传值进程模型检测中诊断信息的生成*

刘 剑+, 林惠民

(中国科学院 软件研究所 计算机科学重点实验室,北京 100080)

Diagnostic Information Generation in Model Checking Value-Passing Processes

LIU Jian+, LIN Hui-Min

(Key Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

+ Corresponding author: Phn: 86-10-62562796, E-mail: ljian@ios.ac.cn

http://lcs.ios.ac.cn

Received 2002-05-10; Accepted 2002-07-29

Liu J, Lin HM. Diagnostic information generation in model checking value-passing processes. *Journal of Software*, 2003,14(1):1~8.

Abstract: Automatic diagnostic information generation is one of the remarkable advantages of model checking methods. It is very important to understand the reason for the failure and fix the problem. In this paper, how to generate effective diagnosis in model checking value-passing processes is discussed. Two diagnostic forms, proof graph and witness, are defined. Moreover, algorithms are proposed to construct them from the search states space in model checking process. By this way, useful diagnoses are generated from the existing information by less calculation. Besides above, the algorithms have been implemented and used to analyze several cases. The experimental results show that this method is efficient.

Key words: process algebra; model checking; proof graph; witness; diagnosis generation

摘 要: 诊断信息自动生成是模型检测方法的基本特征之一,对分析和排错具有重要的意义.讨论了传值进程模型检测中诊断信息的生成问题.引入了两种诊断信息的表示结构:证明图和示例;提出了两种诊断信息构造算法.所采用的方法是从检测过程保存的依赖信息中抽取证明图和示例,这样可以继承已有的信息,从而减少计算量.相应的算法已经实现并用实例作了分析测试.实验结果表明该方法是有效的.

关键词: 进程代数;模型检测;证明图;示例;诊断生成

中图法分类号: TP301 **文献标识码**: A

模型检测是一种针对并发系统的自动分析与验证技术.其基本原理是,用状态迁移图 S 表示所要分析的系统,用模态/时序逻辑公式 φ 描述所要检查的性质.系统是否具有所要求的性质就归结为 $S \vDash \varphi$ 是否成立,对有限状态系统,这个问题是可判定的.与其他验证方法(如模拟、测试、机器证明等)相比,模型检测方法有诸多优点[1],主要体现为:验证过程完全自动化;当抽象模型不满足逻辑公式时,检测工具会自动产生一个反例,说明为什么不满足,以用于查错和修改.诊断信息的生成是模型检测方法的重要组成部分.好的诊断信息不仅有利于人们对

* Supported by the National Natural Science Foundation of China under Grant No.69833020 (国家自然科学基金)

第一作者简介: 刘剑(1976－),男,云南石屏人,博士生,主要研究领域为并发系统的自动验证,模型检测方法的理论和应用.

问题的理解,更有助于在模型不满足逻辑性质时找到问题并排除错误.迄今为止,在进程代数的模型检测方法中,研究工作主要集中于非传值的进程模型,对传值进程(value-passing processes)的模型检测,通常的做法是先将问题转化为非传值的情况,然后再进行检测.这种方法的一个弊病是,产生的诊断信息针对转化后的非传值进程,而不是原先的传值模型,因而对修改和排错没有多大价值.文献[2]基于动态实例化的思想,提出了一种传值进程的模型检测算法.该算法在有限数据域的情况下能取得较好的时空效率.此外,算法还能处理一类无限数据域的情况,即数据无关的传值进程.但是,文献[2]只对检测算法进行讨论,对检测过程中如何生成诊断信息并未加以说明.

本文在文献[2]的基础上讨论传值进程模型检测中诊断信息的生成问题.我们的思想是,在检测过程中记录状态间的依赖信息,从依赖信息构造闭包的状态依赖图.依赖图从语义的角度说明了系统和逻辑公式之间的满足关系,包含了确定满足关系的极少依赖信息.我们对依赖图中的信息进行整理,提取其中的迁移关系,得到一个状态迁移图.该迁移图就是通常所说的正(反)例,它更直观地说明系统为什么满足(不满足)逻辑公式描述的性质.限于篇幅,对于一些基础概念,本文不再赘述,主要是带赋值的符号迁移图(STGA)[3]和图式谓词μ-演算(模态图)[2],前者用来表示传值进程模型,后者是描述传值模型性质的时序逻辑.另外,文献[2]中的模型检测算法也不再列出,相关内容请参阅有关文献.为了方便阅读,本文使用的记号与文献[2,3]相同.本文第 1 节对我们所关心的模型检测算法进行分析,得到几个重要的定义和定理,这是后叙内容的理论基础.第 2 节具体讨论诊断信息的生成,引入证明图、示例等概念,给出证明图和示例的生成算法,最后对数据无关情况下的诊断生成进行说明.第 3 节给出一个实例的分析结果.最后是相关工作的比较,并对全文进行总结.

1 模型检测算法分析

诊断信息的生成总是相对于特定的检测算法.本节分析我们所关心的模型检测算法,引入讨论诊断信息生成所需要的几个概念和结果.根据文献[2],对给定的 STGA $\mathcal{G}$ 和模态图 $\mathcal{M}$,令 $p_0 \equiv n\rho_0'$ 和 $r\rho_0$ 为要检测的目标进程和命题,其中 n 和 r 分别为 $\mathcal{G}$,$\mathcal{M}$ 的根结点,ρ_0' 和 ρ_0 是初始计值.模型检测就是要判定 $p_0 \in [\![r]\!]\ \rho_0$ 是否成立.首先假定 $\mathcal{G}$ 和 $\mathcal{M}$ 都是构造在有限数据域 Val 上.分析文献[2]中的检测算法,可以得到如下定义和定理.

定义 1(后继状态集(following states)).设 $s \equiv (p, n\rho)$ 是检测过程中生成的任意状态,定义 s 的后继状态集 $FOL(s)$ 为

若 $L_O(n) = b \in BExp$, 则 $FOL(s) = \varnothing$;

若 $L_O(n) = \{\wedge, \vee\}$,则 $FOL(s) = \{(p,(n',\rho)) \mid n \to n'\}$;

若 $L_O(n) \in \{\forall x, \exists x\}$,则 $FOL(s) = \{(p,(n',\rho\{x \mapsto v\})) \mid n \to n', v \in Val\}$;

若 $L_O(n) = \overline{x} := \overline{e}$,则 $FOL(s) = \{(p,(n',\rho\{\overline{x} \mapsto \rho(\overline{e})\})) \mid n \to n'\}$;

若 $L_O(n) = \beta \in \{\langle a\rangle, [a]\}$,则 $FOL(s) = \{(p',(n',\rho)) \mid p \xrightarrow{a} p', n \to n'\}$;

若 $L_O(n) = \beta \in \{\langle c!e\rangle, [c!e]\}$,则 $FOL(s) = \{(p',(n',\rho)) \mid p \xrightarrow{c!\rho(e)} p', n \to n'\}$;

若 $L_O(n) = \beta \in \{\langle c?x\rangle, [c?x]\}$,则 $FOL(s) = \{(p'[v/y],(n',\rho\{x \mapsto v\})) \mid p \xrightarrow{c?y} p', n \to n', v \in Val\}$.

令 $\mathcal{R}$ 表示检测算法在执行过程中访问过的状态空间,按照嵌套深度,$\mathcal{R}$ 可以划分为 $k+1$ *个子集 $\{C_i \mid 0 \le i \le k\}$ (其中某些 C_i 可能为空集).又令 $C_i' = \{s \in C_i \mid s.instack = \text{false}\}$ 且 $\mathcal{R}' = \bigcup_{0 \le i \le k} C_i'$.

定义 2(直接依赖状态集(directly dependant states)). 对任意状态 $s \in \mathcal{R}'$,定义其直接依赖状态集 $DDEP(s)$ 为 $DDEP(s) = \{s' \mid s' \in FOL(s) \text{且} s \in s'.D\}$.

定义 3(依赖状态集(dependant states)). 对任意状态 $s \in \mathcal{R}'$,定义其依赖状态集 $DEP(s)$ 为满足如下条件的最小集合:

(1) $s \in DEP(s)$;

* 这里设模态图 $\mathcal{M}$ 有 k+1 块:$B_0, B_1, \ldots, B_k$.

(2) 若 $s'\in DEP(s)$ 且 $s'.instack=\text{false}$, 则 $DDEP(s')\subseteq DEP(s)$.

由 $DEP(s)$ 的定义可知下面的性质成立:

$$\forall s'\in DEP(s)\text{且}s'.instack=\text{false},\text{则}DEP(s')\subseteq DEP(s).$$

也就是说,对给定的 $s\in\mathcal{R}'$,若 $s'\in DEP(s)$ 且 s' 是栈外状态,则确定 s' 取值的所有状态也包含在 s 的依赖状态集中, $DEP(s)$ 构成了一个闭包的状态集.因此,可将 $DEP(s)$ 看作函数 $\zeta_{DEP(s)}:N\to Eval\to 2^S$,定义为

$$\zeta_{DEP(s)}n\rho'=\{p\mid\exists s'\in DEP(s),s.t.s'=(p,n\rho')\text{且}s'.status=VISITED(\text{true})\}.$$

$DEP(s)$ 中的元素按照嵌套深度可划分为 $k+1$ 个子集,令 $C_{k,s}=C_k\cap DEP(s),0\le i\le k$, 则使用模态图语义定义的记号可得

$$[\![\,C_{k,s}\,]\!]\ \ \rho\zeta_{DEP(s)}=\sigma_k(\lambda\eta\ \|C_{k,s}\|\ \rho\zeta_{DEP(s)}\{\eta\lceil C_{k,s}\}),$$

$$[\![\,C_{i,s}\,]\!]\ \ \rho\zeta_{DEP(s)}=\sigma_i(\lambda\eta\|C_{i,s}\|\rho\zeta_{DEP(s)}\{\eta\{\ [\![\,C_{i+1,s}\,]\!]\ \}\}),\ \ 0\le i<k.$$

令 $[\![\,n\,]\!]$ 为 $[\![\,C_{i,s}\,]\!]$ 在 n 上的投影,其中 n 满足存在 $s'=(p,n\rho')\in C_{i,s}$ 的条件.分析模型检测算法中函数 *Close*()的计算过程,有下面的定理成立.

定理 1. ModelCheck 中的 while 循环满足下列不变式:

INV1 对任意 $s\in\mathcal{R}'$,若 $s.status=VISITED(b)$,则 $\forall s'\in DEP(s),s'.status=VISITED(b)$;

INV2 对任意 $s\in\mathcal{R}'$, $\forall s'=(p,n\rho)\in DEP(s)\cap\mathcal{R}'$, $s'.status=VISITED(\text{true})$ iff $p\in[\![\,n\,]\!]\ \rho\zeta_{DEP(s)}$.

证明:(1) 不变式 INV1.

初始情况下 $\mathcal{R}'$ 为空,蕴含式前件为 false, INV1 成立.下面证明函数 *Close*()的计算过程保持 INV1.假设 INV1 成立.此时调用 Close 计算状态 $top()=s$ 的值,且设 $s.status=VISITED(b')$.对 Close 中 B 的计算结果分情形讨论.

① 若 $B=DEFERRED$,则有新状态加入栈中, s 仍保留在栈中, $\mathcal{R}'$ 没有变化.且对任意 $s'\in\mathcal{R}'$, $DEP(s')$ 也没有变化,由假设可知,INV1 成立.

② 若 $B=VISITED(b)$, s 取得新值 b,同时被加入到 $\mathcal{R}'$ 中,且设 $b=b'$.由语义函数 $\|\cdot\|$ 的单调性,CheckAnd 和 CheckOr 的计算过程保证:对任意 $s'\in DDEP(s)$, $s'.VISITED(b)=s.VISITED(b)$.又由归纳假设可得 $\forall s'\in\mathcal{R}'$ 且 $s\ne s',DEP(s')$ 中状态的值与 s' 的值相同.按照 $DEP(*)$ 的定义进行结构归纳可得:对任意 $s'\in\mathcal{R}'$, $\forall s''\in DEP(s')$, s'' 和 s' 的值均相同,INV1 成立.

③ 若 $B=VISITED(b)$ 且 $b\ne b'$,由②可知,对任意 $s'\in DDEP(s)$, $s''.VISITED(b)=s.VISITED(b)$ 成立,同时注意到,在 $\mathcal{R}$ 中仅有 s 的值发生变化,并且对任意 $s''\in\mathcal{R}'$ 且 $s\in DEP(s'')$, *Restore*()重新将 s'' 加入栈中.因此 INV1 成立.

(2) 不变式 INV2.

初始情况下 $\mathcal{R}'$ 为空,蕴含式前件为 false, INV2 成立.下面证明函数 *Close*()的计算过程保持 INV2.假设 INV2 成立.此时调用 Close 计算状态 $top()=s$ 的值,且设 $s.status=VISITED(b')$.对 Close 中 B 的计算结果分情形讨论.

① 若 $B=DEFERRED$,则有新状态加入到栈中,同时 s 仍保留在栈中, $\mathcal{R}'$ 没有变化.且对任意 $s'\in\mathcal{R}'$, $DEP(s')$ 也没有变化,INV2 成立.

② 若 $B=VISITED(b)$,则 s 取得新值 b, s 被加入到 $\mathcal{R}'$ 中,且设 $b=b'$.此时任选 $s'\in\mathcal{R}'$.由 $DEP(s')$ 的定义和 B 的计算过程可知: $\forall s''\in DEP(s')\cap\mathcal{R}'$,其值满足 $\|\rho\zeta_{DEP(s')}$, s'' 的值只依赖于 $DEP(s')$ 中的状态值,并且计算过程没有用到其内部块中状态元素的缺省值.INV2 成立.

③ 若 $B=VISITED(b)$ 且 $b\ne b'$,则 s 取得新值 b 且被加入到 $\mathcal{R}'$ 中,但是,过程 *Restore*()递归地将 $\mathcal{R}'$ 中依赖于 s 旧值的状态重新加入栈,保证了 INV2 成立. □

直观地,INV1 说明,任意栈外状态的当前值与其依赖状态的取值一致;INV2 说明,对任意栈外状态,*Close*()的计算过程保证其值和语义定义一致,并且只与其依赖状态的取值有关.按照文献[2],令 $\mathcal{R}_t$ 表示算法结束时访问过的所有状态的集合,则状态空间 $|\mathcal{R}_t|$ 的上界为 $|\overline{\mathcal{G}}|\cdot|\mathcal{M}|\cdot|Val|^V$,算法访问的边的数目 $|\to_{\mathcal{R}_t}|$ 的上界为 $(|\to_{\overline{\mathcal{G}}}|+2|\overline{\mathcal{G}}|)\cdot|\mathcal{M}|\cdot|Val|^V$.其中 $\overline{\mathcal{G}}$ 表示由 $\mathcal{G}\rho_0'$ 导出的 LTS, $|\overline{\mathcal{G}}|$ 和 $|\to_{\overline{\mathcal{G}}}|$ 分别表示 $\overline{\mathcal{G}}$ 的结点数和迁移数. $|\mathcal{M}|$ 表示模态图 $\mathcal{M}$ 的结点数, $|Val|$ 表示数据域的大小, V 表示 $\mathcal{M}$ 中结点的自由变量的最大数目.

2 诊断信息的生成

诊断信息是检测算法结束时从遍历的状态及其依赖关系中抽象出来的,可分为两类:证明图(proof graph)和示例(witness).证明图本质上是等价于 Tableau[4](或 proof structure[5])的一种结构.它包括了要证明 $p_0 \in [\![r]\!]\rho_0$(或 $p_0 \notin [\![r]\!]\rho_0$)成立所需的状态结点及其上的依赖关系.证明图从语义定义的角度说明了目标命题为什么成立.示例是由从证明图中抽象出来的进程结点和迁移关系构成的 STGA $\mathcal{G}$ 导出 LTS 的子图.当检测结果为真时,示例是 LTS 中满足逻辑性质的一个迁移子图,否则该子图给出足够的信息说明 LTS 不满足逻辑公式描述的性质.示例更直观地说明了进程和逻辑性质之间的满足关系.

2.1 证明图

由定理 1 可以得出,当检测算法结束时,对目标状态 $t=(p_0,r\rho_0)$ 有

$$\forall s=(p,n\rho)\in DEP(t), s.status=VISITED\,(\text{true}) \quad \text{iff} \quad p\in [\![n]\!]\ \rho\zeta_{DEP(t)}.$$

也就是说,t 的取值只与 $DEP(t)$ 中的状态有关,并且 $DEP(t)$ 是闭包的.同时注意到,在检测算法中,B 值的计算过程保证:对任意 $s\in DEP(t)$, $DEP(t)$ 只包含有满足语义函数 $\|\cdot\|$ 的 s 的极少后继状态.例如,对于状态 $s=(p,n\rho)$,若 B 的值为 false 且 $L_O(n)=\wedge$, 则 $DDEP(s)$ 只有一个直接依赖状态;若 B 的值为 true 且 $L_O(n)=\vee$, $DDEP(s)$ 中也只有一个直接依赖状态,其他情况类似.因此,可以通过 $DEP(t)$ 中的状态及依赖信息来构造 t 的证明图.

定义 4(证明图(proof graph)). 设 $t=(p_0,r\rho_0)$ 是初始状态,t 的证明图 $PG=(V,E)$ 定义如下,其中 $V\subseteq \mathcal{R}_t$ 是检测结束时状态空间的子集, $E\subseteq V\times V$ 是 V 中状态间的依赖关系.满足:

(1) $V=DEP(t)$;

(2) $E=\{(s,s')\,|\,s\in V\wedge s'\in DDEP(s)\}$.

证明图生成算法如下:

输入: $(p_0,r\rho_0)$,模型检测算法结束时各状态的 $s.D$;

输出: $(p_0,r\rho_0)$ 的证明图 $PG=(V,E)$.

PGGen($(p_0,r\rho_0)$,各状态的 $s.D$): $PG=(V,E)$

$V:=\{(p_0,r\rho_0)\}; E:=\varnothing; A:=\{(p_0,r\rho_0)\};$

while $A\neq\varnothing$ do

{select s as $(p,n\rho)\in A; A=A\setminus\{s\}$;

$FOL(s)=$ case $L_O(n)$ of

$be\Rightarrow\varnothing$

$\mid \bar{x}:=\bar{e}\Rightarrow\{(p,(n',\rho\{\bar{x}\mapsto\rho(\bar{e})\}))\,|\,n\to n'\}$

$\mid \wedge\Rightarrow\{(p,(n',\rho))\,|\,n\to n'\}$

$\mid \forall x\Rightarrow\{(p,(n',\rho\{x\mapsto v\}))\,|\,n\to n', v\in Val\}$

$\mid [c!e]\Rightarrow\{(p',(n',\rho))\,|\,p\xrightarrow{c!\rho(e)}p', n\to n'\}$

$\mid [c?x]\Rightarrow\{(p'[v/y],(n',\rho\{x\mapsto v\}))\,|\,n\to n', p\xrightarrow{c?y}p', v\in Val\}$

$\mid$ ...

$DDEP(s)=\{s'\,|\,s'\in FOL(s)$ and $s\in s'.D\}$

if $DDEP(s)\neq\varnothing$ then

for each $s'\in DDEP(s)$ do

{add (s,s') to E;

if $s'\notin V$ then {add s' to V; add s' to A}

}

}

定理 2. 算法对二元组 $(p_0,r\rho_0)$ 生成证明图 $PG=(V,E)$,且最坏时间复杂度为 $O(|\mathcal{R}_t|^3)$.

证明:算法的正确性由 while 循环的不变式保证.

INV $\forall s\in V/A, DDEP(s)\subseteq V$ 且 $\forall s'\in DDEP(s),(s,s')\in E$.

直观上,算法是一个求闭包的过程.V 中存放所有已被访问过的状态结点,A 中的元素是 V 中尚未处理其直接依赖状态的元素集.因此,算法结束时 $A=\varnothing$,此时有 $\forall s\in V, DDEP(s)\subseteq V$ 且 $\forall s'\in DDEP(s),(s,s')\in E$ 成立,又 $(p_0,r\rho_0)$ 始终在 V 中,所以算法得到 $(p_0,r\rho_0)$ 的证明图.

在算法执行过程中,while 语句最多执行 $|\mathcal{R}_t|$ 次,在每次执行时,最复杂的部分是对 $DDEP(s)$ 作处理的语句.最坏情况下 s 依赖所有状态,$DDEP(s)$ 等于 $|\mathcal{R}_t|$.对 $DDEP(s)$ 处理的语句最多要花 $O(|\mathcal{R}_t|^2)$ 的时间.因此,算法的最坏时间复杂度为 $O(|\mathcal{R}_t|^3)$.在空间消耗上,集合 A 最多占用 $|\mathcal{R}_t|$ 个存储单元.最终算法生成的证明图在最坏情况下含有 $|\mathcal{R}_t|$ 个结点和 $|\mapsto_{\mathcal{R}_t}|$ 条边. □

2.2 示 例

示例是 STGA $\mathcal{G}$ 的导出迁移图的子图.它由从证明图中抽象出的进程和迁移关系构成.示例中的进程是证明图中进程的集合,迁移关系的提取是检测过程中状态展开过程的逆过程.对状态 $(p,n\rho)$ 而言,若 $L_O(n)$ 是模态算子,则进程 p 与其直接依赖状态的进程之间存在迁移关系;否则,其直接依赖状态是通过对逻辑运算 $L_O(n)$ 展开而得到,p 与直接依赖状态的进程相同.

定义 5(示例 Witness). 设 $PG=(V,E)$ 是 t 的证明图,则 $p_0\in [\![r]\!]\rho_0$(或 $p_0\notin [\![r]\!]\rho_0$)的一个示例 $Wit=(Q,R,L_Q)$ 定义如下,其中 Q 是 $\mathcal{G}$ 导出迁移图中进程的子集,R 是 Q 上的迁移关系,L_Q 是 Q 到命题幂集的映射,满足:

(1) 若 $(p,n\rho)\in V$,则 $p\in Q$;

(2) 若 $(p,n\rho)\in V$,则 $n\rho\in L_Q$;

(3) 若 $(p,n\rho),(p',n'\rho')\in V$,$((p,n\rho),(p',n'\rho'))\in E$ 且 $L_O(n)\in Modop$,则 $(p,p')\in R$.

具体地,示例又可分别称为正例(example)和反例(counterexample).若 $p_0\in [\![r]\!]\rho_0$ 成立,则 $Wit=(Q,R,L_Q)$ 是 $p_0\in [\![r]\!]\rho_0$ 的正例.否则,$Wit=(Q,R,L_Q)$ 是 $p_0\in [\![r]\!]\rho_0$ 的反例.对于正例和反例,$L_Q(p)$ 中的命题和 p 有不同的满足关系.由定理 1 的 INV1 可知:在正例中,$\forall p\in Q,\forall n\rho\in L_Q(p),(p,n\rho).status=VISITED(\text{true})$,因此 $L_Q(p)$ 中均是 p 满足的命题.相反地,对于反例,$L_Q(p)$ 中是 p 不满足的命题.

示例生成算法如下:

输入:$(p_0,r\rho_0)$ 及其证明图 $PG=(V,E)$;

输出:$(p_0,r\rho_0)$ 的一个示例 $Wit=(Q,R,L_Q)$.

WitGen($(p_0,r\rho_0)$,证明图 $PG=(V,E)$): $Wit=(Q,R,L_Q)$

$Q:=\{p_0\}; R:=\varnothing; L_Q(p_0):=\{r\rho_0\}; A:=\{(p_0,r\rho_0)\};$

while $A\neq\varnothing$ do

{select s as $(p,n\rho)\in A$; $A:=A\setminus\{s\}$;

if $E(s)\neq\varnothing$ then

for each $s'\equiv(p',n'\rho')\in E(s)$ do

{if $L_O(n)\in Modop$ then add (p,p') in R;

if $p'\notin Q$ then { add p' to Q; $L_Q(p'):=\{n'\rho'\}$; add $(p',n'\rho')$ to A}

elseif $n'\rho'\notin L_Q(p')$ then { add $n'\rho'$ to $L_Q(p')$; add $(p',n'\rho')$ to A}

endif

}

}

定理 3. 算法对二元组 $(p_0,r\rho_0)$ 生成示例 $Wit=(Q,R,L_Q)$,且最坏时间复杂度为 $O(|\mathcal{R}_t|^3)$.

证明:正确性由示例的定义即可得到.时间复杂度同证明图的生成算法都为 $O(|\mathcal{R}_t|^3)$.空间上集合 A 最多占用 $|\mathcal{R}_t|$ 个存储单元.最坏情况下,算法生成的示例是整个 STGA $\mathcal{G}$ 的导出迁移图,此时 Q 包含 $|\overline{\mathcal{G}}|$ 个结点,R 包含

$\vdash\rightarrow_{\bar{G}}$|条边. □

注意到在上述算法中,对任意状态 s,$E(s)$和 $DDEP(s)$ 一致,因此示例生成算法也可以改用 $(p_0, r\rho_0)$ 和各状态的 $s.D$ 作输入,相应地,只需在算法中增加求解 $DDEP(s)$ 的部分并将 $E(s)$ 改为 $DDEP(s)$ 即可.此时算法不需要保存证明图作为中间结果.我们的实现就采用了这样的模式.

上述结果预先假设了数据域 *Val* 是有限数据域,这一限制可以放宽到一类无限数据域的情况,即数据无关(data independent)的数据域.数据无关的数域要求其中的常量和变量只能进行等于或不等于判断,除此之外不能进行其他任何操作.此时,在模型检测中,对无限数据域需要有限个符号值来代表其中的数据值,同时要对检测算法作相应的改动[2].诊断信息生成方法不需要更改,生成的证明图和示例含有符号值.

3 实例分析

上述算法已经用 SML/NJ 实现,并对几组实例作了测试.附录中的输入文本是测试实例之一,描述的是 Alternating-Bit 协议(ABP)的一个变种,媒体 Mlossy 在信息传送过程中可能将数据丢失.文件中 message 表示传送的数据类型,取值为 1~10 的整数.所有的进程标识、谓词变量、通道和变量都有相应的类型声明.Conjecture 语句定义要求解的问题,语句左部是进程,右部是要检测的逻辑性质.Where 语句包含各个进程标识的递归定义.要检测的性质以谓词方程的形式列在 predicate equation 部分.我们首先检测性质 0,得到 ABP 中不存在死锁状态;然后对所关心的 3 条性质逐条进行检测.检测过程主要包含两个处理步:首先由程序将进程定义和谓词方程转化为相应的 STGA 和模态图,然后对 STGA 和模态图进行检测,得到返回信息.

性质 1. 对任意信息 *m*,ABP 从端口 accept 接收后最终总能从 deliver 端送出,且这样的传送过程可以反复进行.

Model Check Result: false

Diagnosis Information (Counterexample):

S(*sb*)|*Mlossy*|*R*(*rb*){*rb*=false,*sb*=true}::--*accept*?*sm*->*S*1(*sb*,*sm*)|*Mlossy*|*R*(*rb*){*rb*=false,*sb*=true,*sm*=1};

*S*1(*sb*,*sm*)|*Mlossy*|*R*(*rb*){*rb*=false,*sb*=true,*sm*=1}::--*tau*->*S*1(*sb*,*sm*)|*rack*!*ma*.*Mlossy*+*Mlossy*|*R*(*rb*){*ma*=false, *rb*=false,*sb*=true,*sm*=1};

*S*1(*sb*,*sm*)|*rack*!*ma*.*Mlossy*+*Mlossy*|*R*(*rb*){*ma*=false,*rb*=false,*sb*=true,*sm*=1}::--*tau*->*S*1(*sb*,*sm*)| *Mlossy*|*R*(*rb*){*ma*=false,*rb*=false,*sb*=true,*sm*=1}.

算法判定 ABP 不满足性质 1,并给出一个反例.反例是 ABP 的一条迁移路径,其中包含 3 个进程结点和 3 条迁移边:ABP 从端口 accept 接受信息 *sm*=1,其后 *Mlossy* 与 *R*(*rb*)进行了一次同步数据交换,但是由于 *Mlossy* 将数据包丢失,*Mlossy* 与 *R*(*rb*)再次进行同步,系统进入了一个 τ 环,信息没有从 deliver 端送出.

性质 2. 对任意信息 *m*,ABP 从端口 accept 接收后,在满足公平性约束*的前提下总能从 deliver 端送出,且这样的传送过程可以反复进行.

Model Check Result: true

检测算法判定 ABP 满足该性质,并且提供正例说明.正例中对所有 1~10 的整数都进行了展开,限于篇幅,这里不再给出.

性质 3. 对任意信息 *m*,ABP 从端口 accept 接收后,存在一条迁移路径最终将 *m* 从 deliver 端送出.注意到在 ABP 模型中,message 是数据无关的数据域,因此可以在无限数据域的情况下进行模型检测.用语句 message:data 替换输入文本中的相关类型声明,该语句说明 message 是数据无关的数据域,再进行检测得到如下结果:

Model Check Result: true

Diagnosis Information (Example):

---

* 这里的公平性约束是指强公平性(strong fairness)[6],即事件被无限次使能(enabled),则最终要发生.我们考虑的是事件 *deliver*!应被公平对待.直观地,性质 2 可以理解为事件 *accept*?*m* 发生后,系统进入这样的路径:或者 *deliver*!*m* 最终发生,或者该路径对 *deliver*!*m* 不公平.

S(*sb*)|*Mlossy*|*R*(*rb*){*rb*=false,*sb*=true}::--*accept*?*sm*->*S*1(*sb*,*sm*)|*Mlossy*|*R*(*rb*){*rb*=false,*sb*=true,*sm*=*data*1};

*S*1(*sb*,*sm*)|*Mlossy*|*R*(*rb*){*rb*=false,*sb*=true,*sm*=*data*1}::--*tau*->*S*1(*sb*,*sm*)|*rack*!*ma*.*Mlossy*+*Mlossy*|*R*(*rb*){*ma*=false,*rb*=false,*sb*=true,*sm*=*data*1};

*S*1(*sb*,*sm*)|*rack*!*ma*.*Mlossy*+*Mlossy*|*R*(*rb*){*ma*=false,*rb*=false,*sb*=true,*sm*=*data*1}::--*tau*->*S*2(*sb*,*sm*)|*r*!(*ma*,*mm*).*Mlossy*+*Mlossy*|*R*(*rb*){*ma*=true,*mm*=*data*1,*rb*=false,*sb*=true,*sm*=*data*1};

*S*2(*sb*,*sm*)|*r*!(*ma*,*mm*).*Mlossy*+*Mlossy*|*R*(*rb*){*ma*=true,*mm*=data1,*rb*=false,*sb*=true,*sm*=*data*1}::--*tau*->S2(*sb*,*sm*)|*Mlossy*|if *ra*==*rb* then *R*(rb) else *deliver*!*rm*.*sack*!not(*rb*).*R*(not(*rb*)){*ra*=true,*rb*=false,*rm*=data1,*sb*=true,*sm*=*data*1};

*S*2(*sb*,*sm*)|*Mlossy*|if *ra*==rb then *R*(*rb*) else *deliver*!*rm*.*sack*!not(*rb*).*R*(not(*rb*)){*ra*=true,*rb*=false,*rm*=data1,*sb*=true,*sm*=*data*1}::--*deliver*!*data*1->;

*S*2(*sb*,*sm*)|*Mlossy*|*sack*!not(*rb*).*R*(not(*rb*)){*rb*=false,*sb*=true,*sm*=*data*1}.

结果表明,ABP模型满足性质3,并且给出正例.正例是一条迁移路径,说明了ABP从接收数据包到将其送出的整个过程.其中data1是符号值,它表明可以用任意数据值替换正例中data1的出现.

4 结论和相关工作

本文讨论了传值进程模型检测中诊断信息的生成问题.在非传值的情况下,类似的工作已有文献作了介绍.文献[7]讨论了无交错情况下布尔方程求解中诊断信息的生成问题.本文的证明图在无传值和无交错情况下与文献[7]中的诊断(diagnostic)是一致的.但是在实际应用中,仅给出证明图并不直观,所以我们进一步讨论了如何得到更直观的示例.文献[8]讨论了对CTL公式进行检测时诊断信息的生成问题,其生成的示例往往是一条迁移路径,这与CTL公式的语义定义有密切联系.μ-演算的语义定义与CTL不同,且其表达能力更强.它的示例通常是一个迁移子图,而不是简单的路径.文献[9]讨论了用全局算法对μ-演算公式进行模型检测时,诊断信息的生成问题.此文献中生成的示例与本文中生成的示例在形式上是一致的.但是,在检测过程中如何存储信息则完全不同.在某些情况下,用 Kick 的方法得到的结果在节点数或边数上可能会更好,这是由全局算法的特点决定的,全局算法遍历了完全的状态空间.当然,这样也不可避免地会带来其他缺点,即状态空间不能动态生成,算法的空间效率较差.在诊断信息的生成方面,与以上方法均不相同的是Stevens在文献[10]中提出的一种基于games的交互诊断方法.该方法的原理是,在模型检测过程中由机器存储某一选手的必胜策略(history-free winning strategies),如果用户对最终的检测结果有疑问,可以找一条策略与机器博弈.由于机器掌握了所有情况下的对付办法,因此,用户注定失败.通过这样的手段让用户亲自体会问题之所在.这里,问题的关键是如何从多个后继状态中作出正确的选择,即对值为真的析取结点(或对值为假的合取的结点),如何选择合适的直接依赖状态.其实,本文中各状态的 *DDEP*(*s*) 就是文献[10]中存储的博弈策略,因此,检测工具完全可以利用证明图中的信息与用户进行博弈.

诊断信息的自动生成是模型检测方法的基本特征之一.国际上现有的检测工具,如 CADP,XMC,CWB-NC,CWB-ED 等,大多基于传统的标号迁移系统.在对问题进行求解时,事先要将所检测的系统翻译为相应的不带变量的LTS,然后才能在所生成的LTS上进行检测.这样的处理过程往往将原系统的结构信息丢失,使得到的诊断信息不可读.我们的方法由于直接在带赋值的标号迁移图和模态图上进行操作,无须经过这样的翻译步骤,这样就避免了翻译过程中信息的丢失,使生成的诊断信息更有利于人们理解和用于排错.

致谢 感谢Julian Bradfield和我们讨论有关用μ-演算公式来描述并发系统公平性性质的问题.

References:

[1] Clarke EM, Grumberg O, Peled DA. Model Checking. Cambridge, MA: MIT Press, 1999.

[2] Lin HM. Model checking value-passing processes. In: Proceedings of the 8th Asia-Pacific Software Engineering Conference. Macau: IEEE Press, 2001. 3~10.

[3] Lin HM. Symbolic transition graphs with assignment. In: Montanari U, Sassone V, eds. Proceedings of the CONCUR'96. LNCS 1119, Heidelberg: Springer-Verlag, 1996. 50~65.

[4] Stirling C, Walker D. Local model checking in the modal mu-calculus. Theoretical Computer Science, 1991,89(1):161~177.

[5] Bhat GS, Cleaveland R. Efficient local model checking for fragments of the modal calculus. In: Margaria T, Steffen B, eds. Proceedings of the TACAS'96. LNCS 1055, Heidelberg: Springer-Verlag, 1996. 107~126.

[6] Bradfield J, Stirling C. Modal logics and mu-calculi: an introduction. In: Bergstra JA, Ponse A, Smolka SA, eds. Handbook of Process Algebra. North-Holland: Elsevier, 2001. 293~332.

[7] Mateescu R. Efficient diagnostic generation for Boolean equation systems. In: Graf S, Schwartzbach MI, eds. Proceedings of the TACAS'2000. LNCS 1785, Heidelberg: Springer-Verlag, 2000. 251~265.

[8] Clarke EM, Grumberg O, McMillan KL, Zhao X. Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proceedings of the DAC'95. San Francisco: ACM Press, 1995. 427~432.

[9] Kick A. Tableaux and witnesses for the mu-calculus. Technical Report, TR44/95, Karlsruhe: Faculty of Computer Science, University of Karlsruhe, 1995.

[10] Stevens P, Stirling C. Practical model-checking using games. In: Steffen B, ed. Proceedings of the TACAS'98. LNCS 1384, Heidelberg: Springer-Verlag, 1998. 85~101.

附录：ABP 的输入检测文本

--Model Checking Alternating Bit Protocol with Data; lines starting with '--' are comments
type *message*=1,...,10 with *message*<Int
process *S*,*R*: Bool
*S*1,*S*2: Bool message
Mlossy:
predicate *Spec*:
X,*Y*,*Z*: message
channel accept, deliver: message
r,*s*: Bool message
rack,*sack*: Bool
variable *sb*,*sa*,*rb*,*ra*,*ma*: Bool
sm,*rm*,*mm*,*m*,*n*,*u*,*v*,*j*,*k*: message
conjecture (*S*(true)|*Mlossy*|*R*(false))\{*r*,*s*,*rack*,*sack*}|=*Spec*
where --*S*:Sender; *R*:Receiver; *Mlossy*: Media maybe losing data
S(*sb*)=*accept*?*sm*.*S*1(*sb*,*sm*) *S*1(*sb*,*sm*)=*s*!(*sb*,*sm*).*S*2(*sb*,*sm*)
*S*2(*sb*,*sm*)=*S*1(*sb*,*sm*) + rack?sa.(if *sa*==*sb* then *S*(not(sb)) else *S*1(*sb*,*sm*))
R(*rb*)=*sac*k!*rb*.*R*(*rb*)+*r*?(*ra*,*rm*).if *ra*==*rb* then *R*(*rb*) else *deliver*!*rm*.*sack*!(not(*rb*)).*R*(not(*rb*))
Mlossy=*s*?(*ma*,*mm*).(*r*!(*ma*,*mm*).*Mlossy*+*Mlossy*)+*sack*?*ma*.(*rack*!*ma*.*Mlossy*+*Mlossy*)
predicate equation
--propety 0(deadlock-free)
max *Spec*=(*E*_({*m*},⟨*deliver*!*m*⟩true) or ⟨*accept*?*n*⟩ true or ⟨*tau*⟩ true) and (*A*_({*u*},[*deliver*!*u*] *Spec*) and [*accept*?*v*] *Spec* and [*tau*] *Spec*)
--property 1(liveness without fainess constrain)
max *Spec*=[*accept*?*m*] *X*(*m*) and ([*tau*] *Spec* and [*accept*?*u*] *Spec*)
min *X*(*n*)=(⟨*tau*⟩ true or ⟨*deliver*!*n*⟩ *Spec*) and [*tau*]*X*(*n*)
--property 2(liveness under fairness constrain)
max *Spec*=[*accept*?*m*] *X*(*m*) and ([*tau*] *Spec* and [*accept*?*u*] *Spec*)
min *X*(*n*)=*Y*(*n*)
max *Y*(*j*)=*Z*(*j*)
min *Z*(*k*)=([*tau*] *X*(*k*) and (⟨*tau*⟩ true or ⟨*deliver*!*k*⟩ *Spec*)) or (([*tau*] *Y*(*k*) and (⟨*tau*⟩ true or ⟨*deliver*!*k*⟩ *Spec*)) and (⟨*deliver*!*k*⟩ *Spec* or ⟨*tau*⟩*Z*(*k*)))
--property 3(liveness over data independent domain)
max *Spec*=[*accept*?*m*]*X*(*m*)
min *X*(*n*)=⟨*deliver*!*n*⟩ *Spec* or ⟨*tau*⟩*X*(*n*)
end