的多主体认知性质进行验证.比如时态认知性质 $p \rightarrow FK_i p$ 表示如果 p 成立,则最终(F 时态算子)主体 i 会知道 p 成立,其中嵌入的知识子公式 $K_i p$ 如果在状态 s 成立,则表示在主体 i 认为可能的(即与 s 不能区分

的)所有状态下 p 均成立,因此即使主体 i 当前不能肯定自己处于状态 s 中,它也可确定(知道)p 均成立.这类性质体现了主体 i 的认知能力,是其它非认知模型检测工具无法验证的.事实上,我们已开发出一种多主体系统符号化模型检测工具 MCTK[11-13],支持时态认知逻辑性质验证,其验证效率与类似的主流工具 MCMAS[14] 和 MCK[15] 相比均有不同程度的提高.基于上述情况,本文提出并实现一种基于 MCTK 的 Web 服务组合建模和自动验证方法学,支持通信通道的建模,使得我们能够直接以 BPEL 作为输入语言进行建模和验证,不仅可以验证 Web 服务组合的时态逻辑规范,而且还可以验证代表 Web 服务的主体的时态认知性质,这是传统模型检测技术所不支持的.据我们所知,与本文最相关的工作当属 Lomuscio 和 Qu 等人在文献[16-17]的工作,他们应用 MCMAS 对 BPEL 进行了关于时态和知识属性方面的验证,但对形式模型和语言自动转化算法未给出详细说明,对 Web 服务间的通信也未给出具体描述.更重要的是,我们的 MCTK 的时态部分支持 CTL*(包含 CTL 和 LTL),而 MCMAS 的时态部分仅支持 CTL,因此 MCTK 的时态表达能力更强.

本文第 2 节提出 BPEL 流程的形式模型并给出活动执行语义;第 3 节提出从 BPEL 到迁移七元组和从迁移七元组到 MCTK 输入语言的自动转换算法;第 4 节给出一个验证示例及其实验结果;最后在第 5 节给出结论并展望未来工作.

2　BPEL 形式模型和活动执行语义

2.1　BPEL 形式模型 BFM

由于单一 Web 服务的自治性和松耦合性,我们将每一 Web 服务建模为主体.这些主体广泛分布于网络中,因此组合 Web 服务可被视为由多个交互主体组成的多主体分布式系统(简称多主体系统),它由若干主体和一个公共环境组成,某些主体之间可以进行交互.这里,为了便于后续工作对主体交互的建模,我们引入"通道"的概念.通道用来"连接"两个主体,这些主体通过通道发送或者接收信息,交互的主体互相能观察到对方的一部分信息,同时主体还可以和环境进行交互,因此它也可以观察到环境的一部分甚至全部.主体根据自身的当前局部状态和收发的信息决定下一步动作,所有主体和环境的联合动作导致系统状态发生迁移.

我们采用扩展的有限状态机对 BPEL 进行形式建模,这种扩展的有限状态机首先清晰刻画了系统状态的迁移关系.其次,采用的底层验证平台是我们自行研发的多主体系统模型检测工具 MCTK(详见文献[11,13]),其输入语言实际上是有限状态机的符号化描述,该输入语言同时扩展了每一主体的可观察变量集合的定义.因此,该形式模型还应对信息的可观察性进行描述.为此,我们定义 BPEL 语言的形式模型 BFM(BPEL Finite-state Machine)如下:

$$BFM = (\Omega, O, \delta, P_0, F),$$

其中:Ω 为局部状态的有限集合;O 为主体可观察变量的有限集合,包括通道变量的有限集合 O_e、链接变量的有限集合 O_l 和条件变量的有限集合 O_g;δ 为 $\Omega\times\Delta\to\Omega$ 是状态迁移函数,其中 $\Delta = O_e\times O_l\times O_g\times channel\times dir$. $\forall(p,a)\in\Omega\times\Delta$,有 $\delta(p,a)\to q$,表示当前处于状态 p 时,由于主体执行活动产生条件 Δ 后迁移到状态 q,Δ 由主体执行活动产生;P_0 为系统初始状态,$P_0\in\Omega$;F 为终结状态集合,$F\subseteq\Omega$. $\forall p\in F$,p 为终结状态.

该模型中,Ω 表示主体执行过程中所经历的所有状态组成的状态空间.Δ 中 *channel* 表示与此主体"连接"的通道名,在用 MCTK 编程时,不仅要考虑状态之间的迁移关系,也要考虑与主体相"连接"的通道中消息与状态之间的关系以及通道中信息的变化情况.因此,需要知道与每个主体相"连接"的所有通道的名称,它即为 BPEL 活动的端口类型名〈portType name〉.本文中,假设所有通道名皆不相同,并且每个主体皆有两个单向通道与其相连,用于发送及接收消息,对于异步双向交互,两个〈portType name〉即为两个通道的名称,对于同步双向交互,人工定义一个〈portType name〉作为另一个通道的通道名.通道名的定义使得运用 MCTK 输入语言建模时更加方便,但是仅从通道名属性中并不能分辨出该通道是用来发送消息还是用来接收消息.因此,dir 表示此通道相对于与其"连接"的主体是发送消息通道(简称出通道)还是接收消息通道(简称入通道).BPEL 语言中的变量主要分为 3 种:O_e 为主体能观察到的通道变量的有限集合,主体能观察到的通道变量是存储此主体与其它主体或者环境之间交互信息的变量.因此,一个主体除了观察到自身内部变量外,还能观察到一部分通道变量.这些通道变量明确出现在 BPEL 程序的〈variables〉标签中,实际上它们也是主体之间的共享变量.O_l 为主体能观察到的内部链接变量的有限集合,这些变量可以从

〈links〉标签中的〈link〉抽出，〈link〉指定了并发执行活动之间的控制流，可以将这些变量看做布尔变量，当其为真时表示该控制流存在. O_g 为条件变量的有限集合，每个条件变量对应一个可以表达条件表达式真或假的断言，这些条件表达式可以出现在诸如〈while〉，〈switch〉，〈source〉中的〈*transition Condition*〉，〈*targets*〉中的〈*joinCondition*〉中，因此，每个条件变量也是布尔变量，变量值为真说明表达式也为真.

实际上通过上节对 BPEL 语言的分析，我们可以知道该语言主要用来对流程进行描述，它反映了主体之间的数据交互以及主体内部变量赋值等情况，但它无法具体描述主体因为信息交互而发生的状态改变，它可以说明在某种条件下控制流转移到了另一个 Web 服务，却无法说明信息转移后相关主体处于什么样的状态下. MCTK 是基于 NuSMV 开发的，引入了认知逻辑验证功能的模型检测工具，但其建模过程是一个针对具体状态迁移的描述过程，我们需要描述出各主体状态之间迁移关系以及相互联系的“通道”中信息的变化情况，前者可以用来说明主体的状态空间中各状态的关系，后者使得主体之间建立起联系. 因此，如果将 BPEL 语言直接转化为 MCTK 输入语言将是十分困难的，我们必须找到中间“桥梁”将 BPEL 语言与模型检测工具 MCTK 的输入语言联系起来.

为此，我们首先对流程经历的状态进行分析：

BPEL 中某些基本活动的执行可引起变量值的改变，这些变量包括用来与外界交互的变量，也包括内部通过赋值操作改变的变量，如果用这些变量的“与”操作来标示状态，那么这些变量中的任一变量值的改变将导致状态发生迁移. 例如，在状态 p_0 中 v_1、v_2 是外部变量，即用来存储与外部主体的交互信息，v_3 是内部变量(例如该变量利用 assign 赋值活动得到值)，并且变量 $v_1=3$，$v_2=4$，$v_3=5$，那么我们可以用 $(v_1=3)\wedge(v_2=4)\wedge(v_3=5)$ 标示状态 p_0. 若主体执行某活动从外界主体得到新的信息存储于变量 v_2，使得 v_2 的值发生变化变为 5，之后执行活动使得内部变量 v_3 的值变为 6，则系统由状态 p_0 迁移到状态 p_1：$(v_1=3)\wedge(v_2=5)\wedge(v_3=6)$. 但对于 empty 活动，由于活动执行后内部外部变量的值均不发生改变，但主体确实执行了此活动，因此可以加入程序计数器 PC 来标示这种活动的执行. 例如，系统在执行了活动 empty 后由状态 p_0：$(v_1=3)\wedge(v_2=4)\wedge(v_3=5)\wedge pc_0$ 迁移到状态 p_1：$(v_1=3)\wedge(v_2=4)\wedge(v_3=5)\wedge pc_1$. 由此可见，主体在执行过程中的所有状态均可由主体中所有变量与 PC 的合取来标示，因此变量或者 PC 任何一方的改变都将引起状态的迁移，而在 BPEL 流程中基本活动的执行会修改 PC 或者变量的值，结构活动与基本活动不同，前者主要用来说明在其内部的基本活动是按照何种顺序执行，或者顺序执行、或者并发执行、或者选择执行、或者循环执行等等. 因此，欲实现 BPEL 流程的自动化验证就需要在结构活动所提供的框架下，研究其内部基本活动的执行对系统产生的影响，主要包括对主体所处状态的影响，并自动将这种影响刻画出来.

直观上看，对主体所处状态的影响主要指主体相关变量值的改变，由于我们用变量的合取标示状态，活动的执行将导致变量值发生改变，状态自然发生迁移. 考虑到模型的简化，我们省去了状态记数器 PC，只研究变量值的改变对状态迁移产生的影响. 所以我们要找的“桥梁”必须能够直观反映出由于变量值改变引起的这种状态的迁移关系，这是其一；除此以外，该“桥梁”必须便于用 MCTK 对其进行描述从而进行形式化建模. 因此，结合 BPEL 语言以及 MCTK 输入语言的特点，我们还需考虑到：交互信息的“流动”方向，即信息是进入本主体还是离开本主体，这是其二；交互信息进入/离开主体时经过的端口，这是其三.

由此，我们所找的“中间桥梁”应考虑到上述 3 个条件：第 1 个条件用来对 BPEL 流程的执行进行描述，便于 MCTK 描述状态迁移关系，后两个条件是为了方便 MCTK 建模语言对通道机制进行描述.

由以上分析我们知道了 BPEL 活动的执行产生相对应的状态迁移过程，为了刻画这种状态的变化信息，本文提出迁移七元组概念. 它也是我们要找的“中间桥梁”.

一个迁移七元组(简称七元组)应具备如下形式：$\Phi(\Omega, \Delta, \Omega)$. 它将主体初状态、迁移条件、末状态组合在一起，迁移条件是主体执行某活动导致状态变迁过程中的相关信息值. 七元组展开后语法格式为：

$\Phi(cur(state), O_e, O_l, O_g, channel, dir, next(state))$.

其中：$cur(state)$ 为主体执行活动前的当前状态；O_e 为主体执行活动后的通道变量值；O_l 为主体执行活动后链接变量的值；O_g 为主体执行活动后条件变量的值；*channel* 为与主体连接的通道名；*dir* 表示对本主体是出通道还是入通道；$next(state)$ 为主体执

行活动完毕所处状态.

该七元组包含了主体活动执行完毕后状态的变化，由七元组看出状态的变化在这里主要指：

(1) 状态名从 *cur*(*state*)变为 *next*(*state*)，表明状态已发生迁移；

(2) 主体活动执行完毕后 O_e、O_l、O_g、*channel*、*dir* 的变化，这些信息不仅真实反映了 BPEL 活动执行导致的变量值的改变情况，同时也具备了 MCTK 建模所需的必要信息.

需要说明的是，完全运用变量与 PC 的合取标示状态，会对最后验证规范的书写带来很大困难，原因有两点：

(1) 大量变量的书写很麻烦，尤其当系统的状态空间很大时，为表示状态又需要书写多个变量的合取，很容易出错，而实际中的软件系统大多具有超大规模状态空间，因此这种状态表示法并不实用.

(2) 由于规范需要人为书写，因此，一般状态名都具有明确的意义，这符合人类语言习惯. 同时，也为了简化七元组以及 MCTK 代码中状态的表示，降低程序出错的可能性. 规范中状态的命名一般都带有特殊含义使得表达的意义更明确，使人一目了然，例如欲验证：无论何时只要 A 事件发生，则未来 B 事件必将发生，其规范可写为 $AG(A\_happen \rightarrow AF(B\_happen))$，A_happen、B_happen 均为状态名，其意义从字面上看很清晰，并且书写方便也便于 MCTK 代码的生成. 再例如主体发送信息 Msg 以前其状态为 Msg，发送该信息后状态变为 send_Msg，表示主体到达了“已经发送了 Msg 信息的状态”等等. 但是为了扩大可验证规范的种类，同时又要便于迁移七元组的正确生成，我们将部分采取变量的合取标示状态的方法. 具体状态表示方法将在 3.1 节给出.

由此，本文利用迁移七元组将 BPEL 语言刻画的组合 Web 服务与 MCTK 输入语言之间建立起联系. 实际上，迁移七元组从本质上说就是对状态转换图中的每一对有迁移关系的“状态对”的一种文字性刻画，因为传统工作对 BPEL 流程的验证皆需要构造状态转换图，而状态转换图在计算机中是不存在的，这需要大量的人工操作，之后再用形式化验证工具的输入语言对该状态转换图进行刻画，进而验证. 若要实现流程的自动验证，就需要自动生成能够反映状态转换关系的、文字性的、对流程进行刻画的规范. 并且构造该规范时还要考虑到后续工作，也就是是否便于用 MCTK 利用该规范进行形式化建模.

通过对 BPEL 活动执行语义进行分析，我们发现活动的执行过程就是对活动中所涉及的相关变量的一次赋值操作，BPEL 活动的属性显示了相关变量在活动执行完毕后得到的值以及其它一些必要的操作. 因此，在构造七元组时我们需要将 BPEL 语言活动中的对我们后续工作有价值的属性值抽取出来，以构造七元组. 于是，我们给出 BPEL 活动语义并结合该语义说明活动执行后相关七元组的产生.

2.2 BPEL 活动执行语义以及相关七元组的产生

本节给出 BPEL 活动的形式化语义，为了方便后续说明，我们首先定义几个函数：

cur(*state*)表示活动的初始状态，也就是活动刚开始执行时主体所处状态.

next(*state*)表示活动执行后主体所处状态.

value(*variable*)表示通道中的信息值.

c(*source*(*state*)，*dest*(*state*))表示与主体连接的通道名，通道中信息的传输改变主体状态.

dir(*channel*)表示通道中信息传送方向，即信息离开主体还是到达主体.

基本活动.

(1) receive

功能：该活动允许商业流程等待一条匹配消息的到来，当此消息到来时该活动结束.

格式：〈receive *partnerLink*＝*L*　*portType*＝*PT*　*operation*＝*Op*　*variable*＝*V*〉

迁移语义：

$$\delta_{receive}=\{T \mid P_i, P_{i+1}\in\Omega \wedge V\in O_e \wedge cur(state)=P_i \wedge next(state)=P_{i+1} \wedge c(P_i, P_{i+1})=PT \wedge value(PT)=V \wedge dir(PT)=IN\}$$

说明：

① $P_i, P_{i+1}\in\Omega$ 说明 P_i、P_{i+1}为状态空间 Ω 中的状态，即为主体流程执行过程中系统经历的状态；

② $V\in O_e$说明 V 是通道变量，因此，*variable* 属性所指变量值为本主体与外界主体的交互信息，其值存储于变量 V 中；

③ $cur(state)=P_i \wedge next(state)=P_{i+1}$ 指出这两个状态之间存在直接的迁移关系是由状态 P_i迁移到状态P_{i+1}；

④ $c(P_i, P_{i+1})=PT$ 指出端口类型 *portType* 属性指定了与本主体连接的用来发送或者接收消息的通道名；

⑤ $value(PT)=V$ 说明通道中的信息是变量 V 中存储的信息；

⑥ $dir(PT)=$ IN 说明该活动接收外界到来的

信息，信息的流向是进入主体. 同时也说明了 PT 通道是入通道.

所有条件的“与”操作构成了该活动形式化语义，我们将语义中各个元素抽取出，按照迁移七元组定义将它们按顺序载入，得到产生的七元组如下(迁移图如图 1 所示)：

$$T=(P_i, V, \varepsilon, \varepsilon, PT, \mathrm{IN}, P_{i+1}).$$

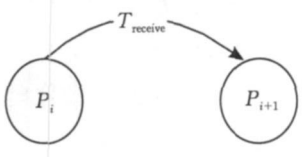


图 1 〈receive〉活动状态迁移示意图

(2) reply

功能：允许企业流程发送一条消息来响应诸如〈receive〉、〈onMessage〉和〈onEvent〉这样的“接收”信息的活动.

格式：〈reply $partnerlink=L$ $portType=PT$ $operation=Op$ $variable=V$〉.

迁移语义：

$$\delta_{reply}=\{T \mid P_i, P_{i+1}\in\Omega \wedge V\in O_e \wedge cur(state)=P_i \wedge next(state)=P_{i+1} \wedge c(P_i, P_{i+1})= PT \wedge value(PT)=V \wedge dir(PT)=\mathrm{OUT}\}.$$

说明：

① $P_i, P_{i+1}\in\Omega$ 说明 P_i、P_{i+1} 为状态空间 Ω 中的状态，即为主体流程执行过程中系统经历的状态；

② $V\in O_e$ 说明 V 是通道变量，因此，$variable$ 属性所指变量值为本主体与外界主体的交互信息，其值存储于变量 V 中；

③ $cur(state)=P_i \wedge next(state)=P_{i+1}$ 指出这两个状态之间存在直接的迁移关系是由状态 P_i 迁移到状态 P_{i+1}；

④ $c(P_i, P_{i+1})=PT$ 指出端口类型 $portType$ 属性指定了与本主体连接的用来发送或者接收消息的通道名；

⑤ $value(PT)=V$ 说明通道中的信息是变量 V 中存储的信息；

⑥ $dir(PT)=\mathrm{OUT}$ 说明该活动响应外界到来的信息，主体执行活动完毕将信息返回给调用方，说明了 PT 通道是出通道.

所有条件的“与”操作构成了该活动形式化语义，我们将语义中各个元素抽取出，按照迁移七元组定义将它们按顺序载入，得到产生的七元组如下(迁移图如图 2 所示)：

$$T=(P_i, V, \varepsilon, \varepsilon, PT, \mathrm{OUT}, P_{i+1}).$$

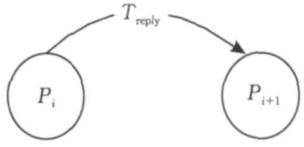


图 2 〈reply〉活动状态迁移示意图

(3) 同步 invoke

功能：允许流程在由合作伙伴提供的端口上调用该合作伙伴提供的服务，同步 invoke 为“请求-响应”式调用.

格式：〈invoke $partnerLink=L$ $portType=PT$ $operation=Op$ $inputVariable=IV$ $outputVariable=OV$〉.

迁移语义：

$$\delta_{syn\text{-}invoke}=\{T_1 \mid P_i, P_{i+1}\in\Omega \wedge IV\in O_e \wedge cur(state)=P_i \wedge next(state)=P_{i+1} \wedge c(P_i, P_{i+1})= PT \wedge value(PT)=IV \wedge dir(PT)=\mathrm{OUT}\} \wedge \{T_2 \mid P_{i+1}, P_{i+2}\in\Omega \wedge OV\in O_e \wedge cur(state)=P_{i+1} \wedge next(state)=P_{i+2} \wedge c(P_{i+1}, P_{i+2})= PT\_back \wedge value(PT\_back)= OV \wedge dir(PT)=\mathrm{IN}\}.$$

说明：

① 执行该活动状态迁移两次，语义中是两次迁移 $T1$, $T2$ 的“与”操作. $P_i, P_{i+1}, P_{i+2}\in\Omega$ 说明 P_i、P_{i+1}、P_{i+2} 均为主体流程执行过程中系统经历的状态. 由于是同步 invoke，因此，主体调用其它 Web 服务后等待响应信息的到来，其发送消息时通道变量的值发生变化，状态迁移一次，接收消息通道变量值再次发生变化，在上步基础上状态再迁移一次；

② $IV, OV\in O_e$ 说明 IV、OV 都是通道变量，因此，$inputVariable$、$outputVariable$ 属性所指变量值为本主体与外界主体的交互信息，其值分别存储于变量 IV 或者 OV 中. 由于是同步 invoke，主体调用其它 Web 服务后等待响应信息的到来，信息的交互是双向的，因此该活动具有两个用于存储交互信息的变量；

③ $cur(state)=P_i \wedge next(state)=P_{i+1}$ 指出这两个状态之间存在直接的迁移关系是由状态 P_i 迁移到状态 P_{i+1}；$cur(state)=P_{i+1} \wedge next(state)=P_{i+2}$ 说明主体在 P_{i+1} 状态中等待，直到接收到外界的响应消息后迁移到状态 P_{i+2}；

④ $T1$ 中 $c(P_i, P_{i+1})=PT$ 与 $T2$ 中 $c(P_{i+1}, P_{i+2})=PT\_back$ 分别指出相应端口类型 $portType$ 属性指定的，与本主体连接的用来发送或者接收消息的通道名. 需要说明的是通道名 PT 是 BPEL 流

程已经给出的，但为了实现通道机制，便于 MCTK 代码的自动生成，我们前面约定任一主体皆有两个单向通道与其相连，作为两个相反方向的信息传输通道.因此，凡具有双向信息交互的活动，若其通道名只有一个，则对另一个通道采用自命名方式.在这里，本文将另一通道命名为 PT_back；

⑤$Fvalue(PT)=IV$ 和 $value(PT\_back)=OV$ 说明通道中的信息是变量 IV 和 OV 中存储的信息；

⑥$dir(PT)=$OUT 和 $dir(PT)=$IN 说明 $T1$ 中通道是出通道，主体首先调用外界服务；之后说明 $T2$ 中通道是入通道，响应信息通过该通道进入主体.

单步迁移中所有条件的“与”操作构成了该步迁移的形式化语义.因为只有等到响应消息的到来，同步 invoke 活动才算结束，因此两步迁移采用“与”操作连接起来.我们将语义中各个元素抽出，按照迁移七元组定义将它们按顺序载入，得到产生的七元组如下(迁移图如图 3 所示)：

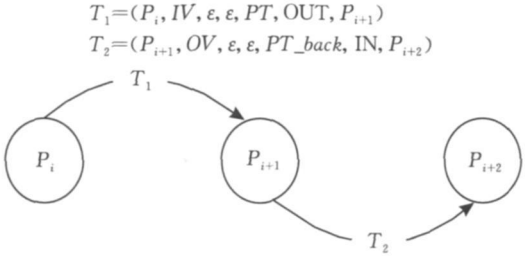


图 3　同步〈invoke〉活动状态迁移示意图

(4) 异步 invoke

功能：允许流程在由合作伙伴提供的端口上调用该合作伙伴提供的服务，异步 invoke 为单向请求式调用.

格式：〈invoke *partnerLink*＝*L*　*portType*＝*PT*　*operation*＝*Op*　*inputVariable*＝*IV*〉.

迁移语义：

$$\delta_{\text{syn-invoke}}=\{T \mid P_i, P_{i+1}\in\Omega \wedge IV\in O_e \wedge cur(state)=P_i \wedge next(state)=P_{i+1} \wedge c(P_i, P_{i+k})=PT \wedge value(PT)=IV \wedge dir(PT)=\text{OUT}\}.$$

说明：

①$P_i, P_{i+1}\in\Omega$ 说明 P_i，P_{i+1} 为状态空间 Ω 中的状态，即为主体流程执行过程中系统经历的状态；

②$IV\in O_e$ 说明 IV 是通道变量，因此，*variable* 属性所指变量值为本主体与外界主体的交互信息，其值存储于变量 IV 中；

③$cur(state)=P_i \wedge next(state)=P_{i+1}$ 指出这两个状态之间存在直接的迁移关系是由状态 P_i 迁移到状态 P_{i+1}；

④$c(P_i, P_{i+1})=PT$ 指出端口类型 *portType* 属性指定了与本主体连接的用来发送或者接收消息的通道名；

⑤$value(PT)=IV$ 说明通道中的信息是变量 IV 中存储的信息；

⑥$dir(PT)=$OUT 说明该活动发送调用信息到其它合作伙伴，信息的流向是进入被调主体，因此 PT 通道是出通道.

所有条件的“与”操作构成了该活动形式化语义，我们将语义中各个元素抽取出，按照迁移七元组定义将它们按顺序载入，得到产生的七元组如下(其迁移图与 reply 活动类似，如图 2 所示)：

$$T=(P_i, IV, \varepsilon, \varepsilon, PT, \text{OUT}, P_{i+1}).$$

结构活动.

(5) switch

功能：用于从一组选择中挑出一个分支，执行该分支包含的活动.

格式：〈switch〉

〈case *condition*＝C_1〉…〈/case〉

〈case *condition*＝C_2〉…〈/case〉

…

〈otherwise〉…〈/otherwise〉

〈/switch〉

迁移语义(假设该活动具有 N 分支)：

$$\delta_{\text{switch}}=\otimes_{1\le k\le n}\{T_k \mid P_i, P_{i+k}\in\Omega \wedge C_k\in O_g \wedge C_k=true \wedge cur(state)=P_i \wedge next(state)=P_{i+k}\}.$$

说明：

①$\otimes_{1\le k\le n}$ 表示互斥，即在待选的 N 分支中，只有一个分支(k 分支，$1\le k\le n$)被选中触发；

②$P_i, P_{i+k}\in\Omega$ 说明 P_i、P_{i+k} 为状态空间 Ω 中的状态，即为主体流程执行过程中系统经历的状态；

③$C_k\in O_g \wedge C_k=$ true 说明 C_k 是条件变量，也可以说是代表 *condition* 后条件表达式的断言变量，其值或为真或为假.$C_k=$true 表明当其为真时，该条件表达式为真，说明该分支被选中，流程执行该 C_k 所在〈case〉内的活动.当一个分支被选中后，在本次实例流程中其余分支将不再被触发；

④$cur(state)=P_i \wedge next(state)=P_{i+k}$ 指出这两个状态之间存在直接的迁移关系是由状态 P_i 迁移到状态 P_{i+k}.需要说明的是，对于 N 分支中的任一分支，其状态迁移的初始状态皆为 P_i，选择不同分支则流程由 P_i 状态迁移到不同状态 P_{i+k}，具体迁移到哪个状态由条件变量的真值决定.

所有条件的“与”操作构成了某一被触发迁移的形式化语义,我们将语义中的各个元素抽取出,按照迁移七元组定义将它们按顺序载入,得到产生的七元组如下(迁移图如图 4 所示).

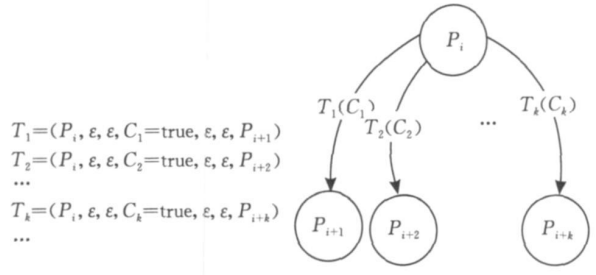


图 4 〈switch〉活动状态迁移示意图

(6) pick

功能:用于等待一系列可能消息中的一个到达本 Web 服务并执行相应活动或者等到超时事件发生.

格式:

〈pick〉

〈onmessage *variable*=m_1 porttype=PT_1〉…〈/onmessage〉

〈onmessage *variable*=m_2 porttype=PT_2〉…〈/onmessage〉

…

〈/pick〉

迁移语义:

$$\delta_{\text{pick}}=\otimes_{1\le k\le n}\{T_k \mid P_i, P_{i+k}\in\Omega\wedge m_i\in O_e\wedge cur(state)=P_i\wedge next(state)=P_{i+k}\wedge c(P_i,P_{i+k})=PT\wedge dir(PT)=\text{IN}\}.$$

说明:

①$\otimes_{1\le k\le n}$表示互斥,〈pick〉等待互斥信息的到来,当到来的消息与某个〈onmessage〉中的 *variable* 属性相匹配时,则相关活动被执行;

②$P_i, P_{i+k}\in\Omega$ 说明 P_i、P_{i+k} 均为状态空间 Ω 中的状态,即为主体流程执行过程中系统经历的状态;

③$cur(state)=P_i\wedge next(state)=P_{i+k}$ 指出这两个状态之间存在直接的迁移关系是由状态 P_i 迁移到状态 P_{i+k}.需要说明的是,对于 N 分支中的任一分支,其状态迁移的初始状态皆为 P_i,不同分支被触发则流程由 P_i 状态迁移到不同状态;

④$m_i\in O_e$ 说明 m_i 是通道变量,因此,*variable* 属性所指变量值为到达本主体的外界信息,也就是主体等待发生的事件,其值存储于变量 m_i 中;

⑤$c(P_i, P_{i+1})=PT$ 指出端口类型 *portType* 属性指定了与本主体连接的用来发送或者接收消息的通道名;

⑥说明主体等待事件的发生,信息的流向是进入该主体,因此 PT 通道是入通道.

所有条件的“与”操作构成了某一被触发迁移的形式化语义,我们将语义中各个元素抽取出,按照迁移七元组定义将它们按顺序载入,得到产生的七元组如下(迁移图如图 5 所示):

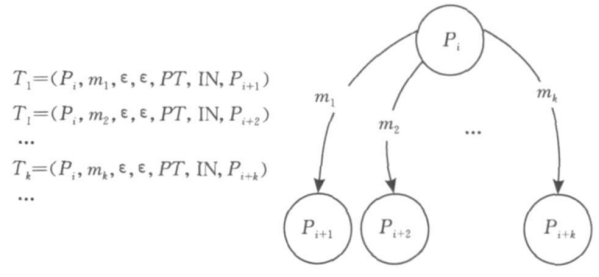


图 5 〈pick〉活动状态迁移示意图

(7) flow-source-activity

功能:用以实现程序的并发同步执行.

格式:〈flow〉

(迁移条件 *transitionCondition* 简写为 *tC*,假设有两个目标链接活动,其余类推)

〈-activity-〉

〈source *linkName*=L_1 *tC*=C_1〉

〈source *linkName*=L_2 *tC*=C_2〉

〈/--activity--〉

迁移语义:

$$\delta_{\text{flow-s-a}}=\{T_1 \mid P_i, P_{i+1}\in\Omega\wedge\cdots\wedge cur(state)=P_i\wedge next(state)=P_{i+1}\}\wedge\{T_2 \mid P_{i+1}, P_{i+2}\in\Omega\wedge L_1, L_2\in O_l\wedge C_1, C_2\in O_g\wedge(C_1\vee C_2)\in O_g\wedge cur(state)=P_{i+1}\wedge next(state)=P_{i+2}\}.$$

说明:

①该活动迁移两次,T_1 迁移触发后再触发 T_2 迁移,语义中是两次迁移 T_1, T_2 的“与”操作,由于 source 容器存在,说明此活动为多个活动的源链接.因此活动先执行,引起状态迁移 T_1, T_1 执行完毕将迁移条件 *tC* 分别赋值为 C_1 和 C_2,由于 C_1, C_2 都是条件变量,因此,由于 *tC* 的赋值导致变量值改变状态再次发生迁移,于是 T_2 被触发;

②$P_i, P_{i+1}, P_{i+2}\in\Omega$ 说明 P_i、P_{i+1}、P_{i+2} 均为主体流程执行过程中系统经历的状态;

③T_1 中 $cur(state)=P_i\wedge next(state)=P_{i+1}$ 指出由于活动执行导致状态迁移,由状态 P_i 迁移到状

态 P_{i+1}；

④ T_2 中 $L_1, L_2 \in O_l \wedge C_1, C_2 \in O_g \wedge C_1 \vee C_2 \in O_g$ 指出 L_1, L_2 为链接变量，当链接存在时，这些变量为真；C_1, C_2 为条件变量，若 $C_1 \vee C_2$ 为真，则该活动作为其它活动的源链接是有效的. 显然，$C_1 \vee C_2 \in O_g$；

⑤ T_2 中 $cur(state)=P_{i+1} \wedge next(state)=P_{i+2}$ 指出由于 T_1 执行完毕给 tC 赋值导致条件变量 O_g 改变，状态再次发生迁移，于是 T_2 被触发，系统状态由 T_1 执行末状态 P_{i+1} 迁移到状态 P_{i+2}.

所有条件的“与”操作构成了单步迁移的形式化语义，由于两步迁移具有先后顺序，我们用“与”操作将其联系起来. 于是，将语义中各个元素抽取出，按照迁移七元组定义将它们按顺序载入，得到产生的七元组如下(迁移图如图 6 所示)：

$$T_1 = (P_i, \cdots, P_{i+1}),$$
$$T_2 = (P_{i+1}, \varepsilon, \varepsilon, C_1 \vee C_2, \varepsilon, \varepsilon, P_{i+2}).$$

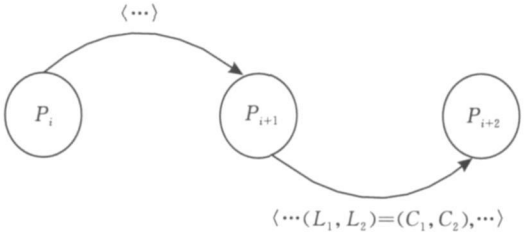


图 6　flow-source-activity 状态迁移示意图

(8) flow-target-activity

格式：⟨flow⟩

(迁移条件 *joinCondition*，简写为 *jC*)

⟨--activity--⟩

⟨target *linkName*=L_1　*jC*=Q_1⟩

⟨target *linkName*=L_2　*jC*=Q_2⟩

…

⟨/−activity−⟩

迁移语义：

$$\delta_{flow\text{-}t\text{-}a}=\{T_1 \mid P_i, P_{i+1} \in \Omega \wedge L_1, L_2 \in O_l \wedge Q_1, Q_2 \in O_g \wedge ((L_1 \wedge Q_1) \vee (L_2 \wedge Q_2)) \in O_g \wedge cur(state)=P_i \wedge next(state)=P_{i+1}\} \wedge \{T_2 \mid P_{i+1}, P_{i+2} \in \Omega \wedge \cdots \wedge cur(state)=P_{i+1} \wedge next(state)=P_{i+2}\}.$$

说明：

①该活动迁移两次，T_1 迁移触发后再触发 T_2 迁移，语义中是两次迁移 T_1，T_2 的“与”操作，由于 target 容器存在，说明此活动为多个活动的目标链接. 因此先判断该目标活动被触发需满足条件的真值，即 jC 被赋值，迁移 T_1 被触发；若条件为真则活动被执行，状态再次发生迁移，T_2 被触发；

② $P_i, P_{i+1}, P_{i+2} \in \Omega$ 说明 P_i、P_{i+1}、P_{i+2} 均为主体流程执行过程中系统经历的状态；

③ T_2 中 $L_1, L_2 \in O_l \wedge Q_1, Q_2 \in O_g \wedge ((L_1 \wedge Q_1) \vee (L_2 \wedge Q_2)) \in O_g$ 指出 L_1，L_2 为链接变量，L_1，L_2 链接状态的确定是链接发生的前提条件，当这两个链接有效时，这些变量为真. 于是，在链接变量有效的前提下，判断 Q_1，Q_2 的真值，若为真，则控制流转移到本活动中来，因此，L_1 与 Q_1 必须同时为真，链接才能转移，对其它情况类似. 显然，$((L_1 \wedge Q_1) \vee (L_2 \wedge Q_2)) \in O_g$；

④ T_1 中 $cur(state)=P_i \wedge next(state)=P_{i+1}$ 指出由于给 jC 触发条件赋值并判断，条件变量 O_g 改变，系统状态由 P_i 迁移到状态 P_{i+1}；

⑤ T_2 中 $cur(state)=P_{i+1} \wedge next(state)=P_{i+2}$ 指出由于目标活动执行条件满足，于是活动执行导致状态迁移，由状态 P_{i+1} 迁移到状态 P_{i+2}.

所有条件的“与”操作构成了单步迁移的形式化语义，由于两步迁移具有先后顺序，我们用“与”操作将其联系起来. 于是，将语义中各个元素抽取出，按照迁移七元组定义将它们按顺序载入，得到产生的七元组如下(迁移图如图 7 所示)：

$$T_1 = (P_i, \varepsilon, \varepsilon, (L_1 \wedge Q_1) \vee (L_2 \wedge Q_2), \varepsilon, \varepsilon, P_{i+1}),$$
$$T_2 = (P_{i+1}, \cdots, P_{i+2}).$$

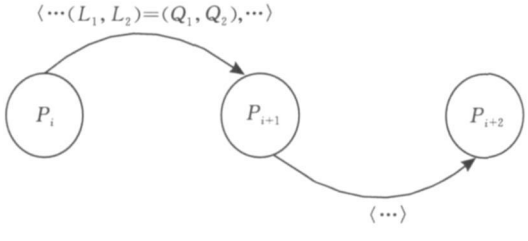


图 7　flow-target 状态迁移示意图

WS-BPEL 2.0 支持的活动种类繁多，因此实际的建模工作复杂且繁琐，以上只给出了部分重要活动语义及相关说明，在后续研究中我们将不断扩展对其它活动的建模方法，因此对于这些活动，本文简化处理如下：⟨assign⟩活动代表变量的赋值操作，⟨exit⟩活动用于立即终止流程，⟨wait⟩活动用于实现流程等待某个时间点的到来，这些活动的语义简单，这里不再详述；⟨throw⟩、⟨rethrow⟩、⟨compensate⟩是作用域⟨scope⟩的实现机制，分别用来抛出错误和补偿处理，它们用来处理流程中的非设计性失误这类错误，我们用⟨empty⟩活动代替它们，从而在建模过程中忽略它们；⟨if⟩活动用来实现双分支选择，可用⟨switch⟩活动替换⟨if⟩活动；对于⟨validate⟩和⟨extensionActivity⟩活动，前者用于验证存储在变量

中的 XML 信息的有效性，后者用于对 BPEL 语言进行扩展，本文暂不考虑这两个活动.另外，〈repeatUntil〉、〈forEach〉用于处理循环，可用〈while〉活动代替.〈while〉活动初步的转化过程并不复杂，但是我们在转化算法实现和测试时发现循环活动往往会生成大量的系统状态，使得转化出的 OBDD 形式模型过大，导致在可接受的时限内无法得到验证结果，因此严谨起见，本文暂不讨论〈while〉等循环活动，我们将在后续工作中深入研究这些活动的优化转化算法.

模型检测的状态爆炸问题始终是阻碍模型检测技术投入实用的一大瓶颈，MCTK 作为认知模型检测工具也不例外.虽然我们在 MCTK 中采用 OBDD 实现符号化建模和验证算法，已大大减少形式模型和验证所需的内存空间，但是 OBDD 的规模可能会因为 OBDD 变量的增多而迅速扩大，因此在 MCTK 输入语言中定义的变量要尽量减少.而且 BPEL 语言本身的复杂性可能令 MCTK 生成复杂的状态迁移关系，也可导致其 OBDD 表示的规模迅速增大.因此，本文在转换过程中忽略掉上述活动中某些与验证无关的属性，从而简化转化过程，减小最终生成的 MCTK 形式模型的规模.此外，我们还将采用3.1节介绍的状态命名规则，避免多变量标示带来的书写以及理解上的麻烦，同时将活动与七元组建立映射关系，使得活动的执行转化为与其对应的七元组的产生，简化了活动执行的复杂性，使得后续算法更好把握流程的执行带来的状态变化的影响.

我们看到 BPEL 结构化活动可以用来组织基本活动，使得基本活动在结构化活动的“安排”下按照某种特定顺序执行，从而使得各 Web 服务协作完成即定任务.前面我们已经分析了基本活动向迁移七元组的转换过程，为了实现流程自动化验证我们必须找到一种算法，使得各基本活动能够按照其所在结构活动的结构框架下按正确顺序执行产生相应迁移七元组.因此，这样的算法实际上是将结构活动的性质用算法的形式表示出来，期间遇到基本活动就按照上节介绍的映射关系将其映射为相应七元组，实现 BPEL 流程向七元组集合的转化.基于这一思想，我们提出将 BPEL 流程转化为迁移七元组集合的算法.

3 BPEL 语言的自动化模型检测方法

由前述可以看出，由于 BPEL 语言自身的复杂性，如何建立该语言到某种规范的统一映射以及如何自动利用模型检测工具对此规范进行描述是实现流程自动化验证需克服的两个技术难点.由于 BPEL 语言与模型检测工具 MCTK 输入语言之间无直接联系，传统工作主要是将 BPEL 语言转化为状态转换图，之后对状态转换图进行描述.根据上节提出的迁移七元组的语法及语义，BPEL 代码中每一个活动的执行都将产生一个相应的七元组，此七元组包括了主体交互的所有信息.因此，如何产生正确的迁移七元组以及如何方便地利用一组迁移七元组写出 MCTK 代码，进而验证组合服务，就是下文的主要内容.

3.1 将 BPEL 模型转化为迁移七元组

(1)状态命名规则

迁移七元组明确指出了状态之间的迁移关系，以及状态迁移后各相关变量值的变化.因此，状态的命名对于迁移七元组的自动生成十分重要.在前面章节我们已经说明，完全用变量的合取标示状态对于实现自动化检测并不适用，但是为了实现对更多的涉及数据流的规范进行验证，本文部分采取用变量的合取标示状态的方法，但与传统方法又有很大不同，具体状态命名法则如下：

① 对于由通道变量 O_e 值的改变引起的状态迁移，状态命名采用“send_消息”表示主体“发送了某信息”状态，用“get_消息”表示主体“接收了某信息”状态，例如主体的初始状态为 sleep，当其发送了信息 Msg 以后，状态从 sleep 迁移到了 send_Msg，表示状态从“睡眠”迁移到“发送了 Msg”状态.

② 对于由链接变量 O_l、条件变量 O_g 值的改变引起的状态迁移，状态命名采用“源状态 $\wedge O_l$”或者“源状态 $\wedge O_g$”原则，“$\wedge$”号后面的变量是在源状态中，其值在活动发生后有可能发生变化的那些 O_l 或者 O_g 类型的变量.例如：主体在 *get_request* 状态下，若活动执行将对该状态下的条件变量 tC 进行赋值操作，那么执行活动后系统状态就由 *get_request* 迁移到状态 *get_request* $\wedge tC$.若主体在 *get_request* 状态下，活动执行将对其中条件变量 tC_1 或者 tC_2 进行赋值操作，那么活动执行后系统状态就由 *get_request* 状态迁移到 *get_request* $\wedge (tC_1 \vee tC_2)$ 状态.

(2)转换算法

〈sequence〉和〈flow〉是 BPEL 语言中最基本的两种控制结构，前者用来处理顺序流程，后者用来处理并发流程.对于顺序流程，活动按照出现的先后顺

序执行，〈sequence〉内可以嵌套基本活动与结构活动〈switch〉或者〈pick〉等，此时，如果将该流程用状态转换图表示出来，则该图是一个多分支的状态转换图，可以用 MCTK 这样验证分支时态逻辑的模型检测工具对其进行刻画，进而验证. 对于嵌套在〈sequence〉中的基本活动，我们可以按照顺序的方式产生与各个活动对应的迁移七元组，然后用 MCTK 对产生的迁移七元组集合进行描述.

下面给出〈sequence〉结构框架下的业务流程转化为迁移七元组的算法，由于在设计七元组时也同时考虑到了 MCTK 输入语言的特点，因此通过该算法生成的七元组可以用来进一步自动生成 MCTK 的输入代码，使得流程验证过程的自动化程度大大提高.

算法 **B2T.**　〈sequence〉→迁移七元组.

```
B2T(ACT(S), STAT(S))
输入: 描述 Web 服务的 BPEL 源码
输出: 迁移七元组集合
1. Init Act. Top=〈sequence〉; Init Stat. Top=sleep;
       cur(state)= sleep;
2. scan from current activity〈…Σ…〉to the last activity;
3. for each element 〈…Σ…〉 do
4.  {/ *将活动及活动执行后的状态入栈 */
5.    Act. Top++ &Act. Push 〈…Σ…〉;
6.    ∞〈…Σ…〉;
7.    Stat. Top++ &Stat. Push(next(state));
8.    cur(state)= next(state);
9.    If(〈…/Σ…〉≡〈/ case〉 OR 〈otherwise〉)
10.    {/ *活动栈及状态栈同步弹栈直到 switch */
11.      do
12.      {
13.        Act. Pop &Act. Top--;
14.        Stat. Pop &Stat. Top--;
15.      }while(Act. Top !≡〈case〉 OR 〈switch〉);
16.      cur(state)= Stat. Top;
17.    }
18.    If(〈…/Σ…〉≡〈/ onmessage〉)
19.    {/ *活动栈及状态栈同步弹栈直到 pick */
20.      do
21.      {
22.        Act. Pop &Act. Top--;
23.        Stat. Pop &Stat. Top--;
24.      }while(Act. Top !≡〈pick〉);
25.      cur(state)= Stat. Top;
26.    }
27. }
```

算法说明：“≡”符号代表纯文本的匹配，当“≡”号左右的文本内容完全一样时，该关系成立. 算法设置两个栈用以实现对结构活动的结构框架的描述，一个状态栈 Stat 用来存放主体在执行过程中所经历的所有状态；一个活动栈 Act 用来存放流程执行过程中所经历的所有活动. 我们用这两个栈来处理流程中活动与状态之间的关系. 〈…Σ…〉表示任一活动的开始标记，〈…/Σ…〉表示活动的结束标记，例如流程遇见〈receive〉活动则表示该活动准备开始执行，遇见〈/ receive〉则表示该活动执行完毕. 需要说明的是〈…/Σ…〉不进栈，因此扫描时跳过〈…/Σ…〉. 我们用 ∞表示“执行活动”，所谓执行就是流程执行该活动指定的操作，因此在本算法中 ∞〈…Σ…〉就是产生与活动相对应的迁移七元组. *cur*(*state*)用来存放下一个即将执行的活动的初始状态，显然初始状态的确定非常重要，对于顺序执行的活动 *A*、*B*，*B* 活动的初始状态即为 *A* 活动的终结状态. 而对于带有分支的活动，例如〈switch〉活动，假设流程刚进入〈switch〉活动时的状态为 $state_0$，则该活动中任一分支的初始状态皆为 $state'_0$，如图 4 所示：设 $P_i = state_0$，$P_{i+1} = state_1$，$P_{i+2} = state_2$，…则流程执行完第 1 个〈case〉包含的活动后状态由 $state_0$ 迁移到 $state_1$，执行完第 2 个〈case〉包含的活动后状态由 $state_0$ 迁移到 $state_2$，….起初，活动栈被初始化为〈sequence〉，状态栈与当前状态 *cur*(*state*)都初始化为 sleep. 算法步 2 指明从当前活动开始向后执行，由于是顺序结构，对于已经执行完毕的活动，流程不可能返回执行，只能继续向下执行直到〈sequence〉结构中的最后一个活动为止；算法步 3 开始循环，从〈receive〉活动开始(〈receive〉用来接收外界信息使流程开始执行，一般它都是流程的第 1 个活动)向下扫描；算法步 5 ~8 将扫描到的基本活动进栈，注意这里说的是活动的名称；步 7 将执行活动完毕产生的末状态 *next*(*state*)进状态栈，之所以保存每个活动执行完毕的末状态原因有两点：其一，该状态是下一个活动的初始状态；其二，若下一活动为带有分支的结构活动，则这些分支的初始状态皆为此进栈状态. 需要注意的是，本算法自始至终要保证活动栈与状态栈的同步进栈，也就是这两个栈的栈顶指针要保持同步增加或者同步减少. 这样做是为了保证两个栈的栈顶元素具有高度相关性，即状态栈的栈顶保存的是活动栈栈顶活动执行完毕的末状态；算法步 8 将进栈状态赋值给 *cur*(*state*)，说明下一活动的初始状态，该算法采用双栈配合使用方法来描述结构活动的相关结构性质，但具体的活动执行时的状态衔接点还需要额外变量保存，这将在实例中看到；步 9 ~17 说明若扫描到〈otherwise〉或者〈case〉，则

表明〈switch〉活动已经进栈，此时需要找到流程刚进入〈switch〉时的状态，因为无论是〈otherwise〉还是〈case〉代表的都是〈switch〉中的分支，这些分支拥有同一个迁移原状态.因此，将状态栈与活动栈同步退栈，当活动栈栈顶指针退栈到 $Act.Top ==$ 〈switch〉时，说明状态栈栈顶指针指向的即为流程刚进入〈switch〉时的状态.之后将 $Stat.Top$ 所指状态赋给 $cur(state)$，用做分支活动的原状态；步 18～25 用来处理〈pick〉活动，其原理与〈switch〉相同，这里不再赘述.由于算法需要从上到下扫描所有活动，当活动数为 n 时，算法的时间复杂度为 $O(n)$.

〈flow〉活动用以处理并发程序的执行，该活动根据迁移条件的不同，业务流程将选择不同的分支路径执行，被执行的活动又可以按照迁移条件选择下一个需要执行的活动….如图 8 所示，第一条链路中链接变量 L_1 的源迁移为活动 Act_1，目标迁移为活动 Act_2，L_2 的源迁移为 Act_2，目标迁移为 Act_3，…，L_k 的源迁移为 Act_k，目标迁移为 Act_{k+1}.而在〈flow〉结构中，一个活动往往是多个目标活动的源迁移，例如链接变量 L'_1 的源迁移依旧为 Act_1，但目标迁移活动变为 Act'_2，链接变量 L'_2 的源迁移活动为 Act'_2，目标活动却为 Act_3 ….

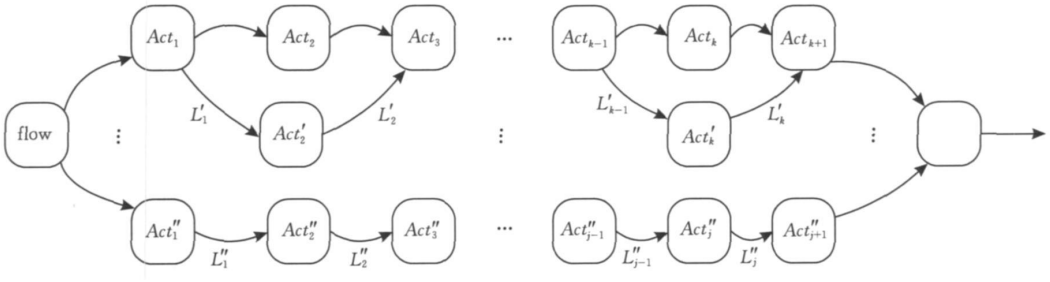


图 8 〈flow〉抽象流程示意图

第 2.2 节给出了 flow-source-activity 和 flow-target-activity 迁移语义，在 flow-source-activity 中，activity 的执行导致 T_1 迁移的触发，执行完毕后给迁移条件 *transitionCondition* 赋值，条件变量 O_g 改变，T_2 迁移触发，系统状态再次发生变化.因此 T_2 迁移的末状态即表示 *transitionCondition* 被赋值后的状态；同理，在 flow-target-activity 中，首先给 *joinCondition* 赋值，O_g 改变，导致 T'_1 触发，状态迁移，之后执行相应活动 activity，T'_2 触发，状态再次发生改变.

B2T 算法给出了〈sequence〉结构框架下 BPEL 流程向迁移七元组的转换方法，下面我们基于 2.2 节给出的活动语义，提出将〈flow〉转化为迁移七元组的转换算法 F2T.为此首先提出活动图的概念，所谓活动图即是由活动作为顶点，迁移作为边的有向图，对于迁移 L，若其迁移源活动为 A，迁移目标活动为 B，则用 $A \rightarrow B$ 代表流程执行完活动 A 转到活动 B 执行.

我们将研究对象简记为 Act〈flow〉，Act〈flow〉本身是简单的〈flow〉活动，所谓简单是指其内部不再嵌套〈flow〉.我们用 T_1、T_2、T_3、…表示顺序执行的迁移序列，其中 T_2 的初状态为 T_1 的末状态，T_3 的初状态为 T_2 的末状态….

算法 **F2T**. 〈flow〉→迁移七元组.

输入：简单〈flow〉结构下的组合业务流程
输出：该组合流程的迁移七元组集合
/ *phase1：构建活动图 */
1. 查找 BPEL 源码中〈link name〉属性值，找到所有迁移，为每个迁移设置集合：$Link_i$｛*sourceAct*，*destAct*｝
2. for each *Link*
3. ｛
4. 找到 $Link_i$ 链接的源活动和目标活动，放入 $Link_i$｛*sourceAct*，*destAct*｝，这可通过查找 $Link_i$ 所在的〈sources〉或者〈targets〉所属的活动实现；
5. 建立单步链接图：$Link_i.sourceAct \xrightarrow{Link_i} Link_i.destAct$；
6. If($Link_i.destAct == Link_j.sourceAct$)
7. 建立单步链接图：$Link_i.destAct \xrightarrow{Link_j} Link_j.destAct$；
8. ｝
/ *phase2：创建状态转换图 */
9. 流程进入〈flow〉，保存此时状态
10. 查找无入链接的活动，从这些活动开始：
11. ｛
12. 对于活动图中具有 $Link_i.sourceAct \xrightarrow{Link_i} Link_i.destAct$ 形式的链接：
13. ｛
14. 根据 flow-source-activity 语义，执行 $Link_i$ 的

源活动 activity：产生相应迁移；

15. 根据 flow-target-activity 语义，以步 14 末状态作为该步初始状态，执行 $Link_i$ 的目标活动 activity：产生相应迁移；
16. If($Link_i.destAct == Link_j.sourceAct$)
17. {
18. 根据 flow-source-activity 语义，执行 $Link_j$ 的源活动：以步 15 末状态作为该步初状态，执行链接条件 *transitionCondition* 的赋值操作，产生相应迁移；
19. 根据 flow-target-activity 语义，以步 18 末状态作为该步初状态，执行 $Link_j$ 的目标活动：产生相应迁移；
20. }
21. }
22. }直到没有新的迁移生成；

算法 phase1 阶段说明了活动图的产生过程，phase2 根据 2.2 节给出的活动语义，执行活动图的每个顶点活动，并转化为相应迁移七元组. 步 10～15 说明若该活动是某迁移源活动，则先执行活动，再给条件变量 *transitionCondition* 赋值；若该活动是某迁移目标活动，则先给条件变量 *joinCondition* 赋值，再执行活动；步 16～21 说明若此活动即是某迁移的目标活动，又是另一迁移的源活动，则当其作为目标活动执行完毕后，给下一迁移的链接条件 *transitionCondition* 赋值，之后执行下一迁移的目标活动.

算法的研究对象是针对简单〈flow〉结构下的组合业务流程，很多时候活动之间是相互嵌套的. 例如：〈flow〉活动内可以嵌套〈sequence〉，〈sequence〉也可以嵌套〈flow〉，同时〈flow〉可以嵌套〈flow〉，对于这些情况，其控制流程有时候是非常复杂的，控制流常常在多个 Web 服务之间来回调用，但由于流程进入〈flow〉活动后，将返回〈flow〉内活动执行的统一结果，因此可以将〈flow〉看做一个具有双向交互性质的基本活动，于是：

对于〈sequence〉嵌套〈flow〉情况可以看作〈sequence〉中包含了一个特殊的基本活动，活动按照顺序流程执行，当执行到〈flow〉时再进行相应处理即可，而〈flow〉的输出状态则是其在〈sequence〉结构中下一活动的输入状态. 该结构如图 9 所示.

对于〈flow〉中嵌套〈sequence〉情况，由于〈flow〉根据不同的迁移条件选择不同的分支执行，因此，可以将每个分支看作〈sequence〉流程，各〈sequence〉流程具有相同的初始状态即为流程刚进入〈flow〉时的状态. 可以对于每个分支调用 B2T 算法产生状态迁移关系，最后验证. 该结构如图 10 所示.

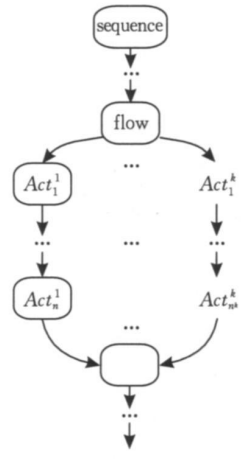


图 9　〈sequence〉中嵌套〈flow〉

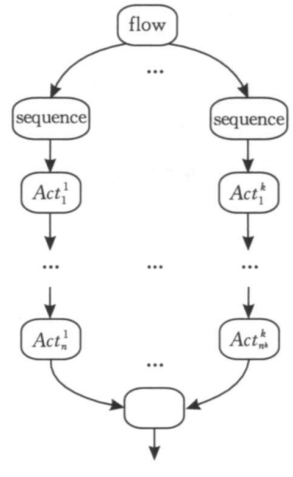


图 10　〈flow〉中嵌套〈sequence〉

对于〈flow〉中嵌套〈flow〉情况，我们对最内层〈flow〉调用 F2T 算法，生成相应状态转换图，之后将内层〈flow〉看作基本活动，对外层〈flow〉递归调用 F2T 算法，生成更大范围的状态转换图…. 该结构如图 11 所示.

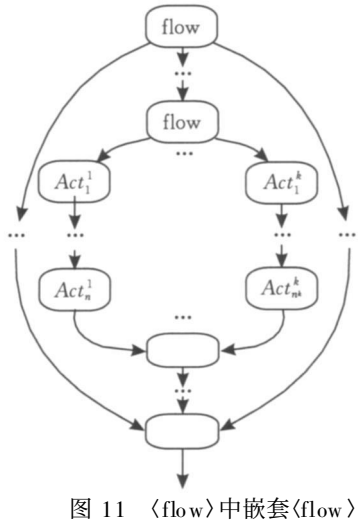


图 11　〈flow〉中嵌套〈flow〉

通过 F2T 算法，我们将具有〈flow〉结构的 BPEL 流程转化为了迁移七元组集合，实际上迁移七元组的提出就是为了描述状态转换关系，因此，该算法的最终目的就是将流程转化为状态转换图，这是进行形式化验证的步 1，也是关键一步，能否自动生成状态转换图是关系到能否实现 BPEL 流程自动化验证的核心环节，有了状态转换图更能方便地利用模型检测工具对流程进行刻画. F2T 算法相比 B2T 算法，加入了对数据流的支持，使得对流程的建模不仅仅是考虑到控制流的转移关系，更涉及到数据变量的变化情况，增加了状态空间中的状态数量，可验证的规范数量也大大增加，更能确保软件的正确性.

B2T 与 F2T 算法主要用来生成迁移七元组，这是工作的第 1 步，而我们的最终目的是验证组合服务流程，因此，下面还必须用合适的模型检测工具对这些转换而来的七元组进行描述. 本文采用自主开发的多主体系统模型检测工具 MCTK 作为底层验证平台.

3.2 MCTK 与主体声明

通过上节算法得到的迁移七元组集合必须自动转化为多主体系统模型检测工具 MCTK 的输入语言，才能实现对组合业务流程的自动化模型检测. 因此，本节简要介绍 MCTK，具体理论和系统开发方法请参考文献[11, 13].

MCTK 是我们开发的多主体系统符号化模型检测工具，在 Linux 环境下用 C 语言在经典模型检测工具 NuSMV 2.1.2 基础上扩展开发而来，采用 Fabio Somenzi 的 CUDD OBDD 软件包开发 MCTK 的符号化模型检测算法. MCTK 的输入语言是我们提出的多主体有限状态机的符号化描述，所刻画的多主体系统由一个环境和多个主体组成，主体间可进行交互. 每一主体可以观察环境的部分或全部，不同的主体对环境的观察可以重叠，即它们可以同时观察环境中的同一部分. 目前 MCTK 支持的时态认知逻辑 ECKLn 是 Halpern 和 Vardi 提出的时态知识逻辑 CKLn[19] 的一种扩展，ECKLn 的语法定义如下：

$$f ::= true \mid p \mid \neg f \mid f \wedge f \mid Xf \mid fUf \mid Ef \mid K_i f \mid C_\Gamma f.$$

显然，ECKLn 语言是在线性时态逻辑 LTL 语言中扩展路径量词 E（存在路径）、知识算子 K_i（对于每个智能体 i）以及公共知识算子 C_Γ（对于一组智能体 Γ）得到的，因此 ECKLn 的时态部分支持计算树逻辑 CTL*. 我们已在引言中通过一个时态认知规范展示其含义. 篇幅所限，相关形式模型和语义请参考文献[11, 13].

MCTK 输入语言是在时态模型检测工具 NuSMV 2.1.2 输入语言上扩展定义的，因此 MCTK 利用 main()模块定义环境变量和行为，并利用 MODULE 定义主体的行为，支持每一主体的可观察变量集合的定义和声明. 值得一提的是 MCTK 不必明确地表示主体的行为，仅需定义系统状态迁移关系即可，这一建模方法即可缩减状态空间，又有利于灵活定义环境和主体的行为所导致的状态迁移.

(1) MCTK 的输入语言

MCTK 的输入语言是在 NuSMV 基础上扩展而来，可以参考 NuSMV 手册来查阅其余相关细节，下面给出 MCTK 输入语言的语法结构：

```
MCTK_program ::=
    EnvDef                  ;; 环境定义
    AgentDefList            ;; 主体模块定义
EnvDef ::=
    "MODULE" "main" "(" ")"
    VarDef                  ;; 环境变量
    AgentList               ;; 主体声明
    [VarAssignDef]          ;; 环境变量赋值
    [VarInitDef]            ;; 环境变量初始化
    [ECKLnSpecDef]          ;; 规范定义
    …
AgentList ::=
    atom ":" "AgentType" ";" | AgentList atom ":" "AgentType" ";"
AgentType ::=               ;; 主体 atom 使用的模块名
    atom [ "(" AParaList ")" ]
    | "array" number ".." number "of" AgentType
AParaList ::=               ;; 实际参数列表
    simple_f | AParaList "," simple_f
AgentDefList ::= AgentDef | AgentDefList AgentDef
AgentDef ::=                ;; 主体模块定义
    "MODULE" atom [ "(" FParaList ")" ]
    LvarDef                 ;; 局部变量
    ActDef                  ;; 动作变量
    OvarAssignDef           ;; 可观察变量的赋值
    ProtDef                 ;; 动作变量的赋值
    …
FParaList ::=               ;; 形式参数列表
    ["Observable"] atom     ;; 如果指定 Observable，则表示
                            ;; 形参 atom 可观察
    | FParaList "," ["Observable"] atom
ActDef ::=                  ;; 动作变量 atom 的定义
    atom ":" "{" AtomList "}" ";"
AtomList ::= atom | AtomList "," atom
atom ::= [A-Za-z][A-Za-z0-9_\#-]*
```

主体运行环境我们用 main 模块定义，VarDef 定义了一些环境变量，这些变量可用来表示环境的状态，当中的某些成员可以作为主体相互通信的共享变量.

(2) 主体声明

我们对每一主体建立模型，下面介绍有关建模过程中涉及到的一些重要概念：

形式参数. 一个主体模型的形式化参数在“FparaList”中定义，其值将被实参替代，主体模型中的形参对于主体来说是可观察的，一个主体可以观察环境的部分变量，甚至其它主体的一部分变量. MCTK 规定可观察变量的前缀必须是“ObsPrm_”，表明该变量是可观察的.

可观察变量. 一个主体的可观察变量的集合 O_i 主要由该主体的局部变量以及可观察的实际参数组成，在我们实验中，主体的局部状态由其可观察变量指定. 如果在 LvarDef 中定义的变量是一个模块实例，则在动作变量集合 ActDef 中定义的变量也是主体的可观察变量.

可观察变量赋值. 主体中关于可观察变量的评价函数实际上用来描述主体的可观察变量的值是如何改变的，是一个从主体部分状态集合到另一个部分状态集合的映射函数，迁移函数在 OVarInitDef 和 OvarTransDef 中定义.

3.3　将七元组转化为 MCTK 的输入语言

上节简单介绍了 MCTK 及其输入语言，由此可见 MCTK 实际上是在 NuSMV 基础上引入了可观察变量的概念，两者形参的唯一不同之处就是 MCTK 需要在形参前面加上前缀“ObsPrm_”，表示该形参是可观察的. 对于模型的描述 MCTK 与 NuSMV 基本相同，需要详细刻画状态转换过程，主要包括某状态在一定条件发生情况下迁移到另一状态. 迁移七元组详细给出了状态转换关系，其第 1 项指明了迁移的初始状态，最后一项指明了迁移的末状态，第 2、3、4 项则分别指明了导致迁移发生的状态变量的变化情况，使得 MCTK 可以方便地利用这些信息对状态迁移过程进行描述.

在 2.1 节我们引入“通道”及“通道变量”的概念，“通道”实际是一种消息机制，是为了方便实现对服务之间交互过程的建模而引入的“通道”概念，它实际是不存在的. 但是，组合 Web 服务的各原子服务必须相互“联系“起来，否则单个主体无法与外界交流，如果反映在状态转换图中，则多个主体的状态转换图相互独立，不存在“联系”，也就无法实现组合流程的模型检测. 因此，从直观上看主体通过所谓“通道”建立联系，但从本质上说主体实际是通过“通道变量”建立起的“联系”，因为“通道变量”是具有直接交互关系的两个主体之间的共享变量，相互“联系”的主体通过此变量实现信息的发送与接收. 为了方便对服务之间信息交互过程进行建模，实现将多个服务“联系”起来的目的，我们将服务之间的交互过程看成是“通道变量”通过“通道”在交互的 Web 服务之间传输的过程. 为此，我们必须对“通道”这种消息机制进行建模.

在用 MCTK 对“通道”建模时，将“通道”看作与 Web 服务一样的实体，只不过主体的状态通过主体的变量标识，而“通道”的状态则由“通道”中当时的信息标识.

对主体状态的迁移我们是这样刻画的：在活动执行前，主体处于状态 $state_0$，信息 Msg 到来，执行活动，此时该主体入通道 PT 的值：$value(PT)=Msg$，主体执行活动完毕迁移到 $state_1$. 假设该主体名为 $AgentExa$，则上述迁移可形式化表述为

$$(AgentExa.state=state_0\ \&value(PT)=Msg)\Rightarrow next(AgentExa.state)=state_1.$$

当我们将“通道”看作与 Web 服务一样的实体时，其描述思想与状态迁移的描述思想是类似的. 需要记录此时刻主体状态以及上一时刻导致主体变为此状态的信息及其经历的通道名. 例如：主体 $AgentExa$ 在接收到通道 $PortTypeIn$ 传来的信息 Msg 后，状态变为 get_Msg，之后又通过通道 $PortTypeOut$ 发送信息 Msg_1，则我们可以将上述迁移形式化描述为

$$(state=get\_Msg)\ \&(PortTypeIn.state=Msg)\Rightarrow next(PortTypeOut.state)=Msg_1.$$

这样，我们通过通道将各个主体连接在了一起，确切地说是通过通道中交互的信息将它们联系起来. 当某个主体发送了某消息后，另一主体则通过相同“通道”接收了该消息，使得两个主体在此时有了联系，更具体地说就是将具体的 Web 服务方联系了起来. 因此，由于七元组中包含了通道名称、该通道中的交互信息以及该信息的流动方向，再加上清晰的状态转换关系和相应变量的赋值，使得我们利用 MCTK 对流程的建模更加方便.

基于上述思想，本文提出了将迁移七元组转换为 MCTK 代码的转换算法，这样做是因为描述 BPEL 的控制流程时，MCTK 的编码工作繁琐、冗长，但更重要的是机械化，为了进一步提高模型检测的自动化程度，我们提出 T2M 算法，用来将产生的一组迁移七元组自动转化为 MCTK 代码.

算法 **T2M** 7-tuple→MCTK 描述代码.

输入：迁移七元组集合

输出：MCTK 描述代码

/ *假设共产生 n 个七元组 */

/ *预处理阶段：该阶段用来初始化与通道建模有关的变量和集合，产生的代码放在 MCTK 文件的最上端 */

1. 设置 *source*{ } 集合和 *dest*{ } 集合，初始化为 none. 搜索多主体组合流程 BPEL 源代码，将〈*partnerLinks*〉中的〈*partnerLink name*〉属性值放入这两个集合中；
2. 设置 Msgtype{ } 集合，初始化为 none. 搜索多主体组合流程 BPEL 源代码，将〈*variables*〉中的〈*variable name*〉属性值放入这个集合中；
3. 搜索每个 Web 服务的 WSDL 源代码，抽取代码中的〈*portType name*〉属性，找到所有与本主体相联系的通道，包括所有入通道和所有出通道；

/ *迁移描述阶段：从步 4 起是针对原子服务的建模过程，步 4 的主体定义完成后，将片段 1 和片段 2 的代码放在其后 */

4. 按照如下格式定义主体：MODULE 主体名(通道 i 名称，ObsPrm_通道 i. *Msgtype*，ObsPrm_通道 i. *source*，ObsPrm_通道 i. *dest*)；
5. 设置 *state*{ } 集合，将本 Web 服务对应产生的所有七元组中的源状态和末状态放入此集合；

/ *片段 1：对单个 Web 服务产生描述状态迁移的代码片段 */

6. for each *tuple*[i] ($i \leq n$)
7. {
8. *next*(*state*) := *case*
9. (*state* = *tuple*[i] (1)) &(*ObsPrm_tuple*[i] (5). *Msgtype* = *tuple*[i] (2)): *tuple*[i] (7);
10. }

/ *片段 2：对单个 Web 服务产生描述通道中信息变化的代码片段：*/

/ *假设满足*tuple*[i] (6) = OUT 的元组共有 m 个，则：*/

11. for each *tuple*[j] ($j \neq i \wedge j \leq m \leq n$)
12. {
13. if (*tuple*[i] (1) == *tuple*[j] (7)) then
14. {
15. *next*(*tuple*[i] (5). *Msgtype*) := *case*
(*state* = *tuple*[i] (1)) &(*ObsPrm_tuple*[j] (5). *Msgtype* = *tuple*[j] (2)): *tuple*[i] (2);
16. *next*(*tuple*[i] (5). *source*) := *case*
(*state* = *tuple*[i] (1)) &(*ObsPrm_tuple*[j] (5). *Msgtype* = *tuple*[j] (2)): *tuple*[i] (5). *sourceAgent*;
17. *next*(*tuple*[i] (5). *dest*) := *case*
(*state* = *tuple*[i] (1)) &(*ObsPrm_tuple*[j] (5). *Msgtype* = *tuple*[j] (2)): *tuple*[i] (5). *destAgent*;
}
}

算法说明：我们标记 *tuple*[j] 的第 i 项为 *tuple*[j] (i)，于是 *tuple*[j] (1)代表迁移七元组的第一项，即活动的初始状态，*tuple*[j] (7)为最后一项，即活动的末状态等等. 我们将预处理阶段和迁移描述阶段的代码按照如图 12 所示顺序组合在一起就完成了 BPEL 流程的形式化描述.

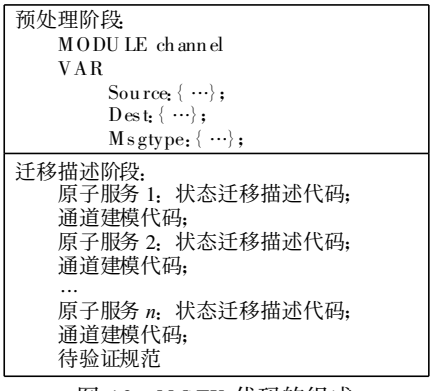


图 12 MCTK 代码的组成

从图 12 中可以看出，预处理阶段产生的代码放在整个程序的最上面. 算法步 1 用来找到所有参与流程的 Web 服务方；步 2、3 分别确定所有用到的交互变量和与每个主体相连接的“通道”. 前 3 步用来初始化为了实现通道机制所必须设置的相关变量和集合. 步 4 是主体定义，其括号中的参数皆为形式参数，用来接收由主模块传来的实参值，该步与后面生成的代码组合在一起构成对单个 Web 服务 BPEL 流程的形式化描述；步 5 声明单个 Web 服务流程的所有状态变量. 片段 1 用来产生七元组中各状态的转换关系，步 6 开始循环，产生描述主体状态迁移关系的代码；步 9 可解释为当主体状态为 *cur*(*state*) 时，通过入/出通道接收/发送信息后，状态变为 *next*(*state*). 片段 2 用来描述各通道中信息的变化情况，为了不产生混乱，我们针对主体的出通道进行建模(实际上，一个主体的入通道必为另一主体的出通道)，当某个七元组 L_1 的 *tuple*[i] (6) = *OUT* 时，说明其中的 channel 表示此主体的出通道，此时搜索其余七元组，若找到七元组 L_2，其末状态与 L_1 初状态相同，说明 L_2 是 L_1 的上一个迁移，则 L_1 对应的活动执行时，其通道的建模需考虑 L_2 通道信息的变化情况，这也是算法步 15 的思想；步 16、17 说明交互通道的源 Web 服务和目标 Web 服务. 由于算法需要针对某一七元组扫描其余 $n-1$ 个七元组以找到满足步 13 判断条件的元组集合，故而在最差情况下其时间复杂度为 $O(n!)$. 由以上说明可知，为了将 Web 服务组合建模为多主体系统，算法在转换过

程中需要先将参与 Web 服务组合的每个服务(组件)建模为主体，然后在环境中对每一主体与代表其它 Web 服务的主体的交互通道进行建模，从而构造相应的多主体系统.以下我们通过一个示例来说明算法的执行过程，实验结果表明实例流程形式化验证的自动化程度大大提高.

4　一个示例：虚拟旅游系统

本节利用我们开发的基于时态认知逻辑的模型检测工具 MCTK 对虚拟旅游系统 Virtual Travel Agency(VTA)[18] 进行验证，VTA 通过对机票预订服务商和酒店预订服务商的整合来为即将旅行的人提供服务，该系统接收用户输入，并将最终相关预订结果返回给用户.如图 13 所示.

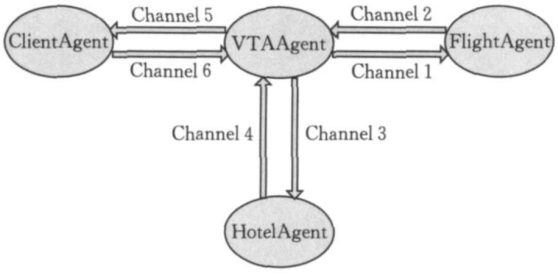


图 13　虚拟旅游订票系统示意图

该 Web 服务组合涉及 4 个主体：用户、VTA、机票预订服务商、酒店预订服务商.用户向 VTA 发送预订机票及酒店请求，之后 VTA 向机票预订服务商发送请求订票信息，机票预订服务商查询航班，若无法满足用户需求(例如航班临时取消)，则返回信息给 VTA，VTA 将该结果返回给用户，流程终止.若可以预订，则 VTA 向酒店预订服务商发送请求信息并将机票预订服务商提供的 offer 返回给用户，用户可以选择接受或者拒绝预订结果，若接受，则最终 VTA 返回用户关于酒店、机票及旅行成本等信息.

我们将这 4 个 Web 服务看作 4 个主体，用 BPEL Designer 创建每个 Web 服务的 BPEL 源代码，在此过程中需要注意各主体之间共享变量的设置.然后将此 BPEL 源代码作为 B2T 算法输入，从而生成四个相应迁移七元组集合.因此，转换工作不是针对 Web 服务组合流程的 BPEL 源代码，而是针对每个参与组合的单个 Web 服务的流程代码.限于篇幅，本文以状态数量较少的机票预订服务商为例说明转换过程，在转换前我们去掉描述该服务的 BPEL 源代码中那些没有在 2.2 节介绍的活动，然后调用 B2T 算法得到迁移七元组集合如下：

(*sleep*, *frequestMsg*, ε, ε, *Flight_PT*, IN, *get_frequestMsg*);
(*get_frequestMsg*, *fofferMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fofferMsg*);
(*send_fofferMsg*, *fackMsg*, ε, ε, *Flight_PT*, IN, *get_fackMsg*);
(*get_fackMsg*, *fticketMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fticketMsg*);
(*send_fticketMsg*, *fticketAckMsg*, ε, ε, *Flight_PT*, IN, *get_fticketAckMsg*);
(*send_fofferMsg*, *fNackMsg*, ε, ε, *Flight_PT*, IN, *get_fNackMsg*);
(*get_frequestMsg*, *fNotavailMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fNotavailMsg*).

下面将迁移七元组作为算法 T2M 的输入，得到自动生成的代码，将其输入 MCTK，从而验证该组合 Web 服务的正确性.根据算法：

(1)搜索该组合流程的 BPEL 源代码，找到用〈partnerLink name〉属性标示的所有 Web 服务，将这些 Web 服务名放入 *source*{}集合和 *target*{}集合中.在该组合流程的 BPEL 源代码中找到 4 个主体的声明，分别是 Flight_PLT、Hotel_PLT、User_PLT、VTA，分别代表机票预订服务商、酒店预订服务商、用户、VTA 代理，将它们放入上述两个集合，则：*source*={*none*, *Flight_PLT*, *Hotel_PLT*, *User_PLT*, *VTA*}；*dest*={*none*, *Flight_PLT*, *Hotel_PLT*, *User_PLT*, *VTA*}；

(2)设置 *Msgtype*{}集合，该集合存放组合流程中所有 Web 服务的交互变量，这些变量可以从组合流程 BPEL 代码中的〈variable name〉属性中抽出.对于机票预订服务商而言，需要将每个迁移七元组的第 2 项 *tuple*[*j*] (2)放入该集合，它们是 *Msgytype*={*frequestMsg*、*fofferMsg*、*fackMsg*、*fticketMsg*、*fticketAckMsg*、*fNackMsg*、*fNotavailMsg*…}；

(3)搜索本 Web 服务的 WSDL 源代码，抽取〈portType name〉属性值，找到本主体与外界联系的通道.对于本例而言，它们是 *Flight_PT* 和 *Flight_CallbackPT*，前者的源主体为 *VTA*，目标主体为 *Flight_PLT*，后者相反.为了书写方便本文分别将它们按上述顺序简写为 *ch*1、*ch*2；

(4)对机票预订服务商模块进行定义：*MODULE Flight_PLT*(*ch*1, *ch*2, *ObsPrm_ch*1.*Msgtype*, *ObsPrm_ch*1.*source*, *ObsPrm_ch*1.*dest*, *ObsPrm_ch*2.*Msgtype*, *ObsPrm_ch*2.*source*, *ObsPrm_ch*2.*dest*);

(5) 设置 *state*{ } 集合, 该集合存放了原子服务流程执行过程中经历的所有状态, 可以从该 Web 服务对应的所有迁移七元组的第一项 *tuple*[*j*] (1)和最后一项 *tuple*[*j*] (7)抽出, 这可由程序自动完成. 对于本例, 它们是 *state*={*sleep*, *get_frequestMsg*, *send_fofferMsg*, *get_fackMsg*, *send_fticketMsg*, *get_fticketAckMsg*, *get_fNackMsg*, *send_fNotavailMsg* …};

(6) 针对每个 Web 服务, 根据其相应七元组集合产生状态迁移描述代码, 对于机票预订服务商模块, 我们以其中一个七元组为例说明代码的产生过程. 例如对于七元组(*get_frequestMsg*, *fofferMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fofferMsg*): 算法说明当主体当前状态为 *get_frequestMsg*, 运用通道 *Flight_CallbackPT* 发送了信息 *fofferMsg* 后, 主体状态迁移到 *send_fofferMsg*, 该过程可写为:

```
next(state) := case
(state= get_frequestMsg) &(Flight_CallbackPT. Msgtype=
      fofferMsg): send_fofferMsg;
```

据此, 对应每个七元组, 再考虑到可观察变量概念, 最后根据算法自动生成相应状态迁移语句. 以下列出描述本主体状态迁移的部分代码:

```
next(state) := case
(state= sleep) &(ObsPrm_ch1. Msgtype= frequestMsg):
   get_frequestMsg;
(state= send_fofferMsg) &(ObsPrm_ch1. Msgtype=
   fackMsg): get_fackMsg;
(state= send_fofferMsg) &(ObsPrm_ch1. Msgtype=
   fNackMsg): get_fNackMsg;
…
```

(7) 针对每个 Web 服务, 根据其相应七元组集合产生描述通道中信息变化的代码模块, 算法首先从迁移七元组中找到 *tuple*[*i*] (6)= *OUT* 的元组, 也就是刻画主体运用出通道发送信息的迁移元组, 在本例中它们是:

(*get_frequestMsg*, *fofferMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fofferMsg*);

(*get_fackMsg*, *fticketMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fticketMsg*);

(*get_frequestMsg*, *fNotavailMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fNotavailMsg*).

然后, 找到末状态与它们的初状态相同的七元组, 按照算法展开操作. 例如: 针对 Φ(*get_fackMsg*, *fticketMsg*, ε, ε, *Flight_CallbackPT*, OUT, *send_fticketMsg*), 搜索其余元组, 若某个元组的末状态与 Φ的初状态 *get_fackMsg* 相同, 证明我们找到了 Φ的上一迁移 Φ_p, 于是, 在 Φ_p 迁移发生的情况下, 下一时刻 Φ的通道 *Flight_CallbackPT* 中的内容变为 *fticketMsg*, 并且根据(3), 该通道源主体为 *Flight_PLT*, 目标主体为 *VTA*.

由此, 对上述 3 个七元组, 根据算法自动生成相应通道描述语句, 以下列出描述本主体出通道模型的代码:

```
/ *说明在上一迁移发生情况下, 本次迁移导致的通道中的
信息变化 */
next(ch2. Msgtype) := case
(state= get_frequestMsg) &(ObsPrm_ch1. Msgtype=
   fRequestMsg): fofferMsg;
(state= get_frequestMsg) &(ObsPrm_ch1. Msgtype=
   fRequestMsg): fNotavailMsg;
(state= get_fackMsg) &(ObsPrm_ch1. Msgtype=
   fackMsg): fticketMsg;
/ *说明 Flight_CallbackPT 的源主体与目标主体 */
next(ch2. source) := case
(state= get_frequestMsg) &(ObsPrm_ch1. Msgtype=
   frequestMsg): Flight_PLT;
(state= get_fackMsg) &(ObsPrm_ch1. Msgtype=
   fackMsg): Flight_PLT;
next(ch2. dest) := case
(state= get_frequestMsg) &(ObsPrm_ch1. Msgtype=
   frequestMsg): VTA;
(state= get_fackMsg) &(ObsPrm_ch1. Msgtype=
   fackMsg): VTA;
```

需要说明的是, 之所以在针对主体的建模过程中, 只描述其出通道信息的变化而未理会入通道, 主要是考虑到统一、整齐, 防止重复建模, 实际上某一主体的入通道必然也是其它主体的出通道, 在对其它主体建模时还是会考虑到;

(8) 最后, 根据图 12, 将(4), (5), (6), (7)生成的代码组合起来, 得到机票预订服务流程的部分 MCTK 描述代码如下:

```
/ *定义主体 */
MODULE Flight_PLT (ch1, ch2, ObsPrm_ch1. Msgtype,
   ObsPrm_ch1. source, ObsPrm_ch1. dest, ObsPrm_ch2.
   Msgtype, ObsPrm_ch2. source, ObsPrm_ch2. dest)
/ *声明状态 */
State: { sleep、get_frequestMsg、send_fofferMsg、get_fackMsg、
   send_fticketMsg、get_fticketAckMsg、get_fNackMsg、
   send_fNotavailMsg}
/ *描述状态迁移 */
next(state) := case
(state= sleep) &(ObsPrm_ch1. Msgtype= frequestMsg):
   get_frequestMsg;
```

```
(state= send_ fofferMsg) &(ObsPrm_ ch1. Msgtype=
    fackMsg): get_fackMsg;
(state= send_ fofferMsg) &(ObsPrm_ ch1. Msgtype=
    fNackMsg): get_fNackMsg;
...
/ *描述通道变化 */
next(ch2. Msgtype) := case
(state= get_ frequestMsg) &(ObsPrm_ ch1. Msgtype=
    frequestMsg): fofferMsg;
(state= get_ frequestMsg) &(ObsPrm_ ch1. Msgtype=
    frequestMsg): fNotavailMsg;
(state= get_ fackMsg) &(ObsPrm_ ch1. Msgtype=
    fackMsg): fticketMsg;
next(ch2. source) := case
(state= get_ frequestMsg) &(ObsPrm_ ch1. Msgtype=
    frequestMsg): Flight_PLT;
(state= get_ fackMsg) &(ObsPrm_ ch1. Msgtype=
    fackMsg): Flight_PLT;
next(ch2. dest) := case
(state= get_ frequestMsg) &(ObsPrm_ ch1. Msgtype=
    frequestMsg): VTA;
(state= get_ fackMsg) &(ObsPrm_ ch1. Msgtype=
    fackMsg): VTA;
```

根据此方法，我们继续生成其它 Web 服务的 MCTK 代码并定义主模块，将它们按照图 12 所示放在同一 MCTK 源文件下，就自动生成了组合 Web 服务流程的 MCTK 描述代码.

在生成 MCTK 代码后，需要手工将待验证规范加入自动生成的 MCTK 代码. 我们已成功验证了若干时态认知规范. 下列规范中 SPEC 表示时态规范，ECK LnSPEC 表示时态认知规范，为了书写方便，用 agentA 代表用户主体、agentB 代表 VTA、agentC 代表航空订票系统、agentD 代表酒店预订系统. 篇幅所限，下面仅列出两个规范的验证结果. 首先是一个时态逻辑规范：

$$SPEC\ AG((agentA.state = send\_crequestMsg) \rightarrow EF(agentD.state = send\_hofferMsg)).$$

这个规范的含义是当用户主体发送了请求信息后，酒店预订服务商有可能在将来某个时刻发来相关酒店预订信息. 验证结果为 true，验证时间为 0.428s. 当用户发送请求信息后，若没有机票可供预订，则预订酒店服务不会被启动，若有机票可以预订，则 VTA 将马上向酒店预订服务商发送请求，若酒店可预订，则此规范成立.

此外，我们还验证了下列时态认知逻辑规范：

$$ECK\ LnSPEC\ AG((ch5.Msgtype = ticketMsg) \rightarrow AF(agentA\ K\ ((ch2.Msgtype = fticketMsg)\ \&\ (ch4.Msgtype = hticketMsg))))$$

其中形如“$a\ K\ f$”的公式表示主体 a 知道 f 成立. 该规范的意思是当用户预订成功相关服务后，它必然知道机票预订服务商发送了机票并且酒店预订服务商发送了酒店预订成功的相关信息. 验证结果为 true，验证时间为 1.832s.

5　结论与展望

本文从 Web 服务流程描述语言 WS-BPEL 的形式化验证需求出发，提出 BPEL 语言的形式模型并给出活动执行语义，结合该语言与模型检测工具 MCTK 自身特点提出迁移七元组这一中间形式，建立 BPEL 活动与迁移七元组之间的一一对应关系，在此基础上给出该语言到模型检测工具 MCTK 输入代码的转换算法，实现 BPEL 流程的自动化验证，从而支持时态和认知逻辑规范的验证. 实验结果表明了该方法的有效性. 下一步工作将对 BPEL 的其它活动，如错误处理等进行建模，从而更完整地支持 BPEL 建模，进一步提高建模和验证过程的自动化程度.

参　考　文　献

[1] Fu X, Bultan T, Su J. Analysis of interacting BPEL web services// Proceedings of the 13th international conference on World Wide Web (WWW'04). New York, USA, 2004: 621-630

[2] Fu Xiang. Formal specification and verification of asynchronously communicating web services[Ph. D. dissertation]. University of California, Santa Barbara, 2004

[3] Kazhamiakin R. Formal analysis of web service composition [Ph. D. dissertation]. University of Trento, Trento, 2007

[4] Foster H, Uchitel S, Magee J, Kramer J. Model-based verification of web service compositions// Proceeding of the 18th IEEE International Conference on Automated Software Engineering(ASE'03), Montreal, Canada, 2003: 152-163

[5] Foster H, Uchitel S, Magee J, Kramer J. Compatibility verification for web service choreography// Proceedings of the IEEE International Conference on Web Services (ICWS'04). San Diego, California, USA, 2004: 738-741

[6] Walton C D. Model checking multi-agent web services// Proceedings of the 2004 AAAI Spring Symposium on Semantic Web Services. Stanford, California, 2004: 68-75

[7] Nakajima S. Lightweight formal analysis of web service flows. Progress in Informatics, 2005(2): 57-76

[8] Mongiello M, Castelluccia D. Modelling and verification of BPEL business processes// Proceedings of the Joint Meeting of the 4th Workshop on Model-Based Development of Computer-Based Systems and the 3rd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software. Potsdam, Germany, 2006: 144-148

[9] Ouyang Chun, Verbeek E, van der Aalst W M P, Breutel S, Dumas Ma, ter Hofstede A H M. WofBPEL: A tool for automated analysis of BPEL processes// Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC 2005). Amsterdam, The Netherlands, 2005: 484-489

[10] Ouyang Chun, van der Aalst W M P, Breutel S, Dumas M, ter Hofstede A H M, Verbeek H M W. Formal semantics and analysis of control flow in WS-BPEL. Science of Computer Programming, 2007, 67(2-3): 162-198

[11] Luo Xiang-Yu. Symbolic model checking multi-agent systems [Ph. D. dissertation]. Sun Yat-sen University, Guangzhou, 2006 (in Chinese)
(骆翔宇. 多主体系统的符号模型检测[博士学位论文]. 中山大学, 广州, 2006)

[12] Su Kai-Le. Model checking temporal logics of knowledge in distributed systems// Proceedings of the 19th National Conference on Artificial Intelligence, the 16th Conference on Innovative Applications of Artificial Intelligence. San Jose, California, USA, 2004: 98-103

[13] Su Kai-Le, Sattar A, Luo Xiang-Yu. Model checking temporal logics of knowledge via OBDDs. The Computer Journal, 2007, 50(4): 403-420

[14] Lomuscio A, Qu Hong-Yang, Raimondi F. MCMAS: A model checker for the verification of multi-agent systems// Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009). Grenoble, France, 2009: 682-688

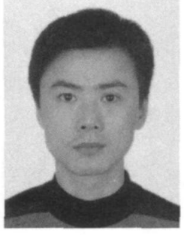
[15] Gammie P, van der Meyden R. MCK: Model checking the logic of knowledge// Proceedings of the 16st International Conference on Computer Aided Verification (CAV 2004). Boston, MA, USA, 2004: 479-483

[16] Lomuscio A, Qu Hong-Yang, Solanki M. Towards verifying contract regulated service composition// Proceedings of the IEEE International Conference on Web Services (ICWS' 08). Beijing, China, 2008: 254-261

[17] Lomuscio A, Qu Hong-Yang, Sergot M, Solanki M. Verifying temporal and epistemic properties of web service composition// Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC 2007). Vienna, Austria, 2007: 456-461

[18] Formal R K. Analysis of web service composition [Ph. D. supervisor]. University of Trento, Trento, 2007: 79-84

[19] Halpern J Y, Vardi M Y. Model checking vs. theorem proving: a manifesto// Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning. Cambridge, Massachusetts, USA, 1991: 325-334



LUO Xiang-Yu, born in 1974, Ph. D., associate professor. His research interests include model checking, temporal logics, epistemic logics, reasoning about knowledge, multi-agent systems, and verification of security protocols.

TAN Zheng, born in 1982, M. S.. His research interests include model checking, temporal logics, epistemic logics, and Web services.

SU Kai-Le, born in 1964, Ph. D., professor, Ph. D. supervisor. His research interests include model checking, reasoning about knowledge, non-monotonic reasoning, multi-agent systems, modal logics, temporal logics, probability reasoning, and verification of security protocols, logic programming.

WU Li-Jun, born in 1965, Ph. D., associate professor. His research interests include model checking, reasoning about knowledge, multi-agent systems, and verification of security protocols.

Background

Today, Web services are widely distributed over the Internet, the tasks of composite Web services are solved through the interactions between these Web services involved. However, during the interaction process, it is possible that some unexpected or "not normal" information interactions will occur, which may cause some incorrect result generated and seriously lower the quality and robustness of a system. On the other hand, due to the autonomy of single Web service and the loose-coupling between composite Web services, it is very suitable to model Web service compositions as multi-agent systems. Therefore, to ensure Web services can provide correct function, it is necessary to apply formal verification techniques, such as model checking, to the verification of the temporal and epistemic specifications of Web service compositions, where epistemic specifications are proper to multi-agent systems. This work is gradually becoming a hot area of research on formal verification domain. To the best of our knowledge, such formal verification techniques for Web service composition are in the initial research stage, and the entire verification process is not automatic, there are a lot of manual operations during the verification process, especially in the modeling stage. Therefore, in order to make the Web services verification process more automatic, the authors propose a series of algorithms for transla-

ting the BPEL flow of Web service composition into the input code of MCTK, a symbolic model checker for temporal epistemic logic developed by the authors. The proposed algorithms can be used to model and verify the temporal and epistemic logical specifications of Web service compositions by using MCTK. Some of the authors' works are listed as follows:

(1) For the Business Process Execution Language BPEL, give the semantics of activities; Introduce the intermediate form of transition seven-tuples to represent the state transition relations; Based on the semantics of these activities, establish the mapping from BPEL activities to transition seven-tuples. It is the foundation for the automatic verification of BPEL flow.

(2) Develop the algorithms B2T and F2T for translating the control flow, data flow of BPEL into transition seven-tuples. These algorithms will automatically convert BPEL process into a set of transition seven-tuples, such that we can convert the composite work flow into a multi-branch state transition diagram.

(3) In order to model the control flow automatically, develop the T2M algorithm to generate the MCTK input code according to the relation between the transition seven-tuples in a set. The characteristics of the MCTK input language is also considered when generating the MCTK input code. This algorithm avoids many tedious manual operations and implements automatic modeling of control flow.

(4) The authors apply the proposed approach to the verification of the example of Virtual Travel Agency. Some temporal and epistemic specifications are checked successfully via MCTK. The experiment results show the correctness and effectiveness of the proposed approach.