Automatic Fault Tree Derivation from Little-JIL Process Definitions

Bin Chen, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil

Department of Computer Science, University of Massachusetts,
Amherst, MA 01003, USA
{chenbin, avrunin, clarke, ljo}@cs.umass.edu

Abstract. Defects in safety critical processes can lead to accidents that result in harm to people or damage to property. Therefore, it is important to find ways to detect and remove defects from such processes. Earlier work has shown that Fault Tree Analysis (FTA) [3] can be effective in detecting safety critical process defects. Unfortunately, it is difficult to build a comprehensive set of Fault Trees for a complex process, especially if this process is not completely well-defined. The Little-JIL process definition language has been shown to be effective for defining complex processes clearly and precisely at whatever level of granularity is desired [1]. In this work, we present an algorithm for generating Fault Trees from Little-JIL process definitions. We demonstrate the value of this work by showing how FTA can identify safety defects in the process from which the Fault Trees were automatically derived.

1 Introduction

A hazard in a safety critical system is “a state or set of conditions of the system that, together with certain other conditions in the environment, will lead inevitably to an accident” [2]. One fundamental requirement of developing a safety critical system, therefore, is to prevent or control the potential hazards. This requires an understanding of what hazards could occur in the system and how they could happen. A variety of hazard analysis techniques have been developed to identify potential hazards, assess their effect, and identify and evaluate the causal factors related to the hazards [2].

Fault Tree Analysis (FTA) [3] is a hazard analysis technique used to systematically identify and evaluate all possible causes of a given hazard. It has been well accepted and applied in many industries such as the nuclear industry [3] and the aerospace industry [4] etc. Given a potential hazard in a system, FTA deductively identifies events (component failures, human errors, etc.) in the system that could lead to the hazard and produces a fault tree, which provides a graphical depiction of all possible parallel and sequential combinations of those events. Once a fault tree has been derived, qualitative and quantitative analysis can be applied to provide information, such as specific sequences and sets of events that are sufficient to cause a hazard and overall system vulnerability to a hazardous outcome resulting from the occurrence of a particular event. This information can then be used as guidance for improvement of the design or implementation of the system.

Many processes such as medical processes are also safety critical. In this paper, we discuss how FTA can help to identify the weaknesses in processes and provide guidance on how to improve processes to reduce their vulnerability to hazards. Since manual fault tree derivation is time-consuming and error-prone, we propose an algorithm that automatically derives fault trees from processes specified using the Little-JIL process definition language [5].

The rest of this paper is organized as follows. Section 2 provides background on the Little-JIL process definition language. Section 3 gives a brief description of FTA and uses a simple process to demonstrate how FTA can facilitate process improvement. Section 4 presents our automatic fault tree derivation algorithm. The final section presents conclusions and suggests future work.

2 Little-JIL Process Definition Language

Little-JIL is a visual language for coordinating tasks that are to be executed by either computation or human agents. A process is defined in Little-JIL using hierarchically decomposed steps, where a step represents some specified task to be done by the assigned agent. We first give a brief overview of the semantics and notation of Little-JIL. For a detailed description of Little-JIL, see the Little-JIL Language Report [5].

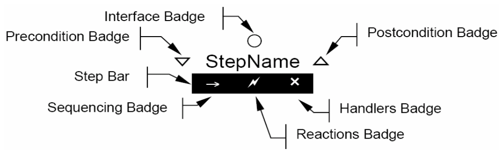

Fig. 1. Little-JIL step icon

Steps: Steps are the basic elements of Little-JIL processes. As shown in Fig. 1, each step has a name and a set of badges to represent the control flow, the interface, exceptions handled, etc. A step having no substeps is called a leaf step, and represents an activity that is to be performed by an agent, without any guidance or control from the process itself.

Step Sequencing: Every non-leaf step has a sequencing badge, which defines the order in which its substeps execute. For example, a sequential step indicates that its substeps are to be executed sequentially from left to right and is only completed after all of its substeps have completed. A parallel step indicates that its substeps can be executed in any (possibly arbitrarily interleaved) order. It, too, is only completed after all of its substeps have completed. A try step also indicates that its substeps are to be executed from left to right and it is completed as soon as one of its substeps is completed. A choice step indicates that any one of its substeps can be selected in order to complete the step.

Artifacts and Artifact Flows: Artifacts are entities that are used or produced by processes. Parameter declarations in the interface to a step specify artifacts read by the step as IN parameters and artifacts produced by the step as OUT parameters. Resources are special kinds of artifacts for which there is contention for access. They are managed by an external resource manager and their acquisitions need to be explicitly specified in step interfaces. After being acquired, resources can be passed as parameters like the other artifacts.

Exception Handling: A step in Little-JIL can throw exceptions when there are aspects of the step's execution that fail. A thrown exception is handled by a matching exception handler associated with the parent step of the step that throws the exception. An exception handler has an associated control-flow badge that indicates how the step catching the exception executes after the handler finishes. For example, the continue badge indicates that the step catching the exception should continue as if the substep that throws the exception completed successfully.

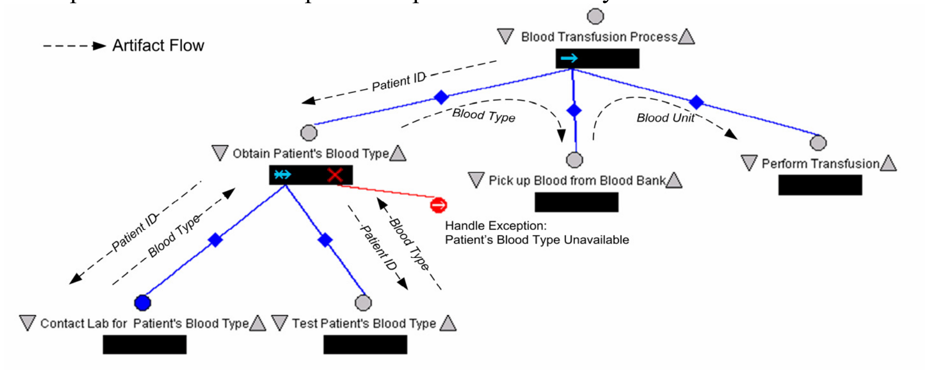
Artifact Flow
Blood Transfusion Process
Patient ID
Blood Type
Blood Unit
Obtain Patient's Blood Type
Pick up Blood from Blood Bank
Perform Transfusion
Handle Exception:
Patient's Blood Type Unavailable
Patient ID
Blood Type
Blood Type
Patient ID
Contact Lab for Patient's Blood Type
Test Patient's Blood Type

Fig. 2. Simple Blood Transfusion Process

Fig. 2 shows a simple Blood Transfusion Process. The root step "Blood Transfusion Process" is a sequential step, which means that its substeps, "Obtain Patient's Blood Type", "Pick up Blood from Blood Bank", and "Perform Transfusion", should be executed one by one, from left to right. Since "Obtain Patient's Blood Type" is a try step, it tries to execute step "Contact Lab for Patient's Blood Type" first. With the given patient ID passed as an argument, "Contact Lab for Patient's Blood Type" attempts to retrieve the patient's blood type from the lab. If the patient's blood type is available, it is returned as an argument to, and completes, step "Obtain Patient's Blood Type". Otherwise, an exception "Patient's Blood Type Unavailable" is thrown. This exception will be handled by an exception handler at "Obtain Patient's Blood Type". Since this handler is a continue exception handler as indicated by the right arrow, the process continues to execute "Test Patient's Blood Type" to get the patient's blood type. Once "Obtain Patient's Blood Type" is completed, the patient's blood type is passed to "Pick up Blood from Blood Bank", which acquires blood from the blood bank. Finally, blood is transfused at "Perform Transfusion".

3 Fault Tree Analysis for Processes

Event and Gates: The basic elements of a fault tree are events and gates. Events are used to represent faults, such as component failures, human errors, or other pertinent conditions in the system or environment. Fig. 3 shows symbols of several commonly used events and gates. Details about the others events and gates can be found in [3].

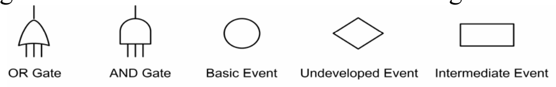

Fig. 3. Symbols of commonly used gates and events

Basic events are basic initiating faults or conditions. Undeveloped events are events that are not developed any further, either because information to derive the fault tree leading to these events is unavailable or because the probabilities of these events are considered to be insignificant. Basic events and undeveloped events are also called primary events because they require no further development. As opposed to primary events, intermediate events are events that need to be developed.

Each gate connects one or more input events to a single output event. The output event of an AND gate occurs if all of the input events occur. While the output event of an OR gate occurs if any of the input events occurs.

Fig. 4 shows a fault tree that represents combinations of faults in the simple Blood Transfusion Process that could lead to the hazard "The blood unit to be transfused is wrong".

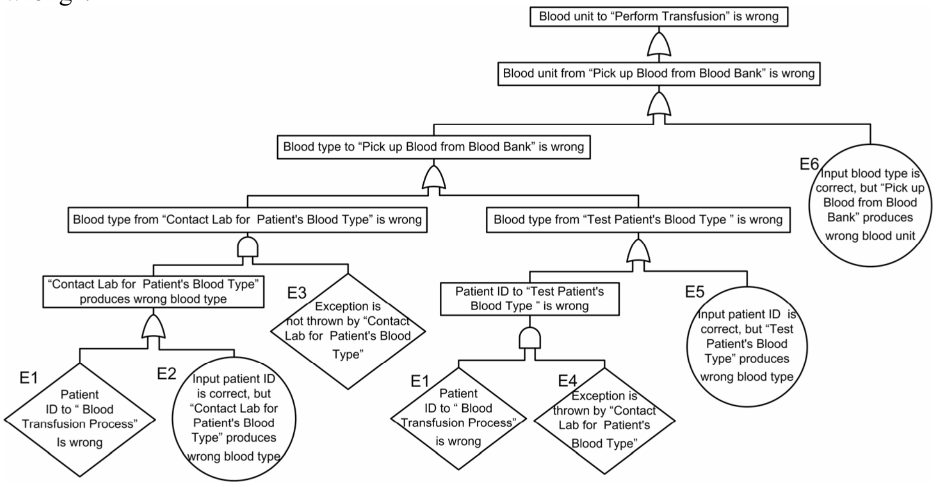
Blood unit to "Perform Transfusion" is wrong
Blood unit from "Pick up Blood from Blood Bank" is wrong
Blood type to "Pick up Blood from Blood Bank" is wrong
E6
Input blood type is correct, but "Pick up Blood from Blood Bank" produces wrong blood unit
Blood type from "Contact Lab for Patient's Blood Type" is wrong
Blood type from "Test Patient's Blood Type" is wrong
"Contact Lab for Patient's Blood Type" produces wrong blood type
E3
Exception is not thrown by "Contact Lab for Patient's Blood Type"
Patient ID to "Test Patient's Blood Type" is wrong
E5
Input patient ID is correct, but "Test Patient's Blood Type" produces wrong blood type
E1
Patient ID to " Blood Transfusion Process" Is wrong
E2
Input patient ID is correct, but "Contact Lab for Patient's Blood Type" produces wrong blood type
E1
Patient ID to " Blood Transfusion Process" is wrong
E4
Exception is thrown by "Contact Lab for Patient's Blood Type"

Fig. 4. Fault tree for the simple Blood Transfusion Process

Deriving Fault Trees: To derive a fault tree, the given hazard is represented as an intermediate event called the TOP event. Starting with this event, the fault tree derivation procedure proceeds to develop intermediate events until all leaf nodes in the fault tree are primary events. An intermediate event is developed by investigating the process, identifying the immediate, necessary, and sufficient events that cause this event, and connecting those events to it via a proper gate.

Analyzing Fault Trees: Once a fault tree has been derived, minimal cut sets (MCSs) for this fault tree can be computed automatically using Boolean algebra. A cut set is a set of primary events whose occurrence ensures that the TOP event occurs. An MCS is a cut set that cannot be further reduced. For example, MCSs of the fault tree in Fig. 4 are: {E1, E3} {E2, E3} {E1, E4} {E5} {E6}. These MCSs indicate that the process is exposed to the single point of failure - the hazard will definitely occur if either E5 or E6 occurs. Therefore subsequent changes need to be made to the process to remove these weaknesses.

There are usually several options that could be applied to control or eliminate a hazard in a process, For instance, a failure-resistant agent could be assigned to some steps where major faults could occur. Additionally, consistency check steps could be added to well-chosen places in the process to stop the propagation of faults. Usually only a few of the most effective options can be applied because of resource limitations or other constraints. The effectiveness of an option can be decided by the reduction in the probability of the hazard, if the probabilities of primary events are available. More details about analyzing fault trees can be found in [3].

3 Automatic Fault Tree Derivation

Fault trees are usually derived manually based on a deep understanding of the process. Due to complicated interleavings of events and inter-process communication, manual fault tree derivation can be time-consuming and error-prone. Analysts might fail to identify some events or include events that could not lead to the given event. These errors directly affect the analysis results that decide the validity of decisions made to improve the system.

Two main difficulties in manual fault tree derivation are: 1) how to be sure that one has found all possible events that could occur in the various steps of the process and 2) how to be sure that one has accurately and completely identified all cause-consequence relationships among events. In Little-JIL process definitions, steps have simple uniform interfaces. Therefore we only need to consider a few kinds of events that could possibly occur in these steps. Moreover, cause-consequence relationships among Little-JIL steps follow several patterns, which can be captured using templates. With these events and templates, a simple algorithm can be applied to automatically derive fault trees from Little-JIL process definitions.

Events: Several kinds of events can be defined based on Little-JIL step interfaces. Four of them represent faults that might occur at that particular step. [1]

– *Resource r acquired at step S is wrong*. When a step is started, resources needed by that step are acquired from an external resource manager. Resources acquired might be wrong because of errors in the resource manager, which is not captured in the process. Therefore these kinds of events are defined as undeveloped events.
– *Artifact o to step S is wrong.* These kinds of events can be either undeveloped events or intermediate events. They are intermediate events if the wrong artifacts are passed from some step in the process. If wrong artifacts are passed directly from the environment, they are defined as undeveloped events,
– *Artifact o from step S is wrong.* Since these kinds of events are always directly caused by other events that occur in the process, they are defined as intermediate events that need to be developed further.
– *All inputs and resources are correct, but step S produces wrong output o*. These kinds of events can only occur at leaf steps and represent the possibility that designated agents fail to execute those steps as required. They are defined as basic events.

[1] Without losing generality, we assume that no faults could occur during artifact passing. Unreliable artifact passing can be explicitly modeled using additional steps.

Two additional kinds of events are used to indicate conditions that decide where faults of a step could be propagated to. They are defined as undeveloped events.

- *No exceptions are thrown by S*. Faults of a step could be propagated to its immediate successors only if no exceptions are thrown by this step.
- *Exception e is thrown by S.* If a step throws an exception, its faults can only be propagated to the corresponding exception handling step.

According to [3], direct connections between gates should be avoided. Therefore temporary events are introduced to connect gates if necessary. They are intermediate events and do not change the semantics of fault trees. In the rest of this paper, temporary events are shown as rectangles drawn with dashed lines.

Templates: As noted above, *Artifact o to step S is wrong* and *Artifact o from step S is wrong* could be intermediate events that need to be further developed. To identify immediate events that could cause these events, several templates are defined based on Little-JIL semantics.

- Templates for *Artifact o from S is wrong*

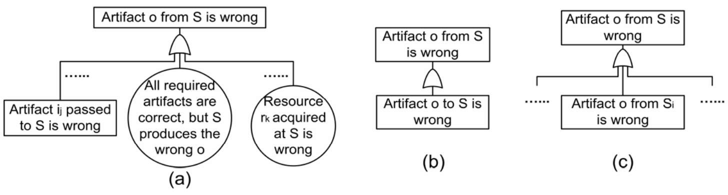
Artifact o from S is wrong
Artifact ij passed to S is wrong
All required artifacts are correct, but S produces the wrong o
Resource rk acquired at S is wrong
(a)
Artifact o from S is wrong
Artifact o to S is wrong
(b)
Artifact o from S is wrong
Artifact o from Si is wrong
(c)

Fig. 5. Templates for *Artifact o from S is wrong*

If S is a leaf step, its OUT parameters are produced by S from IN parameters and resources. Therefore if o is an OUT parameter of S, it might be wrong if any input to S is wrong, any resource acquired at S is wrong, or S produces the wrong output although all required artifacts are correct, as shown in Fig. 5 (a). On the other hand, if o is not an OUT parameter of S, it cannot be changed by S. In this case, o from S is wrong only if the same wrong o is passed to S, as shown in Fig. 5 (b).

If S is a non-leaf step, S itself does not change artifacts that are passed through it. Any artifact that comes out of S is passed from its substeps. Therefore an artifact o from S is wrong only if o coming from one or more of the substeps of S is wrong, as shown in Fig. 5 (c). Since the template is defined to capture the immediate causes, S_i in the figure should be a substep that could be the last substep of S to be executed. Such substeps can be decided according to the control badge of S.

- Templates for *Artifact o to S is wrong*

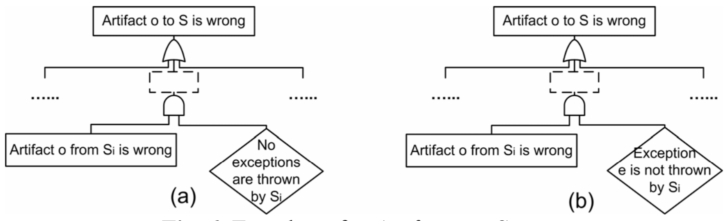
Artifact o to S is wrong
Artifact o from Si is wrong
No exceptions are thrown by Si
(a)
Artifact o to S is wrong
Artifact o from Si is wrong
Exception e is not thrown by Si
(b)

Fig. 6. Templates for *Artifact o to S is wrong*

As shown in Fig. 6 (a), if S is not an exception handling step, wrong artifacts to S might be propagated from a step S_i that might immediately precede S. Moreover, if S_i could throw exceptions, wrong artifacts can only be propagated to S if S_i does not throw exceptions. Steps that might immediately precede S can easily be calculated from the Little-JIL process definition.

For an exception handing step, it is executed only if the corresponding exception is thrown by some steps. Therefore, one step could propagate wrong artifacts to the exception handling step only if it throws the exception handled by the handler step, as shown in Fig. 6 (b).

Algorithm: With a given TOP event, the automated fault tree derivation algorithm keeps expanding the fault tree by applying proper templates to intermediate events that are leaf nodes until all leaf nodes are primary events. Applying this algorithm to the simple Blood Transfusion Process, we can get a fault tree semantically equivalent to the one shown in Fig. 4.

Limitations: The completeness a fault tree derived from a Little-JIL process by the algorithm depends on the completeness of the process. Thus, in cases where the Little-JIL process definition fails to completely represent steps in a real-world process that have an effect upon critical artifact flows, our algorithm will, accordingly, produce an incomplete fault tree.

Moreover, since Little-JIL processes do not specify how a leaf step produces its OUT parameters from its IN parameters and resources, our algorithm has to assume that any OUT parameter of a leaf step depends on all its IN parameters and resources. Thus, leaf steps that do not satisfy this assumption may cause the derived fault tree to contain superfluous subtrees.

Related Works: There exist several approaches for automatic fault derivation. Leveson et al. proposed a partially automated technique that derives fault trees from Ada programs based on templates [6]. We prefer the advantages of a fully automated approach. Another approach by Leveson et al. is a fully automatic fault tree derivation, but from the Requirements State Machine Language (RSML) specifications [7]. The approach by Pai et al. automatically derives fault trees from UML models [8]. This approach requires the dependency relationships to be explicitly specified. McKelvin et al. designed an algorithm that derives fault trees from Fault Tolerant Data Flow (FTDF) models [9]. These other automated approaches seem to us to suffer from their dependence upon modeling formalisms that lack semantics that are sufficient to represent complex processes clearly, completely, and precisely. Different from these approaches, some approaches, such as [10] and [11], use model checking to generate fault trees. They require explicit state machine models to represent the faults that can occur within components.

4 Conclusion

Fault Tree Analysis is a hazard analysis technique that is well accepted and applied to complex systems in various industries. FTA can also help to improve processes. To improve the efficiency and accuracy of FTA, fault trees can be automatically derived if processes are specified by languages that have precise enough semantics. In this paper, we present an automated fault tree derivation algorithm based upon Little-JIL process definitions. The superior clarity and precision of Little-JIL should result in more complete and definitive fault trees which should then subsequently lead to fault-tree analysis that should help us improve the Little-JIL processes.

Acknowledgements

We would like to thank Zongfang Lin and Sandy Wise for their many helpful suggestions with this work. This research was supported by the National Science Foundation under Award Nos. CCR-0204321 and CCR-0205575. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of The National Science Foundation, or the U.S. Government.

References

1. Clarke, L.A., Chen, Y., Avrunin, G.S., Chen, B., Cobleigh R.L., Frederick K., Henneman, E.A., Osterweil, L.J.: Process Programming to Support Medical Safety: A Case Study on Blood Transfusion. Proceedings of the Software Process Workshop (SPW2005), Beijing, China. (2005)
2. Leveson N.G.: Safeware: System Safety and Computers. Addison-Wesley (1995)
3. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault-Tree Handbook, Reg. 0492. US Nuclear Regulatory Comm., Washington, D.C. (1981)
4. W.E. Vesely et al.: Fault Tree Handbook with Aerospace Applications. NASA (2002)
5. Wise, A.: Little-JIL 1.0 Language Report. Technical report (UM-CS-1998-024), Department of Computer Science, University of Massachusetts, Amherst, MA (1998)
6. Cha, S.S., Leveson, N.G., Shimeall, T.J.: Safety Verification in Murphy Using Fault Tree Analysis. ICSE '88: Proceedings of the 10th International Conference on Software Engineering, Singapore (1988) 377-386
7. Ratan, V., Partridge, K., Reese, J., Leveson N.G.: Safety Analysis Tools for Requirements Specifications. http://www.safeware-eng.com/index.php/publications/SafAnTooReq
8. Pai, G.J., Dugan, J.B.: Automatic Synthesis of Dynamic Fault Trees from UML System Models.13th International Symposium on Software Reliability Engineering (ISSRE'02) 243
9. McKelvin M.L.Jr., Eirea, G., Pinello, C., Kanajan, S., Sangiovanni-Vincentelli,A.: A Formal Approach to Fault Tree Synthesis for the Analysis of Distributed Fault Tolerant Systems. Procs. of the 5th ACM International Conference on Embedded Software (2005) 237-246
10. Liggesmeyer, P., Rothfelder, M.: Improving System Reliability with Automatic Fault Tree Generation. FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (1998) 90
11. Bozzano, M., Villafiorita, A.: Improving System Reliability via Model Checking: the FSAP / NuSMV-SA Safety Analysis Platform. In Proceedings of SAFECOMP 2003, LNCS 2788, Edimburgh, Scotland, United Kingdom (2003) 49-62