Process Programming to Support Medical Safety: A Case Study on Blood Transfusion

Lori A. Clarke[1], Yao Chen[1], George S. Avrunin[1], Bin Chen[1], Rachel Cobleigh[1], Elizabeth A. Henneman[2], Kim Frederick[1], and Leon J. Osterweil[1]

[1]Department of Computer Science
University of Massachusetts
Amherst, MA 01003 USA
{clarke, avrunin, chenbin, yaoc, rcobleig, kfrederi, ljo}@cs.umass.edu

[2]School of Nursing
University of Massachusetts
Amherst, MA 01003 USA
henneman@nursing.umass.edu

Abstract. Medical errors are now recognized as a major cause of untimely deaths or other adverse medical outcomes. To reduce the number of medical errors, the Medical Safety Project at the University of Massachusetts is exploring using a process programming language to define medical processes, a requirements elicitation framework for specifying important medical properties, and finite-state verification tools to evaluate whether the process definitions adhere to these properties. In this paper, we describe our experiences to date. Although our findings are preliminary, we have found that defining and evaluating processes helps to detect weaknesses in these processes and leads to improved medical processes definitions.

1 Introduction

It has been estimated that there are approximately 98,000 deaths per year in the United States resulting from medical errors [7]. The Institute of Medicine (IOM) reported that many medical errors are caused by faulty processes and conditions that lead people to make mistakes or fail to prevent them [6]. Although the IOM advocates using more information technology in order to help improve medical care, it does not indicate what kinds of technology should be employed.

In the University of Massachusetts Medical Safety Project, software engineering researchers from the Department of Computer Science have been working with researchers and medical practitioners from the University of Massachusetts School of Nursing and from Baystate Medical Center to evaluate how selected technologies might help reduce medical errors. Although it is not possible to totally eliminate mistakes, it is our hypothesis that medical processes can be defined in such a way that mistakes are less likely to occur.

Medical processes tend to be complex, concurrent, and exception-prone. They tend to involve multiple practitioners with very different perspectives about the on-going process. Thus, we are interested in a process language that can capture this complexity yet still be understandable to a (trained) medical professional. Moreover, the process language should be precise enough to support static analysis techniques and to eventually drive simulations and executions.

To date we have experimented with using the Little-JIL process programming language [11], the Propel property elucidation system [10], and several finite-state verification systems, specifically LTSA [1, 9], SPIN [5], and FLAVERS [4]. In this paper we report on our experiences using these technologies to define and evaluate a blood transfusion process. Blood transfusion plays a vital process in modern health systems. Although blood transfusion errors are rare, when they do occur, they can result in death and are among the most serious types of medical errors. Thus, we use blood transfusion as an example to demonstrate how our approach is effective at improving the safety of medical processes.

The rest of this paper is organized as follows. Section 2 presents a brief overview of the Little-JIL process programming language. Section 3 presents part of the blood transfusion process as specified using Little-JIL. Section 4 describes how properties are specified using Propel and the results of our analysis using finite state verification. The final section highlights our results and discusses future work.

2 Little-JIL Features

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. Steps may also indicate any prerequisites, postrequisites, and exception handling behavior that should be associated with the step. Non-leaf steps, in addition to the above, also indicate the order for processing all substeps. The language has precise enough semantics that Little-JIL programs can be executed or can serve as the subject of careful static analysis.

To help the reader understand the Blood Transfusion process example, we first give an overview of the semantics and notation of Little-JIL. For a detailed description of Little-JIL, see the Little-JIL Language Report [11].

Steps: Steps are the basic elements of Little-JIL programs. As shown in Figure 1, each step has a name and a set of badges to represent the control flow, exceptions handled, prerequisites, and postrequisites. Each step need only be defined once, but can be referenced many times. References are represented by a step with the name of the referenced step, but with no badges. Although not shown in our examples here, steps also can indicate the resources required, including the agent responsible for step execution.

Step Execution: At run-time, a step can be in one of five states: posted, started, completed, terminated and retracted as shown in Figure 2. When a step is eligible to be started, it is moved into the posted state. It is started when the agent assigned to the step obtains the resources that it requires and begins to do the work. If the step is finished successfully, it is moved into the completed state and resources are released. If the agent

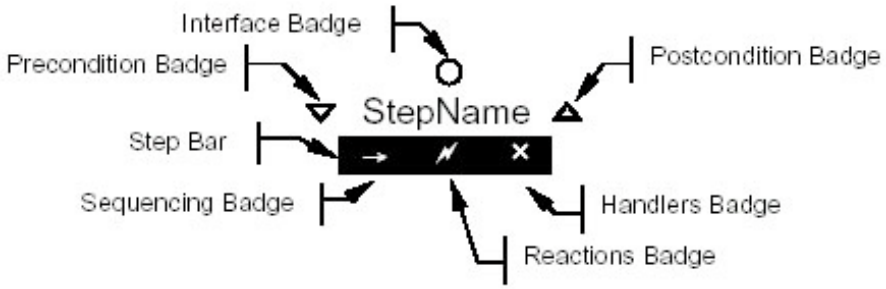

Fig. 1. Little-JIL step icon

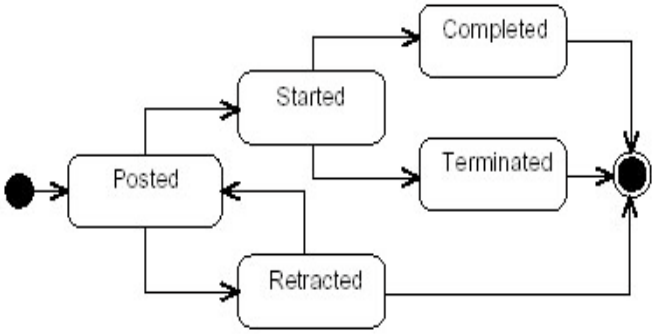

Fig. 2. States associated with Little-JIL step execution

fails to complete the work, the step is moved to the terminated state. A step is retracted if it is withdrawn from an agenda after having been posted but without being started by the agent. In the analysis phase, we often want to refer to a specific state of a step. To do this, we append the state name to the name of the step. Thus, "Transfuse_STARTED" refers to the step "Transfuse" when it is in the started state.

Step Sequencing: Every non-leaf step has a sequencing badge, which defines the order in which its substeps execute. A sequential step indicates that its substeps are to be executed from left to right and is only successfully completed after one of its substeps has successfully completed. A parallel step indicates that its substeps are to be executed asynchronously and that it cannot be successfully completed until all of its substeps successfully complete. A choice step allows the agent to dynamically select a substep to execute among its substeps. A choice step is considered completed only after one of its substeps have completed. A try step indicates that its substeps are executed from left to right until one of them has been completed. A try step is successfully completed only if one of its substeps successfully complete.

Exception Handling: A step in Little-JIL can throw exceptions when aspects of the step fail. For example, if a prerequisite is not satisfied, it may indicate that an exception is to be thrown. A thrown exception is handled by a matching exception handler associated with the parent step of the step that throws the exception or, if no such handler is found, the exception is rethrown by the parent step.

An exception handler has an associated control-flow badge that indicates how the step catching the exception executes after the handler finishes. There are four kinds of control badges:

- continue: the step catching the exception should continue as if the substep that throws the exception completed successfully;
- complete: the step catching the exception should be completed;
- rethrow: the step catching the exception should be terminated and the exception rethrown to the parent of this step;
- restart: the step with the exception handler should be restarted.

Requisites: Each step may have a prerequisite and a postrequisite. Requisites provide a way to check entry and exit conditions associated with a step. A prerequisite has to be completed before its associated step is initiated. A postrequisite has to be completed before its associated step is completed. When a requisite cannot successfully complete, the associated step is terminated and an exception is thrown.

Deadlines: Deadlines determine the time by which a step must be completed. Deadlines are used to define the maximum time allowed for a certain task. If a step continues to execute past its stated deadline, an exception is thrown.

Resources and Agents: The interface to a step specifies the resources used by the step, where agent is a special type of resource. For example, in a medical process, the agent might be a nurse, doctor, patient, or computer system. Each step must have an agent; if no agent is declared, the agent is inherited from the parent step.

Diagrams: To facilitating viewing, Little-JIL programs are decomposed into diagrams, where a diagram usually fits into a single window. Diagrams are usually used to decompose a Little-JIL program into conceptually meaningful subprocesses.

3 Blood Transfusion Example

We have used Little-JIL to model a real-world blood transfusion process. This process model consists of 20 Little-JIL diagrams, comprised of about 112 steps. In this section, we present a few of the Little-JIL blood transfusion diagrams to give the reader an indication of what the model looks like.

A patient blood transfusion process cannot start unless there is a blood transfusion order from a physician. One order may require that several units of blood be transfused to the patient. Once the required units have been transfused, the process completes. Figure 3 shows the top diagram of this process.

In the root step, *Patient Blood Transfusion Process* has a prerequisite step *Order Blood*. There is a cardinality "+" adjacent to the edge between the *Patient Blood Transfusion Process* step and *Perform Transfusion Order* step, which means that *Perform Transfusion Order* will be done at least once. Since *Patient Blood Transfusion Process* is a sequential step, instances of *Perform Transfusion Order* must be executed sequentially. Before *Perform Transfusion Order* starts, the agent (agent assignments are not shown) must check the form signed by the patient, indicating consent for the blood transfusion. If the consent form is not signed, a *NoPatientConsent* exception will be thrown and then handled by the *No Patient Consent* exception handler associated with

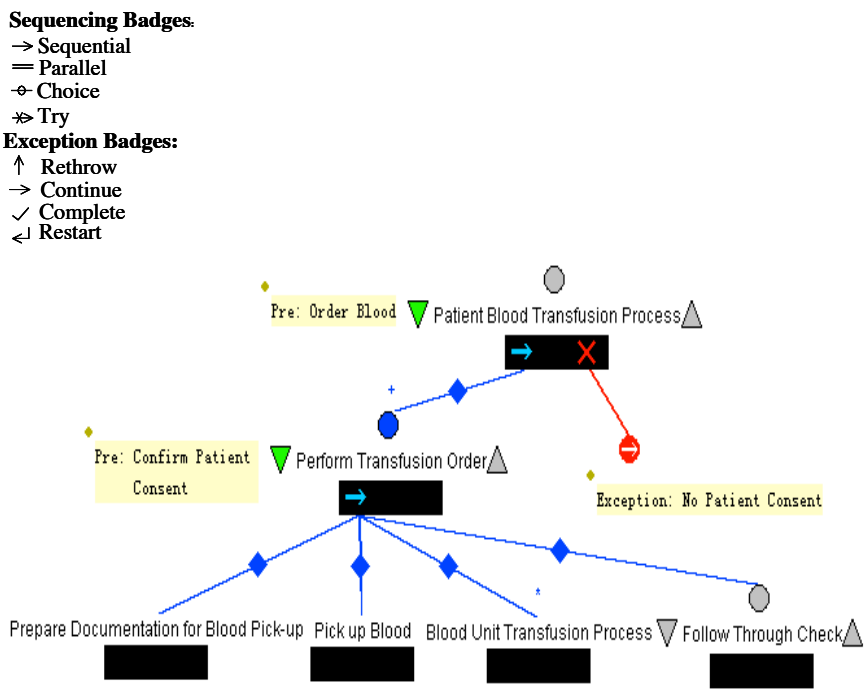

Fig. 3. Patient Blood Transfusion Process

the *Perform Transfusion Order* step. Since this handler is a continue exception handler, as indicated by the right arrow, after completion of the handler, the process continues the sequential execution of the *Patient Blood Transfusion Process* step, meaning that the "next" instance of the *Perform Transfusion Order step* may start. If the consent form is signed, the agent can start to execute *Perform Transfusion Order*. The *Perform Transfusion Order* step has four substeps: *Prepare Documentation for Blood Pick-up*, *Pick up Blood*, *Blood Unit Transfusion Process,* and *Follow Through Check*. The right arrow sequencing badge specifies that these substeps should be executed one by one, from left to right. *E*ach one of these substeps is a reference to a step defined in another diagram, so none of these steps are elaborated in this diagram. There is a cardinality "*" adjacent to the edge between the *Perform Transfusion Order* step and *Blood Unit Transfusion Process* step, which means that *Blood Unit Transfusion Process* will be done once per unit of blood.

Figure 4 shows the diagram that elaborates the *Blood Unit Transfusion Process* step. According to clinical research, the most common adverse outcomes during blood transfusions are caused by a failure to detect that an incorrect unit had been issued at the bedside [7]. To prevent such common errors, bedside checks are recommended. Thus, in our process definition, there are two bedside checks, *Identify Patient* and *Product Verification*. *Identify Patient* requires that the identity of the patient be established.

The *Product Verification* step definition, which is shown in Figure 5, requires a visual comparison of the information on the transfusion tag with the blood product bag.

All identifying information on the blood product, the transfusion tag, and the patient identification armband must be verified. Thus there are four substeps to be executed: *Verify Product Tag Matched to Product Label, Check Product Expiration Date*, *Verify Product Tag Matched to Patient Armband,* and *Verify Product Type Matched to Patient Record*. Since these verification steps are independent of each other, they can be done in any order, as indicated by the parallel sequencing badge. If any of these substeps finds a discrepancy, a *FailedProductVerification* exception is thrown. This exception is rethrown to the handler *Handle Failed Product Verification* associated with the parent of *Product Verification*, step *Bedside Checks*. This exception handler, although not shown here, would handle this discrepancy according to hospital policy.

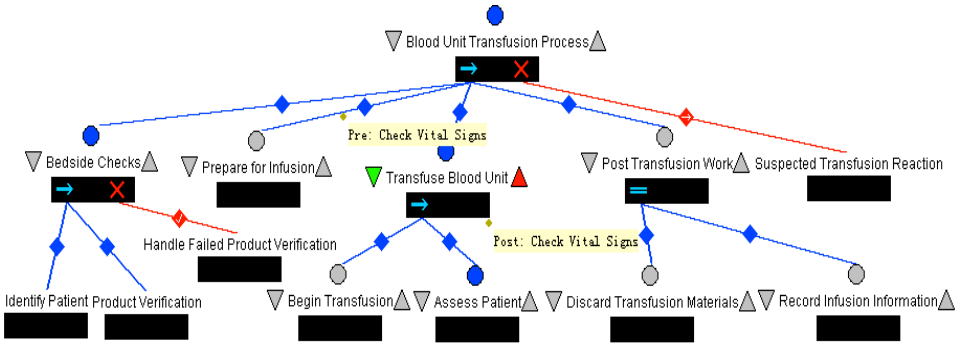

Fig. 4. Blood Unit Transfusion Process

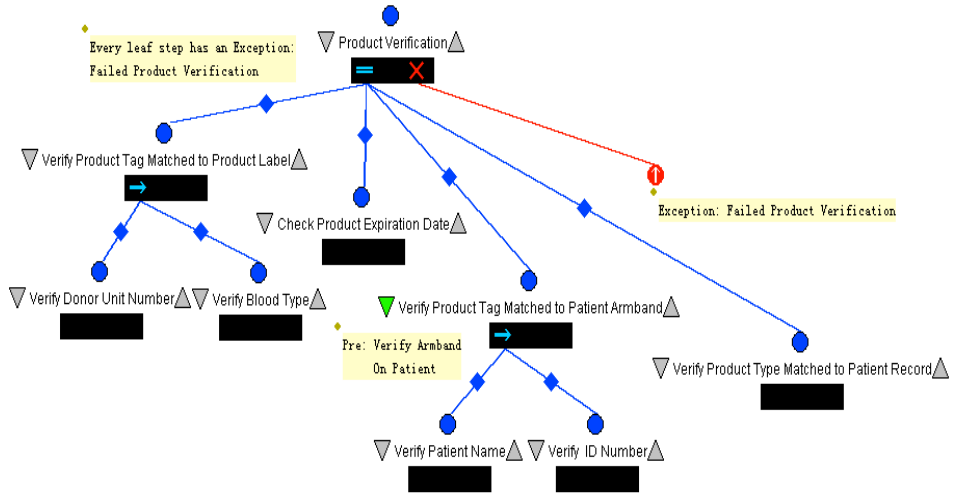

Fig. 5. Product Verification Process

4 Analyzing Processes

Although we have only shown a small part of the Blood Transfusion process, it is easy to see that it quickly becomes quite complex. The Little-JIL definition tersely describes complex control flow. This is both a strength and a weakness. It is a strength because medical professionals can understand the process definitions and help to describe them and develop improvements to them. Moreover, the process definition can easily be decomposed into subprocesses (e.g., diagrams) so that one's focus can be directed to relatively small, coherent aspects of the process. This terseness is a weakness, however, because it is easy for humans to overlook or misunderstand some of the complex flows through the system or among subprocesses. This is particularly true when exceptions or parallel execution can occur [2].

One way to help validate a process is to use analysis techniques to verify that important policies are not violated by the process definition. These policies can be represented as formal properties stated in terms of the states of the steps. We then apply finite-state verification techniques to determine if these properties will always hold on all possible traces through the process. For example, for the blood transfusion process, patient identification on the patient's armband must match the patient information on the tag affixed to the blood product before that unit of blood is transfused. If this property does not hold for the process definition, the finite-state verification tool will provide a counterexample trace through the system showing where at least one such violation occurs. We can use this trace to identify and correct the error in the process and then try to reverify the revised process definition.

In this section we first describe some of the properties that need to be verified for the Blood Transfusion process and how we represented those properties and then describe what techniques we used to verify these properties.

4.1 Representing Properties

It is a surprisingly difficult task to determine the properties that should be verified. In the medical field, policies often exist that are a starting point for these properties. Below are some example policies often associated with the blood transfusion process:

- Patient informed consent must be confirmed prior to each blood transfusion process being initiated.
- Patient identification must be verified prior to obtaining a blood specimen for a type and cross match.
- Patient identification (name/ID number) on the armband must match the patient identification information on the tag affixed to the blood product before preparing for infusion.

- All blood product infusions must be started within 30 minutes of the arrival of the blood units.
- Blood product must be checked at the patient's bedside before preparing for the transfusion.
- Donor unit number on blood tag must be matched to donor unit number on blood bag before preparing for blood transfusion.
- Check that the blood product has not expired must be performed before preparing for blood transfusion.
- Vital signs must be obtained and documented at the start of the transfusion.
- Vital signs must be obtained and documented 15 minutes into the transfusion.
- A physician's order must be confirmed in order for a blood transfusion to occur.
- Blood transfusion must be stopped immediately if a reaction is suspected.
- Doctor and Transfusion Services must be notified immediately if a transfusion is stopped because of a suspected reaction.

Such policies are often vague however and need to be translated into a precise instantiation based on the process that is actually being applied. For example, "confirm patient consent" must be represented in terms of the consent form that is actually used at the hospital where the process is being applied. Moreover, who is to do this confirmation and how is this confirmation documented?

Beyond that, finite-state verification requires a rigorous representation of each property. It is rare for English descriptions to describe accurately and unambiguously all the situations that need to be considered. The Propel system [10] is designed to help users consider all the situations associated with formulating a property. Propel provides a question tree that guides the user through the options that should be considered. Figure 6 shows an example of the question tree. After making some initial selections in this question tree, the user can continue to select options from the question tree or can choose instead to select options from a template of English phrases, called disciplined natural language (DNL), or from a finite-state automaton (FSA) template. Figure 7 shows the Propel GUI when formulating the DNL and FSA representation of the resulting property. (Because of space limitations, we have reduced the event names and English description in this example.)

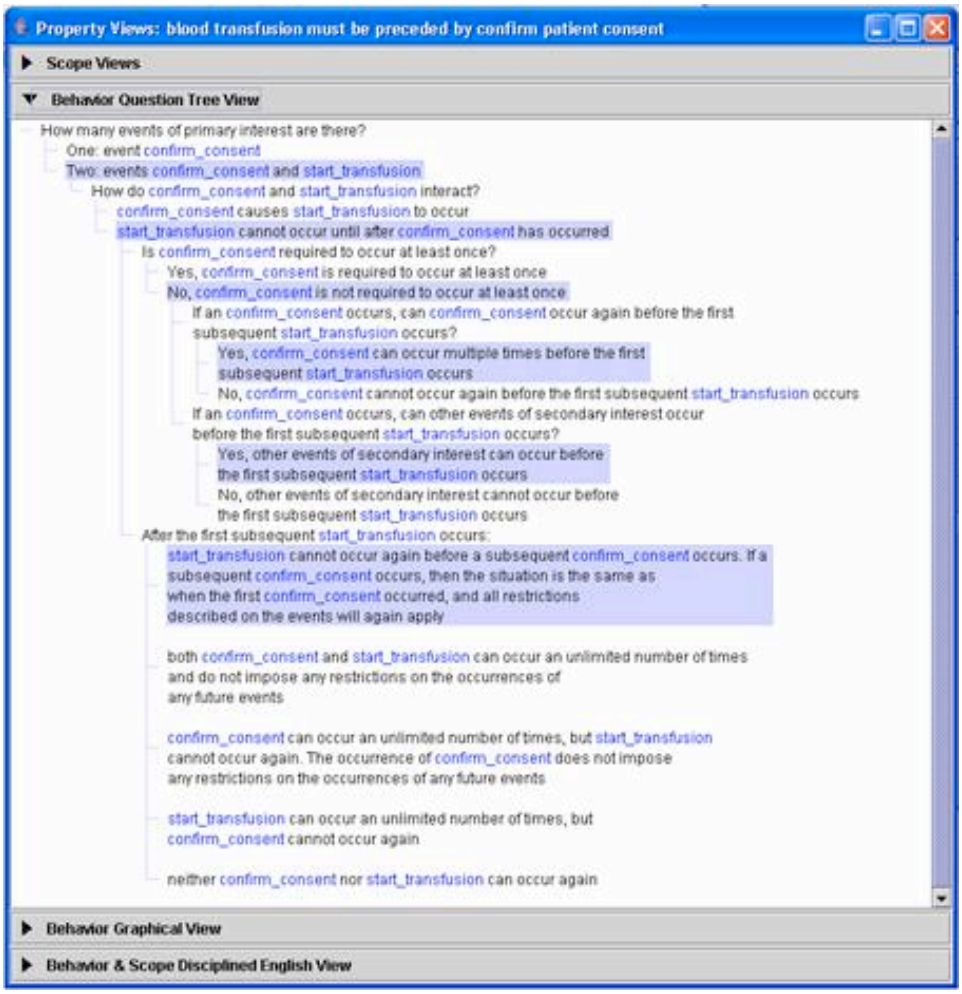

Fig. 6. Propel Question Tree

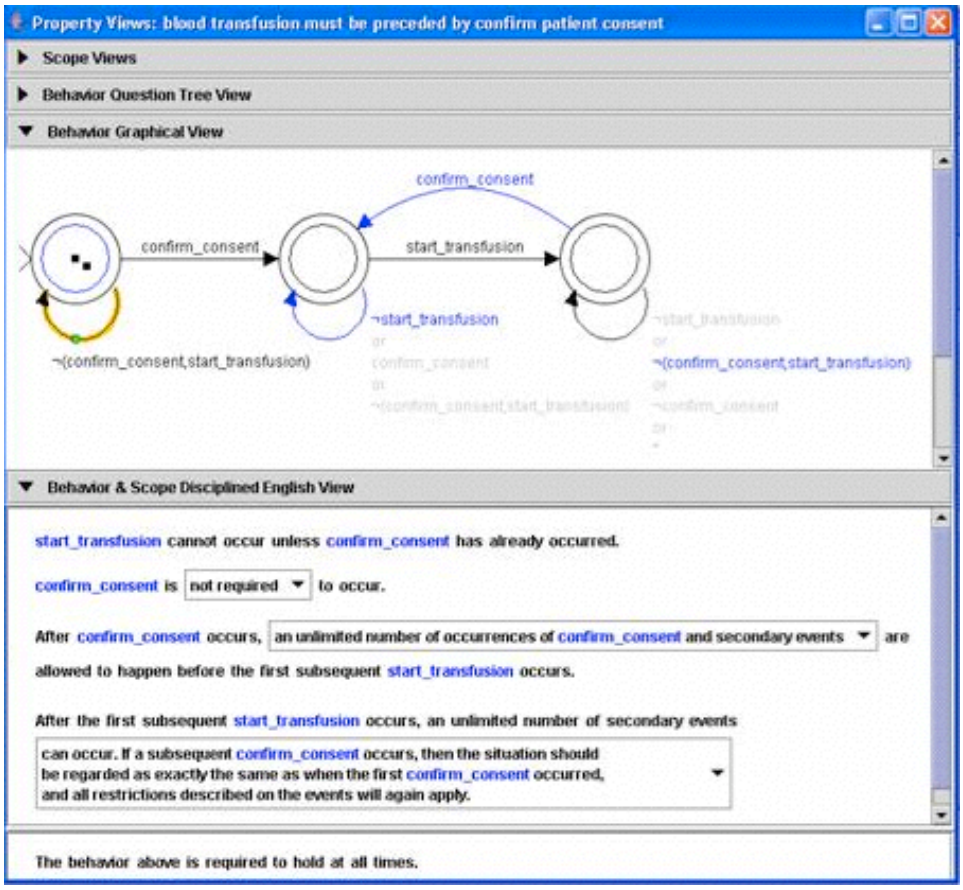

Fig. 7. Propel DNL and FSA representation

Thus, after using Propel, the first policy:

Patient informed consent must be confirmed prior to each blood transfusion process being initiated.

would be represented by the following DNL representation:

Blood_Transfusion_STARTED cannot occur unless Confirm_Patient_Consent_COMPLETED has already occurred.

Confirm_Patient_Consent_COMPLETED is not required to occur.

After Confirm_Patient_Consent_COMPLETED occurs, an unlimited number of occurrences of Confirm_Patient_Consent_COMPLETED and all other events in the alphabet of this property (except Blood_Transfusion_STARTED) are allowed to happen before the first subsequent Blood_Transfusion_STARTED occurs.

After the first subsequent Blood_Transfusion_STARTED occurs, an unlimited number of other events in the alphabet of this property can occur.

Blood_Transfusion_STARTED cannot occur again until after another Confirm_Patient_Consent_COMPLETED occurs. If another Confirm_Patient_Consent_COMPLETED occurs, then the restrictions described on the events following a Confirm_Patient_Consent_COMPLETED would again apply.

The reader might be surprised at how long and detailed the resulting disciplined natural language is for this one relatively simple property. A careful examination of Figures 6 and 7, however, shows the number of issues that must be addressed in precisely specifying such a property. The resulting FSA would be the basis for verifying the process definition. Some finite-state verification systems, such as FLAVERS, accept a property represented as a FSA. For others, the FSA would need to be translated into their property representation. For example, for SPIN, the FSA must first be translated into linear time temporal logic.

4.2 Process Verification

There are several finite-state verification tools that could be used to determine if the process definition is consistent with a property. To date, we have investigated using three such tools, SPIN, FLAVERS, LTSA. To facilitate using different tools, we first translate the Little-JIL process into an intermediate representation, called the Bandera Intermediate Representation (BIR). BIR was specifically designed to support finite-state verification and thus was a natural choice [3]. Once we have the BIR representation, we translate BIR to the internal form required for the particular verifier. Figure 8 depicts this two-state translation process.

A common problem with finite-state verification is that the size of the state space that must be explored grows too large. Direct translation of a process usually results in a model that is too large to be verified. Therefore, we use several optimizations and abstractions to reduce the size of the model generated. Some of these transformations have been previously reported [2, 8] and some are currently being investigated. All the transformations that are used must be shown to be conservative for the property and process definition. This means that a process will not be reported to be consistent with a property unless that is indeed the case for the unoptimized version as well. False positives, violations that do not correspond to any real trace through the system, can be a problem but are less likely to occur for process descriptions than for detailed designs or source code.

All the verifiers that we have used have been able to find (the same) errors in the process and to prove interesting properties about the blood transfusion process. All of them have some limitations and their translation and optimization process is being improved to address these concerns. FLAVERS is currently best able to handle the larger problems, but requires more insight about the constraints that must be introduced to eliminate false positives.

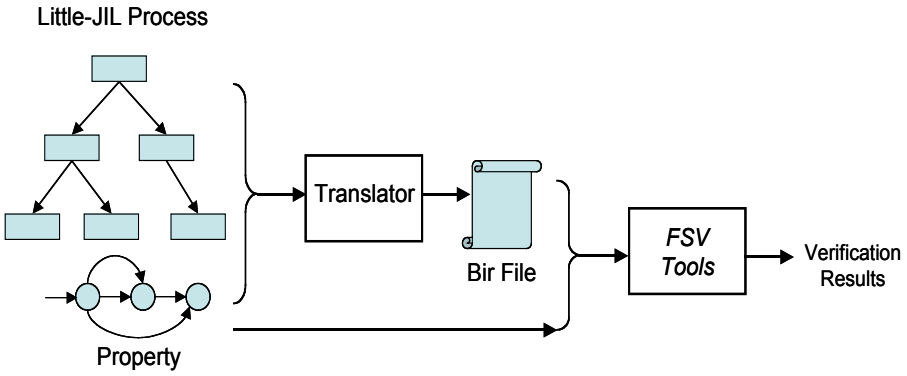

Fig. 8. The Little-JIL translation to BIR and then the BIR translation to the expected input for the selected Finite-State Verification (FSV) tool

5. Conclusions and Future Work

We have successfully used Little-JIL to specify a real-world, non-trivial blood transfusion process and verified that the process satisfies some important safety properties. We have also learned a considerable amount about the strengths and weaknesses in technology that we are using.

The Little-JIL process language has been extremely useful in representing the blood transfusion process. Surprisingly, the medical professionals have become very adept at understanding the Little-JIL processes. It has turned out to be an excellent medium for describing the blood transfusion process and discussing alternative processes. The medical professionals have shied away from actually creating the process definitions. Instead they rely on the computer scientists to create the process definitions, although they are quick to point out problems or suggest improvement. As noted, there is also a tension between the expressiveness of the process language and the analyzability of the resulting processes. Humans like flexible processes, but such processes are much more difficult to analyze since they result in more choices and thus more cases to consider.

As might be expected, simply rigorously defining a process uncovers problems with that process. Often there were disagreements among the medical professionals about the process definitions. Sometimes this could be attributed to the different roles that medical professionals have (e.g., the nurse's view versus the doctor's view), but sometimes these disagreements revealed a real problem in the underlying process and an opportunity for a medical error to occur. In the future we are interested in exploring how best to decompose (and then compose) the process definitions according to the different roles.

Property specification also helped improve the process definitions. In considering a property, it often became clear that the process definition omitted important details. The medical policies that we had available before trying to define the process were useful, but the extra detailed required to formulate a property resulted in deeper under-

standing of the problem that eventually was reflected in the process definition. For example, considering the details about patient consent for a blood transfusion revealed that we needed to consider how long a delay could exist between initial consent and the transfusion, how many transfusions could occur with one consent, and what happens if the patient rescinds consent.

The verification of the process definition did indeed reveal errors in the process. Some were problems that appeared obvious once they were revealed. The more interesting errors involved exceptions and concurrent behavior that lead to unexpected event orderings. We found the verification useful in helping us debug the process definitions (and the translators). The medical process definitions are ripe for detecting event-ordering problems. Medical professionals are often involved in multiple parallel activities and dealing with exceptional conditions upon exceptional conditions. It is a problem domain that appears well matched with the technology we are applying.

There are many areas of future investigation. This case study has revealed limitations in the process language, the property specification approach, and the verification tools. For example, all three technologies need to be extended to have better support for timing constraints. The process language needs better support for visualizing the process. The property specification framework is still awkward to use, and the verification tools need much improved, process-specific optimization techniques. The Little-JIL to BIR translator currently does not support recursion. To handle recursion, we simply unroll the recursive step up to a given bound, but this might make the verification unsound.

The medical professionals are very interested in evaluating different kinds of medical processes, not just blood transfusion processes. In addition to improving safety, they are interested in improving efficiency with respect to turn around and through put. They would like to see how efficiency is affected by different symptom mixes (e.g. ankle sprains versus cardiac pain), different resources, different resource allocation strategies, and different processes. Such evaluations will depend on doing extensive simulations using real event histories. Finally, in the long term it would be desirable to actually execute carefully evaluated processes in the clinical setting. These processes could help medical professionals track and prioritize their numerous tasks.

Acknowledgments

We would like to thank Stephen Siegel, Jamieson Cobleigh, Sandy Wise, Ethan Katz-Bassett, and Barbara Staudt Lerner for their many helpful suggestions with this work.

This material is based upon work supported by the National Science Foundation under Award No. CCF-0427071, the U. S. Army Research Office under Award No. DAAD19-01-1-0564, and the U. S. Department of Defense/Army Research Office under Award No. DAAD19-03-1-0133.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the U. S. Army Research Office, or the U. S. Department of Defense/Army Research Office.

References

1. Cheung, S.C., D. Giannakopoulou, and J. Kramer. Verification of Liveness Properties Using Compositional Reachability Analysis, in Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering. 1997. p. 227-243.

2. Cobleigh, J.M., L.A. Clarke, and L.J. Osterweil. Verifying Properties of Process Definitions, in ACM SIGSOFT International Symposium on Software Testing and Analysis. 2000. Portland, OR: ACM Press. p. 96-101.

3. Corbett, J.C., M.B. Dwyer, J. Hatcliff, and Robby. Bandera: A Source-level Interface for Model Checking Java Programs, in 22nd International Conference on Software Engineering. 2000. Limerick, Ireland. p. 762-765.

4. Dwyer, M.B., L.A. Clarke, J.M. Cobleigh, and G. Naumovich, Flow Analysis for Verifying Properties of Concurrent Software Systems. ACM Transactions on Software Engineering and Methodology (to appear), 2004.

5. Holzmann, G.J., The Model Checker SPIN. IEEE Transactions on Software Engineering, 1997. 23(5): p. 279-294.

6. Institute of Medicine, A New Health System for the 21st Century, in Crossing the Quality Chasm. 2001, National Academy Press: Washington, DC. p. 23-38.

7. Kohn, L.T., J.M. Corrigan, and M.S. Donaldson, eds. To Err is Human: Building a Safer Health System. 1999, National Academy Press: Washington DC.

8. Lerner, B.S. Verifying Process Models Built Using Parameterized State Machines. in ACM SIGSOFT International Symposium on Software Testing and Analysis. 2004. Boston, Ma. p. 274-284.

9. Magee, J. and J. Kramer, Concurrency: State Models and Java Programs. 1999: John Wiley & Sons.

10. Smith, R.L., G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. PROPEL: An Approach Supporting Property Elucidation, in 24th International Conference on Software Engineering. 2002. Orlando, FL. p. 11-21.

11. Wise, A., Little-JIL 1.0 Language Report. 1998, Department of Computer Science, University of Massachusetts: Amherst, MA.