

基于模型的故障树自动生成方法



展万里^{1,2} 胡 军^{1,2} 谷青范³ 荣 灏³ 祁 健^{1,2} 董彦宏^{1,2}

1 南京航空航天大学计算机科学与技术学院 南京 211106

2 软件新技术与产业化协同创新中心 南京 210007

3 中国航空无线电电子研究所 上海 200233

(xxzhlw@163.com)

摘 要 基于模型的安全性分析方法能够提高复杂安全关键系统的建模与分析能力。目前故障树被广泛应用于系统安全及可靠性分析中。故障树分析(Fault Tree Analysis, FTA)是一种自上而下的演绎式失效方法,根据故障树分析系统中不希望出现的状态,系统工程中可以尽早确定当前系统模型可能出现的问题并及时避免。面向一类安全关键性系统领域中的系统安全性建模语言 AltaRica,基于其语义模型卫士转换系统(Guarded Transition Systems, GTS),设计了从平展化的 GTS 模型自动构造系统故障树的方法,节省了人工构造故障树的时间,从而加快了系统分析的进度。根据 AltaRica3.0 语言的语义规则,提取平展化 GTS 模型的数据构建实例对象;设计了 GTS 模型划分算法,得到一组独立 GTS 模型与一个独立断言,通过邻接矩阵构建独立 GTS 的可达图并获取关键事件序列。最后将处理结束的独立 GTS 与独立断言相结合,通过断言传播算法得到整个系统的状态及关键事件序列,生成系统故障树。最后通过实例来检验算法的有效性,结果表明,该算法能有效完成从平展化 GTS 模型自动生成故障树。

关键词: 故障树分析; AltaRica; 平展化 GTS; 划分算法; 可达图; 断言传播算法

中图法分类号 TP311

Model-based Fault Tree Automatic Generation Method

ZHAN Wan-li^{1,2}, HU Jun^{1,2}, GU Qing-fan³, RONG Hao³, QI Jian^{1,2} and DONG Yan-hong^{1,2}

1 College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China

2 Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210007, China

3 China National Aeronautic Radio Electronics Research Institute, Shanghai 200233, China

Abstract Model-based safety analysis methods can improve the modeling and analysis capabilities of current complex safety-critical systems. At present, fault tree is widely used in system safety analysis and reliability analysis. Fault tree analysis (FTA) is a top-down deductive failure method, which analyzes undesired states in the system according to the fault tree. In the system engineering, the possible problems of the current system model can be determined as early as possible and avoided in time. The work of this paper is oriented to a type of system safety modeling language AltaRica in the aerospace field. Based on its semantic model GTS (guarded transition systems), a method for automatically constructing a system fault tree from the flattened GTS model is designed, which saves the time of artificial fault tree construction, and speeds up the progress of system analysis. According to the semantic rules of the AltaRica3.0 language, extracting the data of the flattened GTS model to construct instance objects, designing the GTS model division algorithm to obtain a set of independent GTS models and an independent assertion, constructing the reachability graph of the independent GTS through the adjacency matrix and obtaining the key event sequence, then, the independent GTS that has been processed is combined with the independent assertion, the state of the entire system and the sequence of key events are obtained through the assertion propagation algorithm, and the system fault tree is generated. Finally, an example system shows that the algorithm can effectively complete the automatic generation of fault trees from the flattened GTS model.

Keywords FTA, AltaRica, Flattened GTS, Division algorithm, Reachability graph, Assertion propagation algorithm

到稿日期:2020-08-27 返修日期:2020-10-29 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。

基金项目:国家重点基础研究发展计划(973 计划)(2014CB744900)

This work was supported by the National Basic Research Program of China(2014CB744900).

通信作者:胡军(hujun@nuaa.edu.cn)

1 引言

近年来,随着各类安全关键系统功能的不断增强和需求的日益增加,传统的系统工程开发和分析性技术受到了很大的挑战。基于模型的安全性分析^[1-2](Model Based Safety Analysis, MBSA)是近年来出现的用于对复杂系统建模并基于系统模型实现自动或半自动的安全分析及验证的一类技术方法。如何在基于模型的框架中有效地生成经典的系统安全性与可靠性分析制品(如故障树、FMEA 等)已经成为系统建模和分析领域的一个热点研究问题。

AltaRica^[3-5]是一种用于安全性分析的高级建模语言,具有较好的层次化系统结构和动态的描述机制,能提供基于故障的失效行为建模元素,并且具备形式化模型语义,因此在航空航天等安全关键领域逐渐得到广泛的应用。本文工作面向 AltaRica 的最新版本——AltaRica3.0,基于其语义模型卫士转换系统^[6-7](Guarded Transition Systems, GTS)设计了从平展化的 GTS 模型自动构造系统故障树的方法。卫士转换系统的工作包括:首先根据 AltaRica3.0 语言的语义规则提取平展化 GTS 模型的数据,构建实例对象;然后设计 GTS 模型划分算法,得到一组独立 GTS 模型与一组独立断言,构建独立 GTS 的可达图并获取关键事件序列;最后将处理结束的独立 GTS 与独立断言相结合,设计了断言传播算法,得到整个系统的状态及关键事件序列,从而生成系统故障树。之后便可使用 XFTA 等计算引擎评估生成的故障树,依靠强大的原始算法提取最小割集^[8-10],并且实现顶事件等常用可靠性指标的计算。利用这些分析故障树所得的各种可靠性指标可以帮助系统设计人员和分析人员轻松地发现当前系统模型中的问题所在,进而优化系统模型设计。

本文第 2 节讨论本算法研究中所涉及的技术的基本概念;第 3 节整理由 AltaRica3.0 的平展化 GTS 模型自动生成故障树的算法和思路;第 4 节详细描述算法思想及伪代码;第 5 节通过实例来检验算法的有效性,并对得到的结果进行评估;第 6 节对本文工作进行总结并讨论未来的工作方向。

2 GTS 模型与故障树概述

GTS 模型是 AltaRica 语言建模平展化之后的一种语义模型。故障树是目前系统分析领域着重使用的一项技术。本文以 GTS 模型为算法输入,以故障树为算法输出,将两种技术共同应用在算法中,本节将详细介绍这两种技术的相关概念。

2.1 GTS 概述

本文以基于 AltaRica3.0 语言建立的平展化 GTS 语义模型为输入模型,对 GTS 模型进行读取、分析、处理,最终得到系统的故障树模型。平展化的 GTS 的形式化定义^[11-12]如下。

定义 1 GTS 由一个五元组 $\langle V, E, T, A, \iota \rangle$ 组成,其中:

- (1) V 是一组变量。 V 是状态变量集 S 和流式变量集 F 的不相交并, $V = S \cup F$ 。
- (2) E 是一组事件。
- (3) T 是一组转换。转换又被表示为三元组 $\langle e, G, P \rangle$ 。其中 e 是事件, G 是建立在变量 V 上的布尔表达式, P 是建立

在变量 V 上的指令。为了清楚起见,将转换 $\langle e, G, P \rangle$ 写成 $e: G \rightarrow P$ 。

- (4) A 是一组断言。断言即建立在变量 V 上的指令。
- (5) ι 是变量 V 的赋值。

在给定状态下,即变量赋值 σ ,如果 $\sigma(G) = \text{真}$,则转换 $e: G \rightarrow P$ 是可实现的。转换 $e: G \rightarrow P$ 的确定过程分为两步:1)状态变量更新;2)如前所述传播变量的值。

GTS 是 Kripke 结构^[13]的隐式表示,结点由变量赋值标记,边由事件标记。更确切地说,卫士转换系统的正则语义是最小的 Kripke 结构,其中 Σ 是一组状态,而 Θ 是一组转换,则:

$\sigma_0 = \text{Propagate}(A, \iota, \iota) \in \Sigma$, σ_0 是当前 Kripke 结构中的初始状态。

如果 $\sigma \in \Sigma$,而且 T 包含一个转换 $e: G \rightarrow P$,并且该转换可以在 σ 中执行,则 $\tau = \text{Fire}(P, A, \iota, \sigma) \in \Sigma$ and $\sigma \rightarrow \tau \in \Theta$ 。

GTS 的组成部分如图 1 所示。

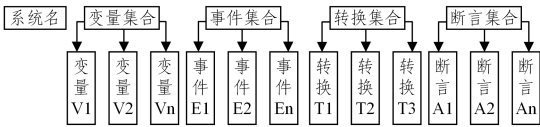


图 1 GTS 的组成部分

Fig. 1 Composition of GTS

图 2 给出了航电系统中一个简单制冷系统的模型图示。

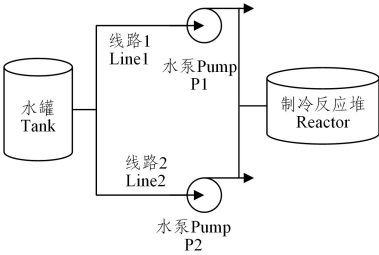


图 2 简单制冷系统

Fig. 2 Simple cooling system

在简单制冷系统中,由水罐 Tank 流出的冷却水作为系统后续的输入源,通过线路 1 或者线路 2 分别到达水泵 P1 和水泵 P2,最后流入制冷反应罐 Reactor 中,此为正常情况下的系统。其中组件包含水罐和水泵两类,水泵属于可修复性组件,整个系统由水罐、两条线路和冷却系统组成,两条线路中包含水泵 P1 和水泵 P2。系统中还添加了观察者变量,用来对制冷反应堆的正常启用状态做标记,整个系统包含事件 $evPumpsCCF$ 及其对应的转换,以及一个断言。简单制冷系统的 AltaRica3.0 建模语言所得到的模型组件包含的内容如下:

```
class Pump
    extends RepairableComponent;
    Boolean vfInFlow, vfOutFlow (reset=false);
    assertion
        vfOutFlow := if vsWorking then vfInFlow else false;
end

class Tank
    Boolean vsIsEmpty (init=false);
    Boolean vfOutFlow (reset=true);
```

```
event evGetEmpty;
transition
evGetEmpty;not vsIsEmpty -> vsIsEmpty:=true;
assertion
vfOutFlow:=not vsIsEmpty;
end
其系统模型定义如下:
block CoolingSystem
Tank TK (evGetEmpty, delay=Dirac(0,0));
block Line1
embeds main. TK as t1;
Pump P;
assertion
P, vfInFlow:=t1, vfOutFlow;
end
block Line2
embeds main. TK as t2;
Pump P;
assertion
P, vfInFlow:=t2, vfOutFlow;
end
block Reactor
Boolean vfInFlow (reset=false);
end
observer Boolean oCooledReactor=Reactor, vfInFlow;
parameter Real pPumpsCCF=1.0×10-6;
event evPumpsCCF (delay=exponential(pPumpsCCF));
transition
evPumpsCCF;! Line1, P, evFailure & ! Line2, P, evFailure;
assertion
Reactor, vfInFlow:=Line1, P, vfOutFlow or Line2, P, vfOut-
Flow;
end
简单制冷系统的 AltaRica 3.0 模型对应的平展化 GTS
模型包含的内容如下:
```

```
domain RepairableState {WORKING,FAILED}
block CoolingSystem
Boolean T, isEmpty(init=false);
Boolean T, outFlow (reset=true);
RepairableState Line1, P1, s (init=WORKING);
RepairableState Line2, P2, s (init=WORKING);
Boolean Reactor, inFlow (reset=false);
Boolean Line1, P1, inFlow (reset=false);
Boolean Line2, P2, inFlow (reset=false);
Boolean Line1, P1, outFlow (reset=false);
Boolean Line2, P2, outFlow (reset=false);
event T, getEmpty;
event Line1, P1, repair;
event Line2, P2, repair;
event Line1, P1, failure;
event Line2, P2, failure;
transition
Line1, P1, failure;Line1, P1, s=WORKING -> Line1, P1, s:=
FAILED;
```

```
Line1, P1, repair;Line1, P1, s=FAILED -> Line1, P1, s:=
WORKING;
Line2, P2, failure;Line2, P2, s=WORKING -> Line2, P2, s:=
FAILED;
Line2, P2, repair;Line2, P2, s=FAILED -> Line2, P2, s:=
WORKING;
T, getEmpty;not T, isEmpty -> T, isEmpty:=true;
assertion
Line1, P1, outFlow:=Line1, P1, s=WORKING and Line1, P1,
inFlow;
Line1, P1, inFlow:=T, outFlow;
Line2, P2, outFlow:=Line2, P2, s=WORKING and Line2, P2,
inFlow;
Line2, P2, inFlow:=T, outFlow;T, outFlow:=not T, isEmpty;
Reactor, inFlow:=Line1, P1, outFlow or Line2, P2, outFlow;
end
```

2.2 故障树

故障树分析是安全关键系统设计过程中进行安全性、可靠性分析的重要方法^[14-16]。故障树的构造通常是由专门的系统安全分析人员进行,即在系统设计过程中,综合考虑系统设计方案、各种失效场景以及各类故障模式来人工设计出故障树模型,然后展开最小割集、顶事件发生概率、各基本事件概率等各种分析,确定导致系统失效的各种组合方式以及失效的概率,找出导致系统失效的关键点,并据此设计合理的纠正措施以保障系统最终设计得安全可靠。一个典型的故障树模型如图 3 所示,具体的各类逻辑门的定义可参见文献^[17]。

近年来,随着安全关键系统的功能越发复杂,单纯地依赖安全分析人员的工程经验来构造复杂系统的故障树已经不能满足可靠性和安全性规范的要求。因此在基于模型的系统工程(Model-Based Systems Engineering, MBSE)^[18] 框架下,如何基于系统架构设计模型自动生成相应的故障树,已经成为目前一个重要的研究热点。

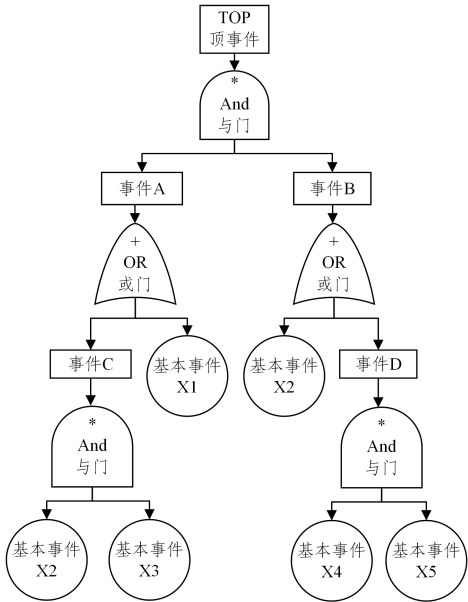


图 3 简单故障树模型
Fig. 3 Simple fault tree model

3 基于平展化 GTS 模型的故障树自动生成方法框架

本节总体上给出了如何面向 AltaRica3.0 的平展化 GTS 模型来自动生成故障树(GTS to Fault Tree,GTS2FTA)的方法框架以及所包含的步骤,然后第 4 节中将详细说明每个步骤所涉及的算法设计。

如图 4 所示,GTS2FTA 方法框架总体上分为以下 5 个主要步骤:

- (1)读取平展化的 GTS 模型描述文件,构造得到对应的 GTS 对象;
- (2)对输入的 GTS 对象划分得到一组独立 GTS 与一个独立断言;
- (3)对独立 GTS 集合进行迭代处理,获取每个独立 GTS 对应的可达图,方便后续获取布尔公式;
- (4)对多个独立 GTS 的可达图进行迭代操作,获取对应的多组布尔表达式集合;
- (5)将划分所得的独立断言与各个独立 GTS 可达图编译所得的布尔公式进行同步处理,获得整个模型的布尔公式集合,即为这个 GTS 模型的故障树组成。

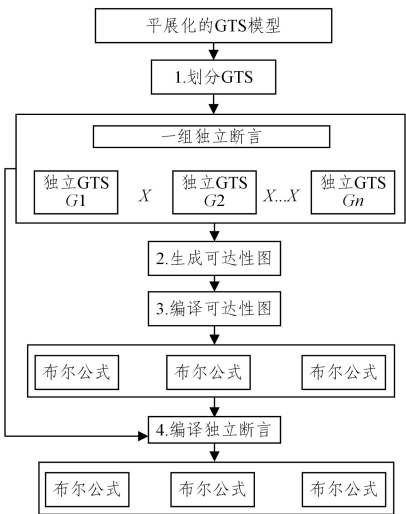


图 4 平展化 GTS 自动生成故障树算法框架
Fig. 4 Framework of automatic fault tree generation algorithm

下面对每个步骤作进一步的说明:

步骤 1 读取平展化的 GTS 模型描述文件,并解析得到 GTS 实体对象。OpenAltaRica 等工具直接编译 AltaRica 模型生成的 XML 格式的平展化 GTS,之后进行生成故障树、模型分析和仿真等工作。若系统分析阶段出现问题,则难以直接根据故障树来源——平展化 GTS 模型找出问题所在。而本文中的平展化 GTS 文件是采用文献[19]中基于 ANTLR (Another Tool for Language Recognition)将 AltaRica3.0 模型的 S2ML(System Structure Modeling Language)转化为语义等价的平展化 GTS 文件的方法,这样能够得到更加直观的 GTS 语义模型,从而增强平展化 GTS 的可读性。在系统分析阶段如果出现问题可以直接针对平展化 GTS 进行分析和修改。在这个平展化的过程中,一个高度复杂的系统模

型将被压缩,各个系统内容可以在同一个维度考虑,极大地简化了原有的复杂模型。

这个平展化的 GTS 模型描述文件的简单实例如图 5 所示。该描述文件的内容由 GTS 的模型名称、内含的变量集合、事件集合、转换集合、断言集合组成。根据这个描述文件,将该 GTS 中的所有信息一并取出并根据定义的 GTS 数据结构进行存储,创建一个 GTS 的实例对象。

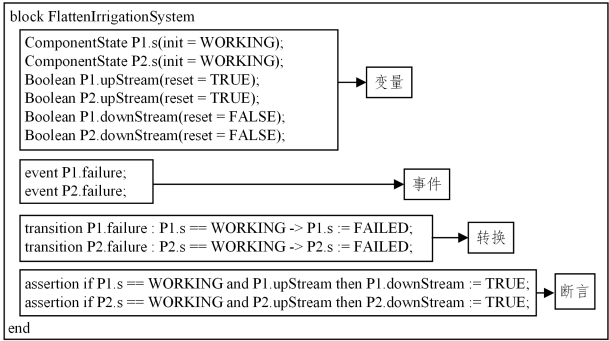


图 5 一个简单的 GTS 模型描述文件
Fig. 5 A simple GTS model description file

步骤 2 对 GTS 实例对象的划分操作。考虑到系统模型的规模较大,在得到 GTS 实例对象后,为了简化后续步骤,此处采用较为高效的划分算法(详见 4.1 节)来提升算法效率,以应对系统模型较为庞大的情况。

划分算法将一个模型划分为多个组件与模块,之后对各个组件与模块进行单独处理,在最终获取故障树时结合数据结果一并处理,这将简化整个算法的中间步骤。在具体的划分算法中,主要通过变量之间的依赖关系来划分独立 GTS 与独立断言。变量间的依赖关系可由断言和转换获取,根据变量间的依赖关系,将相关变量、事件、转换、断言放在一起作为独立 GTS,剩下的部分组成独立断言。步骤 1 和步骤 2 的具体算法对应于 4.1 节中的 GTS 划分算法。

步骤 3 求取各个独立 GTS 对应的可达图。这一步主要针对各个独立 GTS 进行处理,获得相关的可达图。一个可达图包含一个结点集合,而一个结点是以一个变量集合的形式存在,用当前各个变量的值表示当前系统状态。步骤 3 的具体算法对应于 4.2 节中子 GTS 模型的可达性图构造。

可达图的示例效果如图 6 所示。

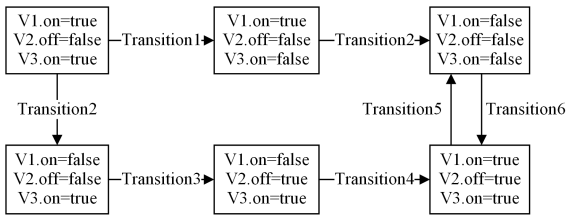


图 6 一个 GTS 的可达图
Fig. 6 Reachability graph of a GTS

步骤 4 编译各个可达图并获取独立 GTS 的布尔公式,此步骤也被称为获取关键事件序列。由于最终算法需要获得整个故障树,因此必须获取布尔公式,并且本算法利用了划分算法来解决大型系统模型导致计算复杂度呈指数级增长的问题。

题。在上一步骤中算法得到了关于各个 GTS 的 GTS 可达图。获取布尔公式,简而言之就是在这个可达图中找到两个结点之间所有可能的路径。步骤 4 的具体算法对应于 4.3 节中子 GTS 模型可达图的布尔公式生成。

在生成关键事件序列的步骤中,运算符“&&”被定义为序列的“与”门,运算符“||”被定义为序列的“或”门,运算符“Q”表示序列集合。“与”门运算符将进一步标注为:搜索从状态 S0 到图的其他状态的所有路径。沿路径 π 发生的事件被转换为事件的“与”关系。先将图的每个状态与通过编译路径获得的序列析取相关联。然后将每对(变量/值)与序列的析取相关联,其中变量采用此时经过触发事件所得到的值,如触发事件并未修改该变量的值,则取原值。输出形式如下:

$$\begin{aligned} \text{QS0} &= \overline{CCF} \wedge \overline{W1.failure} \wedge \overline{W2.failure} \\ \text{QS1} &= \overline{CCF} \wedge W1.failure \wedge \overline{W2.failure} \\ \text{QS2} &= \overline{CCF} \wedge \overline{W1.failure} \wedge W2.failure \\ \text{QS3} &= CCF \wedge W1.failure \wedge W2.failure \end{aligned}$$

步骤 5 编译独立断言并与各个独立 GTS 的关键事件序列组合处理,得到故障树的关键事件序列。

在步骤 4 中已经得到了所有独立 GTS 的关键事件序列,这里为方便后续处理先对这些关键事件序列遍历,获取每一个独立 GTS 中包含的变量在取不同值时所对应的关键事件序列,内容如下所示:

```
[V1.s,WORKING]:[null,[V1.failure],[V1.close]]
[V2.isClosed,FALSE]:[[V2.open],[V2.open&&V2.failure]]
[V2.isClosed,TRUE]:[null,[V2.failure],[V2.open]]
[V3.s,FAILED]:[[V3.failure],[V3.open&&V3.failure]]
```

由于国外在 AltaRica 相关工具上进行了技术保密,我们无法掌握一些核心技术和工作原理,在编译独立断言的步骤中更是如此。上述划分步骤中对系统进行划分得到了独立的系统模块,并未考虑模块间的关系。编译独立断言这一步骤对独立的系统模块的状态集合求取笛卡尔积,进而得到系统的状态全集。之后以独立断言作为系统约束,通过传播断言对上述划分好的独立系统模块进行关系约束,从而保证整个系统的可靠性和完整性。最终利用布尔公式获取事件发生序列,并生成故障树,保证故障树与系统的一致性。步骤 5 的具体算法对应于 4.4 节中 GTS 模型的完整布尔公式生成。

4 GTS2FTA 框架中的核心算法集

本节将详细给出 GTS2FTA 方法框架中每个步骤所设计的算法集,主要包括 GTS 模型的预处理与划分、子 GTS 模型的可达图构造、子可达图的布尔公式生成、完整布尔公式生成等。

4.1 GTS 模型的预处理与划分

算法 1 GTS 划分算法伪代码

Input:GTS Class
Output:Independent GTS and Independent Assertion

```
1.function Partitioning(GTS);
2.List<Transition>transitionList ← Q;
3.List<Var>vars ← Q;
4.List<Event>events ← Q;
5.List<GTS>independentGTS ← Q;
6.List<GTS>independentAssertion ← Q;
7.    function ProcessTransition(transitionList)
8.        t∈transitionList events.add(t)
9.        v∈t vars.add(v)
10.       if (v depend on vtemp) then
11.           vars.add(vtemp)
12.       function getIntersection(vars);
13.       gTemp.setVars
14.       gTemp.setEvents
15.       gTemp.setTransitions
16.       independentGTS.add(gTemp)
17.       independentAssertion.add(allAssertions)
18.   return List<GTS>;
19.end function
```

在自动生成故障树算法中,由于 GTS 文件本身所含信息较多,首先对数据结构进行定义,本算法中共定义了多种基本数据结构,包括断言、布尔表达式、事件、卫士、指令、后继、转换、变量、状态结点、权值等。

断言 (Assertion)定义了一种关系约束,如果当前关系满足断言,必然会发生后续指令。

事件(Event)定义了一个系统允许发生的事件,并且可能有定时、延时等参数。

转换(Transition)是对事件的补充,一般由卫士、指令组成。

卫士(Guard)为一种条件,当满足条件之后方可执行后续的指令(Instruction)。

状态结点(Node)、权值(Weight)用于后续获取可达性图。

卫士转换系统是由上述数据结构共同组成,对于整个系统来说,系统内部的一些事件被触发时会由转换进行处理,如果不能满足转换中的卫士条件将无法正常完成该事件。

在算法 1 的第一步中,要先对扁平化的 GTS 模型描述文件进行读取分析,并取出相应的数据填充 GTS 的模型。由于 AltaRica3.0 的语法严格,本文采用了字符串处理方式提取文件数据,并为各个数据实例化对象、填充数据,取得所有数据之后赋予 GTS 实例对象,即当前程序中 GTS 的对象。

得到实例化 GTS 对象之后开始第二步的划分处理,将给定的 GTS 划分为一个或多个独立的 GTS 和独立的断言。在划分算法中,划分模块和组件的重要依据是变量之间的依赖关系,即一个变量 Var 1 的赋值若取决于其他变量 Var n,则 Var 1 是依赖变量,Var n 是被依赖变量,Var 1 依赖于 Var n。因此划分算法中的第一步就是处理 GTS 实例,得到其中所有变量的依赖关系。在 GTS 中,状态变量一次性全部被初始化,初始化的值由初始化属性给予。流式变量则在每个转换开启之后被更新,它们的值也被断言或者其重置值所修改。

令 $G=V,E,T,A,\iota$ 为 GTS 并且令 $t=e,g,p$ 为 transition 之一,此处 transition 中的 e 代指 *event* 事件, g 代指 *guard* 卫士(判断依据), p 代指 *instruction* 指令(结果赋值)。我们用变量 $var(t)$ 表示在 *transition* 中涉及的变量。

$$var(t)=var(g)\cup var(p)\cup V'$$

其中, V' 是变量,因此 $var(g)\cup var(p)$ 的变量通过断言 A 依赖于它们。

$Var(t)$:Transition 中的变量集合,由 $Var(g)$ 与 $Var(p)$ 组成。

$Var(g)$:Guard 中的变量集合,可以看作条件语句中涉及的变量集合。

$Var(p)$:Instruct 中的变量集合,可以看作指令语句中涉及的变量集合。

定义 2 $V'=\{v\in V;\exists u\in var(g)\cup var(p)|u\text{ depends on }v\text{ in }A\}$

根据 V' 的定义可知, V' 中的变量 v 属于全体变量集合,但是有变量 u 属于 $Var(g)$ 与 $Var(p)$ 的并集,且变量 u 在断言中会依赖于 v 。对整个 GTS 变量进行划分之后便可涉及该变量的断言和转换,事件被划分到不同的 GTS 中,剩下的部分全部归为独立断言。至此可得到一组独立 GTS 的集合与一组独立断言。

该部分算法的复杂度分析如下:从时间复杂度来看,划分算法的模块算法首先针对变量依赖关系进行处理,之后预处理每一个 Transition,最后完成独立 GTS 的划分。在变量依赖关系的处理中,断言集合的循环为第一层循环,由于每一条断言中指令的个数不确定,需要开始嵌套第二层的指令循环,找出变量与变量之间的关系。获取变量处于第三层的循环嵌套,假设断言集合由 x 个断言组成,而每个断言的指令集又由 y 个指令组成,当前变量集合由 z 个变量组成,因此此模块的时间复杂度为 $O(n^{(x*y*z)})$ 。

在预处理模块中,需要遍历 Transition 集合,取出变量,此处同样为 3 层循环的嵌套结构,与获取变量依赖关系处于同一复杂度量级。而在划分模块中,算法对预处理所得的 Transition 辅助存储集合进行遍历,之后嵌套再对 Transition 临时存储子集合进行遍历,从而获取当前 Transition 所对应的事件。假设以上各个集合中存放着 a,b,c 3 个元素,那么此模块的时间复杂度为 $O(n^{(a*b*c)})$ 。因此,划分算法的最终时间复杂度是 $O(n^{(x*y*z)})$ 。

从空间复杂度来看,划分算法以变量集合、事件集合、转换集合与断言集合为主,该算法的占用空间为一个数组空间,空间复杂度为 $O(n)$ 。

划分好独立 GTS 与独立断言之后,我们得到了一组独立 GTS 的集合与一个独立断言,如图 7 所示。

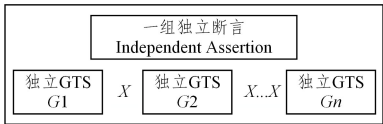


图 7 独立 GTS 集合与独立断言

Fig. 7 Independent GTS set and independent assertion

4.2 子 GTS 模型的可达图构造

算法 2 编译获取独立 GTS 的可达图算法和伪代码

Input:Independent GTS

Output:Reachability graph

```
1. function GetGraph(IndepentGTS);
2. List<Transition>list ← GTS.getTranstions();
3. List<Var>s0 ← GTS.getVars();
4. List<List<Var>>graph ← ∅;
5. function ProcessStart(s0)
6.     graph.add(s0)
7.     function ProcessTransitions(list)
8.         t∈list;
9.         if (s0 satisfy t) then
10.             function ProcessVars(s0)
11.                 return ProcessStart(s1)
12.             return graph;
13. end function
```

接下来对算法 1 中划分所得的每一个独立 GTS 求可达图。假设每个独立的划分 $G=V,E,T,A,\iota$ 描述了可能失效的系统,图 $\Gamma=(\Sigma,\Theta)$ 是 G 的可达图。初始状态或初始变量赋值 $\sigma_0\in\Sigma$ 表示系统的标称状态,来自 E 的事件表示系统组件的故障,一些状态(变量赋值) $\sigma\in\Sigma$ 表示失效状态,从 σ_0 到这些状态的路径表示失效的情况, Σ 代表状态结点集合, Θ 代表 Transition 转换集合。Transition 的组成部分大致分为条件语句与指令语句。条件语句代表该 Transition 要满足的条件,而指令语句代表若满足该 Transition 的条件则需要对某些变量进行修改。

在图论中,可达性指在图中从一个顶点到另一个顶点的容易程度。在无向图中,可以通过识别图的连接分量来确定所有顶点对之间的可达性。

本算法中的可达图信息如下。结点中包含着该 GTS 的所有变量,表示当前系统状态,采用 Transition 代表边。当一个状态发生某个 Event 事件时会触发 Transition,从而导致变量的值发生变化,转入下一个状态。算法中用一个变量集合代表一个状态结点,每一个独立 GTS 的所有变量组成了这个 GTS 可达图的初始结点 S_0 。将 S_0 加入到这个状态结点集合中,开始准备处理。在处理每一个结点之前,都要先判断该结点是否已经被处理过了,以免重复处理,之后对集合中的所有结点进行处理。每一个结点都是一组变量,并已经赋予了初值。算法利用每一组变量去遍历当前 GTS 中的所有转换事件 Transition,若该结点满足当前 Transition T_1 的布尔表达式 Guard,则会通过指令 Instruct 为依赖变量赋予新值,进而产生新的变量集合。这个新的变量集合即新的状态结点,将其加入到状态结点集合中,直至遍历完所有状态结点,将这些结点都加入集合,并将当前 GTS 实例对象的 Graph 属性赋值为该集合。

下面对该部分算法进行复杂度分析:

时间复杂度分析。算法要对每一个独立 GTS 的状态结点集合循环遍历,两者是嵌套关系。针对每一个结点集合中

的结点,又需要遍历当前独立 GTS 的 Transition 集合,原因是处理 Transition 的过程中需要处理布尔表达式和指令集。一个 Transition 中可能存在多个布尔表达式和多个指令,这一过程中需要嵌套对布尔表达式和指令集的循环。因此假设以上各个集合中的元素个数分别为 x, y, z , 即可以得到此模块算法的复杂度为 $O(n^{(x * y * z)})$ 。

空间复杂度分析。获取可达图算法以独立 GTS 集合、变量集合、事件集合和转换集合为主,它们在空间上为一个数组空间,空间复杂度为 $O(n)$ 。

4.3 子 GTS 模型可达图的布尔公式生成

算法 3 编译独立 GTS 的可达图获取布尔公式的算法和伪代码

```
Input: Reachability graph
Output: Boolean formula

1. function ComGraph(Graph);
2. List<Transition>list ← g. getTranstions();
3. List<Node>nodes ← g. getNodes();
4. Stack<Integer>stack;
5. Map<Integer, Boolean>states;
6. function ProcessNodes(nodes)
7. function ConstructingMatrix(nodes)
8.   while(!stack.isEmpty())
9.     topNode←stack. peek();
10.    if(topNode==y)
11.      printPath();
12.      adjvexNode=stack. pop();
13.      states.put(adjvexNode,false);
14.    else
15.      nextNode=getNextNode(graph,topNode,adjvexNode);
16.      if(nextNode!=−1)
17.        stack.push(nextNode);
18.        states.put(nextNode,true);
19.        adjvexNode=−1;
20.      else
21.        adjvexNode=stack. pop();
22.        states.put(adjvexNode,false);
23. end function
```

在算法 1 所得的可达图中,由于该可达性图只是一组变量结点的集合,需要对这个集合进行处理,获得结点和边的数量,构建结点数组和权值数组,并利用邻接矩阵的形式将集合中的数据填充到可达图中。可达图中结点代表系统状态,边代表事件,从某一状态发生某一事件会产生新的状态,这个过程可以用布尔公式来表示。以状态结点名作为图中结点名,以转换事件 Transition 名作为图中的权值边,组成可达图邻接矩阵,空结点在矩阵中对应的值为 Null。上一步中,算法在获取可达图结点集合的同时,也采用了边结点集合的方式来存储可达图信息,这里先对其进行处理。边结点集合中以边为结点,即以 Transition 转换事件作为结点属性,并分别设 InNode 和 OutNode 两个属性存放该转换事件可能的入度和出度。

结点集合经过初步处理,得到构建好的邻接矩阵,开始编译生成布尔公式。如果两个结点之间没有边,则权值为 Null,并且先用集合存放所有结点。此处采用栈的形式来分步处理,分为结点栈与路径栈,结点栈用来存放当前 GTS 处理过程中的结点,并且处理过程中采用一个 HashMap 来代表某结点是否还在栈中,避免产生回路。

处理过程中,最初所有结点都要初始化为未访问,之后定义栈顶元素 Top 和 adjvex_node 元素,adjvex_node 元素主要用于存放 Top 元素上次访问过的邻接点。之后开始入栈,并修改结点的访问标志位。当栈不为空时,先取栈顶元素 Top,判断其是否为终止遍历的目标结点。如果是目标结点,则意味着当前从初始结点到终止结点的路径被找到,即可利用路径栈的弹栈输出这条路径。在栈顶元素被弹出之前,将栈顶元素重新设置为未访问,这一步是为了找出之前栈顶元素未通过的路径。如果当前结点不是终止结点,则需要找到邻接结点,并且该邻接结点不应被访问过。若存在邻接结点,此时将其入栈,并将对应的边也入栈,设置该结点访问,重置邻接点。如果无其他邻接点,则直接弹出当前结点,并将路径栈全部弹出。如此遍历完所有结点集合,即可输出路径集合作为该路径的布尔公式。之后将各个路径的布尔表达式存入其独立 GTS 的路径集合中,以便后续获取整个系统故障树使用。

下面对该部分算法进行复杂度分析:

时间复杂度分析。得到可达图之后要将其编译为布尔公式,这一步中,算法同样要先对独立 GTS 的集合进行遍历,之后构建可达图的邻接矩阵,还要嵌套对状态结点集合的循环。另外要通过获取所有可达性路径的方法来获取布尔公式,该方法中包含了对当前栈的遍历。因此,假设独立 GTS 集合元素个数为 a , 每一个状态结点集合的元素个数为 b , 栈中元素个数为 c , 则当前算法的时间复杂度为 $O(n^{(a * b * c)})$ 。

空间复杂度分析。编译可达图算法以独立 GTS 集合、变量集合、事件集合、转换集合和结点集合为主,它们在空间上为一个数组空间,空间复杂度为 $O(n)$ 。

4.4 GTS 模型的完整布尔公式生成

算法 4 编译独立断言获取故障树的算法和伪代码

```
Input: Independent Asseriton, Independent GTS
Output: Boolean formula

1. function ComAssertion(GTS)
2. List<Var>vars ← Asseriton. getVars();
3. List<Arraylist<Var>>varsList←∅;
4.   function GetCartesianProduct(GTS);
5.   function GetPathIntersection(Product);
6.   function ProcessTransition(Asseriton)
7.     if (vars satisfy assertion) then
8.       printPath();
9.       return newVars;
10.    varsList.add(newVars);
11. end function
```

最后对独立断言进行编译。将上述可达图中的状态结点集合与其他独立 GTS 的状态结点集合取笛卡尔积(意味着所

有未涉及独立断言中变量的其他变量在经过各种事件以后会得到状态总集合)。在获取笛卡尔积之后,对每组笛卡尔积,求出该组变量集合中所有路径的交集,也可以获得当前变量状态集合需要经过的 Transition 布尔表达式,这里需要用到算法 3 所得的独立 GTS 可达性图的路径的布尔表达式结果来共同求取。为了保证所有独立 GTS(整个系统)可以同时获取一个状态,必须要满足:各个独立 GTS 的路径布尔表达式可以取得交集,若交集不为空,意味着这些独立 GTS 可以通过同一个关键事件序列触发。接着向每组笛卡尔积结果注入独立断言所包含的变量(此时这些变量均应为初始值),便可得到独立断言编译中所需要的所有完整变量集合。

然后,对上述变量集合中的每一组变量遍历独立断言,这就是独立断言的传播。为了简化处理,在遍历独立断言时,先判断独立断言中的依赖变量和被依赖变量是否均处于该组变量集合中。若不存在,则意味着该独立断言不受任何影响,直接跳过;若存在,则要进行处理。

在这个处理过程中,先判断当前变量集合所有的变量状态是否满足独立断言的判断条件,若满足,则依赖变量将获取一些值,这些值可能等于原值,也可能是新值。当为原值时,不对变量做任何处理;当为新值时,对原变量的值进行修改,以便下一个独立断言进行处理。重复以上步骤直至遍历完所有的独立断言,这标志着该组变量的集合已经做完了断言传播,即可产生新的变量集合,而这个变量集合即当前变量状态最终所得结果,之后输出布尔表达式,可以得到关键事件序列。当遍历完所有笛卡尔积结果,将获得整个系统所有变量的状态集合,并且每个结果都有一个对应的布尔表达式作为故障路径,即生成故障树。

下面对该部分算法进行复杂度分析:

时间复杂度分析。编译独立断言中,首先要对之前划分好的 GTS 集合进行遍历,并遍历内部的状态结点集合以及变量集合,求取笛卡尔积,之后通过断言传播遍历断言集合,得到最终表达式。递归求笛卡尔积算法的时间复杂度为 $O(n * O(f(n)))$,其中 n 代表循环次数, $f(n)$ 表示 `descartes()` 方法的复杂度,其递归结束的位置在递归集合 `dimvalue` 中的元素个数为 `layer+1`。因此该独立断言编译算法的时间复杂度也为 $O(n * O(f(n)))$ 。

空间复杂度分析。编译独立断言算法以独立 GTS 集合、变量集合、事件集合、转换集合和结点集合为主,它们在空间上为一个数组空间,空间复杂度为 $O(n)$ 。由于求取笛卡尔积需要利用递归算法,若递归深度层数为 N ,则空间复杂度为 $O(N)$ 。

5 实例分析

本节通过一个实例来对算法的有效性进行验证。文中采用的实例模型是一个电力供应系统,基于其相应的 AltaRica 3.0 平展化 GTS 模型进行分析,根据上述算法框架实现了自动生成故障树。

5.1 实例描述和环境

文献[20]采用了简单水供给系统进行实验分析,而本文

实验采用了更为复杂且在航电系统中更常见的电力供应系统(Power Supply System,PSS)来验证自动生成故障树算法。

对于电力供应系统,系统输入源为电源 R1 和 R2,接受源为组件 F1,F2,F3。与 R1,R2 直接相连的是两个电泵 P1 和 P2。两个电泵从电源获取电力,经过传输线导向组件。在传输线之间穿插有 3 个断路器,用来控制电力供应。实例如图 8 所示。

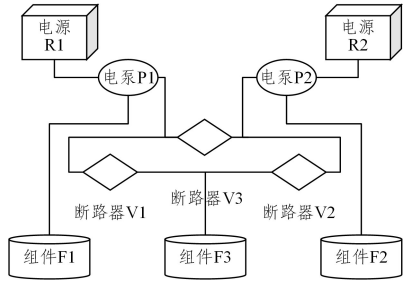


图 8 电力供应系统

Fig. 8 Power supply system

本实验运用基于 AltaRica 3.0 语言建立的平展化模型 GTS 的自动生成故障树算法对电力供应系统的模型进行自动生成故障树的操作。本文所涉及的实验数据均在一台 CPU 为 Intel(R) Core(TM) i5-9300H CPU @ 2.40 GHz、内存为 16 GB、操作系统为 Windows 10 专业版 1909、版本 18363.720 的计算机上得到,算法利用 Java 语言实现。

系统中所涉及的变量由 2 个电源、2 个电泵、3 个断路器以及 3 个组件的各变量组成,并且都设定有默认值,其中有 5 个状态变量,16 个流式变量。相关转换事件共 12 个,断言 21 个。利用电力供应系统的 Altarica3.0 模型文件和平展化 GTS 模型描述文件作为本实验的输入文件,内容如下所示:

```
block FlattenPowerSupplySystem
ComponentState V1. s(init=WORKING);
ComponentState V2. s(init=WORKING);
ComponentState V3. s(init=WORKING);
ComponentState P1. s(init=WORKING);
ComponentState P2. s(init=WORKING);
Boolean V3. isClosed(init=TRUE);
Boolean V2. isClosed(init=TRUE);
Boolean V1. isClosed(init=FALSE);
Boolean P1. upStream(reset=TRUE);
Boolean P2. upStream(reset=TRUE);
Boolean P1. downStream(reset=FALSE);
Boolean P2. downStream(reset=FALSE);
Boolean F1. inStream(reset=FALSE);
Boolean F2. inStream(reset=FALSE);
Boolean F3. inStream(reset=FALSE);
Boolean V1. leftStream(reset=FALSE);
Boolean V2. leftStream(reset=FALSE);
Boolean V3. leftStream(reset=FALSE);
Boolean V1. rightStream(reset=FALSE);
```

```

Boolean V2, rightStream(reset=FALSE);
Boolean V3, rightStream(reset=FALSE);
Event
V1, open; V2, open; V3, open; V1, close;
V2, close; V3, close; V1, failure; V2, failure;
V3, failure; P1, failure; P2, failure; ccf;
Transition
V1, open; V1, isClosed and V1, s = WORKING → V1,
isClosed := FALSE;
V2, open; V2, isClosed and V2, s = WORKING → V2,
isClosed := FALSE;
V3, open; V3, isClosed and V3, s = WORKING → V3,
isClosed := FALSE;
V1, close; not V1, isClosed and V1, s = WORKING →
V1, isClosed := TRUE;
V2, close; not V2, isClosed and V2, s = WORKING →
V2, isClosed := TRUE;
V3, close; not V3, isClosed and V3, s = WORKING →
V3, isClosed := TRUE;
V1, failure; V1, s = WORKING → V1, s := FAILED;
V2, failure; V2, s = WORKING → V2, s := FAILED;
V3, failure; V3, s = WORKING → V3, s := FAILED;
P1, failure; P1, s = WORKING → P1, s := FAILED;
P2, failure; P2, s = WORKING → P2, s := FAILED;
ccf; P1, s = WORKING or P2, s = WORKING →
{if P1, s = WORKING then P1, s := FAILED;
if P2, s = WORKING then P2, s := FAILED;};
Assertion
if P1, s = WORKING and P1, upStream then P1, down-
Stream := TRUE;
if P2, s = WORKING and P2, upStream then P2, down-
Stream := TRUE;
if not V1, isClosed then V1, leftStream := V1, right-
Stream;
if not V1, isClosed then V1, rightStream := V1, left-
Stream;
if not V2, isClosed then V2, leftStream := V2, right-
Stream;
if not V2, isClosed then V2, rightStream := V2, left-
Stream;
if not V3, isClosed then V3, leftStream := V3, right-
Stream;
if not V3, isClosed then V3, rightStream := V3, left-
Stream;
V1, rightStream := V2, leftStream;
V2, leftStream := V1, rightStream;
V2, rightStream := V3, rightStream;
V3, rightStream := V2, rightStream;
V3, leftStream := V1, leftStream;
V1, leftStream := V3, leftStream;
V3, rightStream := P2, downStream;

```

```

P2, downStream := V3, rightStream;
V3, leftStream := P1, downStream;
P1, downStream := V3, leftStream;
F1, inStream := P1, downStream;
F2, inStream := P2, downStream;
F3, inStream := V1, rightStream;
end

```

5.2 实验结果和分析

从实际得到的结果来看,第一步得到填充好的 GTS 实例对象,共有变量 21 个,转换事件 12 个,断言 21 个。

得到实例化对象后再对其进行划分,得到 4 个独立 GTS 与一个独立断言。然后利用 4 个独立 GTS 求出可达图,并编译为布尔公式,结果如图 9 所示。

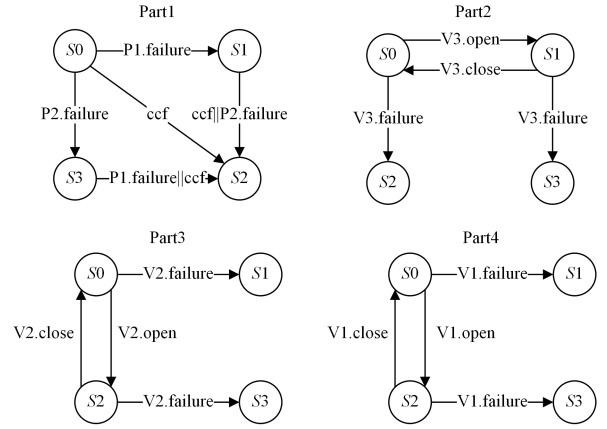


图 9 独立 GTS 的可达性图

Fig. 9 Reachability graph of independent GTS

Part1 中的布尔公式如下:

```

ØS0 = [null]
ØS1 = [[P1, failure]]
ØS2 = [[ccf], [P1, failure & (ccf || P2, failure)], [P2,
failure & (P1, failure || ccf)]]
ØS3 = [[P2, failure]]

```

Part2 中的布尔公式如下:

```

ØS0 = [null]
ØS1 = [[V3, open]]
ØS2 = [[V3, failure]]
ØS3 = [[V3, open & V3, failure]]

```

Part3 中的布尔公式如下:

```

ØS0 = [null]
ØS1 = [[V2, failure]]
ØS2 = [[V2, open]]
ØS3 = [[V2, open & V2, failure]]

```

Part4 中的布尔公式如下:

```

ØS0 = [null]
ØS1 = [[V1, failure]]
ØS2 = [[V1, open]]
ØS3 = [[V1, open & V1, failure]]

```

由于本系统较为复杂,包含的变量、转换、断言较多,以致编译独立断言之后的结果较多,这里只截取部分结果。

例如,在以组件 F3 无法正常工作为顶事件的故障树中,其中可能的直接原因是断路器 V1 和 V2 无输出,而 V1 和 V2 无输出可能源于其无输入,当电泵 P1 和 P2 处于关闭状态, V3 启动失败时造成 F3 无法正常获取电力。故障树如图 10 所示。

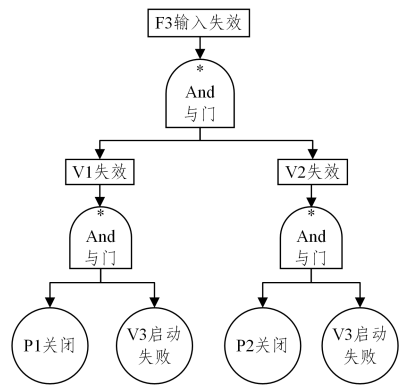


图 10 F3 失效的一个故障树
Fig. 10 Fault tree of F3 failure

所有结果均符合预期,故该算法的正确性得到了验证。整个算法运算耗时 1 s372 ms,经过划分优化得到了极好的效能。

考虑到故障树是一种多叉树,这里最终生成的 FTA 模型文件以 XML 格式为标准。利用 XML 文件保存故障树信息,无需考虑不同软件中故障树的存储结构之间的差异,其次 XML 也具有很好的可读性。故障树内部包含顶事件结点、中间事件结点和基本事件结点。其主要信息包括:1)故障树的树结构,其中结点元素是嵌套的;2)故障树中所有基本事件的故障概率等属性值,且数据在程序中的存储被设计成以事件 ID 为键值的字典存储数据。

6 相关工作与总结

目前工业界常用的故障树建模与分析工具,如 ISO-Graph^[21],Relax^[22]等,在进行系统安全性设计时,给用户提供了良好的故障树建模与分析等功能,但是此类工具仍然需要经验丰富的安全性分析人员在综合考虑系统设计方案和安全因素之后,手动建立复杂系统的故障树模型,目前并不具备从系统设计模型自动生成故障树模型的能力。

此外,近年来有一类工作是基于模型检测技术的系统故障自动分析的方法,例如,文献[23]所设计的工具 COMPASS 是基于 MBSA 的 SLIM 建模方法,内核中采用了分析引擎 xSAP,将系统的正常模型(Nominal Model)结合故障库进行自动化扩展,然后用模型检验工具 NuSMV 进行故障树相关性质的直接分析,如最小割集生成、顶事件概率计算、FMEA 的生成等。但比较而言,这种方法没有生成显式的故障树模型,缺少直观的工程判断和理解。

近年来关于自动生成故障树的研究越来越受大家的重视,但是基于航空航天领域经常使用的 AltaRica 语言建模自动生成故障树算法的研究还没有较大进展,本文便以 AltaRica 模型为基础,根据其平展化结果 GTS 研究如何自动生成故障树。与本文工作直接相关的是文献[24-26],它们的 Open-

AltaRica 开源平台中具备从 AltaRica3.0 模型编译生成故障树模型的功能。文献[27]也提及了相关的生成方法,同样包含划分、生成可达性图、生成布尔公式、编译独立断言,但该方法非常简略,缺少具体的设计实现细节。例如前文提到的编译独立断言到布尔公式的步骤,文献[27]中的伪代码并不能给出很多关键性细节,本文均以自主掌握的方法进行了创新,包括传播断言部分和求取系统状态全集方面。目前 OpenAltaRica 对较大规模的系统模型故障树的自动生成支持效果并不理想,从平展化 GTS 开始便经常出现各种意外的 Error,而本文以我们之前所做的平展化 GTS 作为故障树生成的来源,最终生成的故障树内容可读性更强,系统分析更加高效。

目前国内对 AltaRica 的研究工作处于初步阶段,本文工作是对 GTS 模型到故障树模型自动生成方法的详细分析、设计与实现,实现了自主掌握的核心方法。相比较而言,本文算法是全面性的设计与实现,首先划分算法是整个算法最能提升模型生成故障树效率的部分,并且在之后获取可达图的模块中还做到了以边结点集合、变量状态结点集合的两种方式存储,然后采用邻接矩阵表示。这种方法带来的好处就是具有容错性,利用不同的存储方式来存储图中信息,以便在之后面对大型系统模型时可以多方面验证。在编译独立断言到布尔公式的算法中,将之前的分治系统聚合,统一为一个系统,并在已经得到多种状态结果的情况下进行断言传播,其相比直接对一个完整的 GTS 进行分析生成故障树更为高效。

在接下来的工作中,我们将进一步把前期所做的工作和现在的工作结合起来,构建一个完整的面向 AltaRica3.0 的系统安全性建模与分析工具,并在典型的航空领域中进行较大规模的应用实例研究。

参 考 文 献

[1] WANG P, WU K, YAN F, et al. Security verification method of safety critical software based on system theoretic process analysis[J]. Journal of Computer Applications, 2019, 39(11): 3298-3303.

[2] CHE C, LIU Y F. Research on Model Based Safety Analysis [J]. Advances in Aeronautical Science and Engineering, 2016, 7(3): 369-373.

[3] HU J, CHEN S, WANG M M. A transformation method for AltaRica3.0 to Promela and its verification[J]. Computer Engineering and Science, 2017(4): 708-716.

[4] PROSVIRNOVA T. AltaRica 3.0: a model-based approach for safety analyses[EB/OL]. http://www.researchgate.net/publication/278827421_AltaRica_30_a_Model-Based_approach_for_Safety_Analyses.

[5] PROSVIRNOVA T, BATTEUX M, BRAMERET P A, et al. The AltaRica 3.0 project for model-based safety assessment[J]. IFAC Proceedings, 2013, 46(22): 127-132.

[6] BATTEUX M, PROSVIRNOVA T, RAUZY A. Advances in the simplification of Fault Trees automatically generated from AltaRica 3.0 models[M]// Safety and Reliability-Safe Societies in a Changing World. 2018: 907-914.

[7] RAUZY A B. Guarded transition systems: a new states/events

- formalism for reliability studies[J]. *Journal of Risk and Reliability*, 2008, 222(4): 495-505.
- [8] OLIVA J J R, LLANES J S, OJEDA M P, et al. Advanced combinatorial method for solving complex fault trees[J]. *Annals of Nuclear Energy*, 2018, 120: 666-681.
- [9] LI Z, ZHANG Y. Software Security Test Case Generation Based on Minimum Cut Sets [J]. *Computer and Digital Engineering*, 2019, 47(7): 1772-1775.
- [10] HAN S H. A top-down iteration algorithm for Monte Carlo method for probability estimation of a fault tree with circular logic[J]. *Nuclear Engineering and Technology*, 2018, 50(6): 854-859.
- [11] BATTEUX M, PROSVIRNOVA T, RAUZY A. Safety modeling and assessment with AltaRica 3.0[J/OL]. <http://www.altarica-association.org/ressources/Tutorial-IMBSA2019.pdf>
- [12] SONG J, CHEN B, LI X, et al. The software fault prediction model based on the AltaRica language[C]// 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). IEEE, 2019: 2549-2552.
- [13] RIVIECCIO U, JUNG A, JANSANA R. Four-valued modal logic: Kripke semantics and duality[J]. *Journal of Logic and Computation*, 2017, 27(1): 155-199.
- [14] CHEN R L. Fault tree analysis and calculation method [J]. *Technology Innovation and Application*, 2018 (244): 24.
- [15] KABIR S. An overview of fault tree analysis and its application in model based dependability analysis[J]. *Expert Systems with Applications*, 2017, 77: 114-135.
- [16] KATOEN J P, STOELINGA M. Boosting fault tree analysis by formal methods [M] // *ModelEd, TestEd, TrustEd*. Springer, Lecture Notes in Computer Science. Springer, Cham, 2017: 368-389.
- [17] ZHU D Q. Research on Fault Diagnosis Technology of avionics equipment[D]. Nanjing: Nanjing University of Aeronautics and Astronautics, 2002.
- [18] ESTEFAN J A. Survey of model-based systems engineering (MBSE) methodologies[J]. *Incose MBSE Focus Group*, 2007, 25(8): 1-12.
- [19] CHEN S, HU J, WANG L S. Design and Implementation of Flattening Algorithm for AltaRica 3.0 Model Based on ANTLR [J]. *Journal of Chinese Computer Systems*, 2020, 41(7): 1476-1487.
- [20] PROSVIRNOVA T, RAUZY A. AltaRica 3.0 project: compile Guarded Transition Systems into Fault Trees[C]// *European Safety and Reliability Conference, ESREL*, 2013.
- [21] DENG B, DAI B Q, WANG T. Dynamic fault tree analysis of SUBWAY vehicle-mounted ATP system based on Isograph [J]. *Journal of Safety Science and Technology*, 2016, 12(5): 80-85.
- [22] LEE D W, KIM I S, NA J W. A Case Study on Safety Analysis Procedure of Aircraft System using the Relex[J]. *The Journal of Advanced Navigation Technology*, 2018, 22(3): 179-188.
- [23] BOZZANO M, CIMATTI A, KATOEN J, et al. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems[C]// *International Conference on Computer Safety Reliability and Security*, 2009.
- [24] PROSVIRNOVA T, BATTEUX M, BRAMERET P A, et al. The altarica 3.0 project for model-based safety assessment[J]. *IFAC Proceedings Volumes*, 2013, 46(22): 127-132.
- [25] MORTADA H, PROSVIRNOVA T, RAUZY A. Safety assessment of an electrical system with AltaRica 3.0[C]// *International Symposium on Model-Based Safety and Assessment*. Springer, Cham, 2014: 181-194.
- [26] CHEN S. Research on System Safety Design Verification Methodology Based on AltaRica Model[D]. Nanjing: Nanjing University of Aeronautics and Astronautics, 2017.
- [27] PROSVIRNOVA T. AltaRica 3.0: a model-based approach for safety analyses[EB/OL]. http://www.researchgate.net/publication/278827421_AltaRica_30_a_Model-Based_approach_for_Safety_Analyses.



ZHAN Wan-li, born in 1997, postgraduate. His main research interests include software safety analysis and formal methods, etc.



HU Jun, born in 1973, Ph.D, associate professor, is a member of China Computer Federation. His main research interests include model-based safety analysis, software verification and embedded system design, etc.