Engineering Medical Processes to Improve Their Safety

An Experience Report

Leon J. Osterweil[1], George S. Avrunin[1], Bin Chen[1], Lori A. Clarke[1], Rachel Cobleigh[1], Elizabeth A. Henneman[2] and Philip L. Henneman[3]
1 Laboratory for Advanced Software Engineering Research (LASER)
University of Massachusetts at Amherst, Amherst, MA 01003
{ljo, avrunin, chenbin, clarke, rcobleig} @ cs.umass.edu
2 School of Nursing, University of Massachusetts at Amherst, Amherst, MA 01003, henneman@nursing.umass.edu
3 Baystate Medical Center, Springfield, MA and Tufts University School of Medicine, Boston, MA, philip.henneman@bhs.org

Abstract. This paper describes experiences in using precise definitions of medical processes as the basis for analyses aimed at finding and correcting defects leading to improvements in patient safety. The work entails the use of the Little-JIL process definition language for creating the precise definitions, the Propel system for creating precise specifications of process requirements, and the FLAVERS systems for analyzing process definitions. The paper describes the details of using these technologies, employing a blood transfusion process as an example. Although this work is still ongoing, early experiences suggest that our approach is viable and promising. The work has also helped us to learn about the desiderata for process definition and analysis technologies that are intended to be used to engineer methods.

1 Introduction: The Problem and Our Proposed Approach

Medical errors cause approximately 98,000 patients to die each year [1] in the United States. US Institute of Medicine (IOM) reports have suggested that the delivery of healthcare must fundamentally change to address medical error (eg. see [1, 2]). In particular, these studies suggest that many serious medical errors result from *system* rather than individual failures, leading the IOM to advocate the development of healthcare systems that directly address patient safety. In particular, the IOM report states, "what is most disturbing is the absence of real progress... in information technology to improve clinical *processes* [italics ours]" ([1 pg. 3]).

Please use the following format when citing this chapter:

Osterweil, L. J., Avrunin, G. S., Chen, B., Clarke, L. A., Cobleigh, R., Henneman, E. A., Henneman, P. L., 2007, in IFIP International Federation for Information Processing, Volume 244, Situational Method Engineering: Fundamentals and Experiences, eds. Ralyté, J., Brinkkemper, S., Henderson-Sellers B., (Boston Springer), pp. 267-282.

Encouraged by these findings, the authors of this paper began a project to investigate how software engineering research in process definition and analysis might be applied and extended to help reduce errors and improve safety in medical processes.

Our preliminary research (eg. [3]) showed that in many cases current medical processes are often described only at a high-level of generality and are usually not defined completely and precisely. These processes typically describe standard practices, but usually do not address how healthcare providers should react when unusual, yet expectable, situations arise. Because of this, healthcare providers can often find themselves in situations that are not directly addressed by the processes they learned, and thus are often unsure of whether or not their actions conform to recommended care guidelines. In addition, aspects of current care process descriptions are frequently vague, ambiguous, or inconsistent, allowing different providers to arrive at different understandings about their specifics. Such descriptions may lead workers to believe they are following recommended care guidelines when, in fact, their care has deviated, increasing the possibility of error.

In the work we describe here, software engineering researchers and medical experts developed precise, rigorous definitions of medical processes that capture not only the standard cases, but also the exceptional situations that can arise. The process definitions also captured the inherent concurrency and multi-tasking frequently undertaken by busy healthcare providers, as well as details of the complex use of resources in performing medical processes. The processes defined covered different aspects of medical care, such as blood transfusion, chemotherapy, and emergency department patient flow. In all of these domains, the literature indicates that errors can be frequent and can result in serious negative consequences [1, 4, 5].

This preliminary investigation indicated somewhat different goals for the engineering of methods in these different areas of medical practice, and thus suggested somewhat different approaches. The Emergency Department (ED) sought to reduce patient waiting time, as delay is a safety hazard (and a source of pain and inconvenience). Moreover, the highly concurrent nature of Emergency Department activities is believed to increase the chance of incorrect process execution, which also leads to safety hazards. Other concerns included identifying bottlenecks and improving resource utilization. This suggested the desirability of analyzing precise, rigorous process definitions to study their concurrency and resource utilization.

In blood transfusion and chemotherapy there was concern for the identification and removal of process defects that create hazards to patient health and safety. These concerns suggested the value of at least two complementary engineering approaches, namely fault tree analysis and finite-state verification, each applied to a precise definition of safety-critical processes. Analysis of fault trees promises to indicate possible effects of incorrect performance of process steps [6, 13], while finite state verification (eg., [8, 9]) promises to identify sequences of tasks that, even if performed perfectly, could still lead to safety hazards [1610].

Our project aims to evaluate the effectiveness of defining medical processes using a rigorously defined language, carrying out rigorous analysis of the processes to detect defects, and then improving the processes by defect removal. Here we address in detail only one research activity, namely our work in improving processes related to blood transfusion in a clinical setting and only touch briefly upon some of our other activities. In the next section we present the Little-JIL process definition

language and provide some examples of how it was used to define a blood transfusion process. Section 3 describes and evaluates our experiences, and Section 4 summarizes some related work. Section 5 summarizes some of our other work on medical processes, and suggests future directions for this research.

2 An Example: A Clinical Blood Transfusion Process

The administration of blood and blood products is a common, high-risk, resource-intensive medical intervention. Despite strict regulation by the US Food and Drug Administration as well as healthcare accreditation agencies, the error rate in transfusion medicine is significant and believed to be underreported [11]. To investigate whether the precise definition and analysis of this process could help identify defects that lead to such errors, we used the Little-JIL process definition language [12, 13] to define a transfusion process in detail. We then used the Propel property definition system [14] to specify desired properties, and then used the FLAVERS finite-state verification system [9] to determine whether the properties could ever be violated by any path through the defined process.

2.1 Principal Features of Little-JIL

Little-JIL [12, 13] is a language originally developed for defining the processes by which software is developed and maintained. Wise [13] provides full technical details of the language. Here we outline its salient features. A Little-JIL process is defined by means of specification of three components, an artifact collection, a resource repository, and a coordination specification. Each addresses a different area of concern. The artifact collection contains the various items, initial, intermediate, and final, that are the focus of the activities carried out by the process. The resource repository specifies the agents and other capabilities available to support performing the activities. The coordination specification ties these together by specifying which agents, aided by which supplementary capabilities, will perform which activities upon which artifacts at which time(s). Because of its central role in specifying this, the coordination diagram is generally the central focus of a Little-JIL process definition.

A Little-JIL coordination specification has a visual representation, but is, nevertheless, precisely defined using finite-state automata. This renders processes defined in Little-JIL amenable to definitive analyses that are analogous to those used to evaluate application software. Among the key features of Little-JIL that distinguish it from most process languages are its 1) use of abstraction to support scalability and clarity, 2) use of scoping to make the use of step parameterization clear, 3) facilities for specifying parallel processing, 4) extensive capabilities for defining how to handle exceptional conditions, and 5) clarity and precision in specifying iteration.

A Little-JIL coordination specification is defined using hierarchically decomposed steps (Figure 1), where a step represents a task to be done by an assigned agent. Each step has a name and a set of badges to represent control flow

among its sub-steps, its interface (a specification of its input/output artifacts and the resources it requires), the exceptions it handles, etc. A step with no sub-steps is called a leaf step and represents an activity to be performed by an agent, without any guidance from the process.

Resources and Agents—Each Little-JIL step contains as part of its interface a specification of the types of resources that are required in order to support the execution of the step. Some examples of resources are physicians, blood units, beds, and accesses to medical records of various sorts. The assignment of an actual resource instance is carried out by a separate Resource Manager, which maintains a repository of available resources and their capabilities, and identifies a specific resource instance to be assigned in response to the step's request. Each step always requires one specially designated resource instance, called its agent, which is the resource that is assigned responsibility for the performance of the step. Little-JIL agents may be either humans or automated devices. In some cases either might be appropriate, and the choice is then made by the Resource Manager, rather than being dictated by the process definition.

Substep Decomposition—Little-JIL steps may be decomposed into substeps of two different kinds, ordinary substeps and exception handlers. The ordinary substeps define the details of how the step is to be executed. The substeps are connected to their parent by edges, which may be annotated by specifications of the artifacts that flow between parent and substep and also by cardinality specifications. Cardinality specifications define the number of times the substep is to be instantiated and may be a fixed number, a Kleene *, a Kleene +, or a Boolean expression (indicating whether the substep is to be instantiated). Exception handlers define how exceptions thrown by the step's descendants are handled. The edge from exception handler to parent is annotated with the type of the exception being handled, parameters being passed, and an indication of how execution continues after the exception has been handled.

Step sequencing –A non-leaf step has a sequencing badge (an icon embedded on the left of the step bar; e.g., the right arrow in Figure 1), which defines the order of substep execution. For example, a sequential step (right arrow) indicates that its substeps execute from left to right. A parallel step (equal sign) indicates that its substeps execute in any (possibly interleaved) order. A choice step (circle slashed with a horizontal line) indicates that step execution is by choosing any of the alternative substeps. A try step (right arrow with an X on its tail) mandates a sequence in which substeps are to be tried as alternatives.

Artifacts and artifact flows – An artifact is an entity (e.g., a physical entity or data item) that is used or produced by a step. Parameter declarations are specified in the interface to a step (circle atop the step bar) as lists of the artifacts used by the step (IN parameters) and the artifacts produced by the step (OUT parameters). Artifact flow through steps can be defined to take place in one of two different ways, 1) hierarchically, as the flow of artifacts between parent and child steps, and 2) by means of data channels. The flow of artifacts along a parent-child edge is indicated by attaching to the edge identification of the artifacts and their direction of flow.

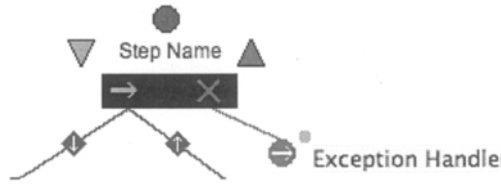

Figure 1 – A Little-JIL step icon.

Data Channels—Data Channels are named entities that directly connect specifically identified source step(s) with specifically identified destination step(s). A data channel acts much like a buffer, with some steps using the data channel as an output and others using it as an input. This construct helps define how streaming data, for example, is handled by a process. It can also be used to synchronize concurrently executing steps, since steps may choose to block when sending or receiving.

Requisites – A Little-JIL step optionally can be preceded or succeeded by a step that is executed before or after execution of the main body of the step. A prerequisite is represented by a down arrowhead to the left of the step bar, and a post-requisite is represented by an up arrowhead to the right of the step bar. Requisites facilitate checking for a condition either before executing a step or to assure that execution has been acceptable. The failure of a requisite triggers the occurrence of an exception.

Exception Handling – A step in Little-JIL can signal the occurrence of exceptional conditions when some aspect of the step's execution fails (e.g., the violation of one of the step's requisites). This triggers the execution of a matching exception handler associated with an ancestor step that throws the exception (and represented as a step attached by an edge to an X on the right of the step bar in Figure 1). Little-JIL also incorporates a facility for specifying in which, of a variety of ways, execution should proceed after completion of the exception handler. This is an important feature that is difficult to represent in many other languages.

Scoping – The parent step and its descendants represent a scope in Little-JIL, enabling specification that certain entities and datasets can be considered local to that scope. Little-JIL also supports recursive specifications of steps within its own scope, which clarifies the iterative application of a process step to its defined arguments.

2.2 An Example Using Little-JIL to Define a Blood Transfusion Process

Figure 2 is a blood transfusion process coordination diagram. The actual process has 112 Little-JIL steps, and is too large to present here. Thus, we present a version that fits the needs of terse exposition, but is still representative of the actual process.

Figure 2 shows that the full transfusion process, *Single-Unit Transfusion Process*, consists of four substeps to be executed in sequence (note the right arrow in the step bar), namely *Bedside Checks, Prepare for Infusion, Transfuse Blood*, and *Post Transfusion Work*. The first three are all decomposed into subprocesses that are

defined by separate diagrams. In this paper we show only the decomposition of the first step, *Bedside Checks* (in Figure 3). The last of the four substeps, *Post Transfusion Work* is further decomposed in Figure 2 into two substeps that can be executed in parallel (note the equal sign in its step bar), namely *Discard Transfusion Materials* and *Record Infusion Information*. Here too, these substeps are further decomposed in separate coordination diagrams, each of which adds further details. Substeps are the primary method of supporting the incorporation of details into Little-JIL process definitions, since decomposition can proceed to any level of abstraction.

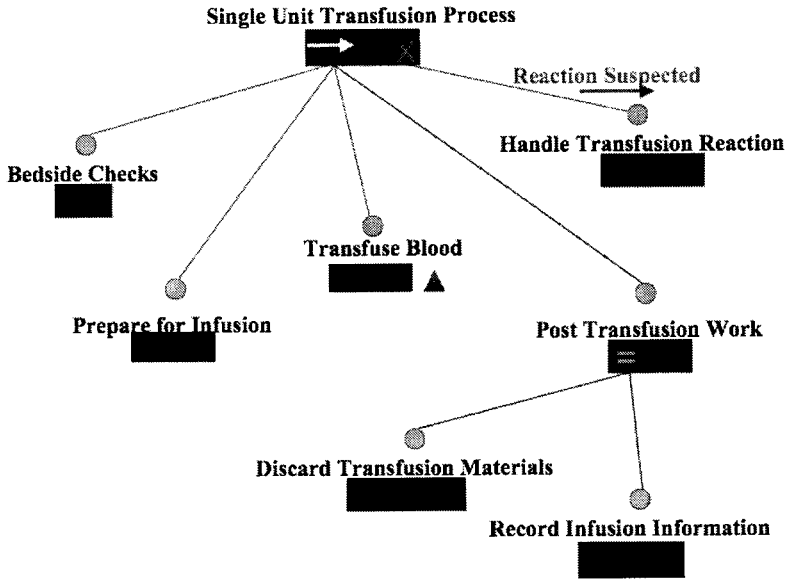

Figure 2: A coordination diagram of a Little-JIL blood transfusion process.

Note also that the *Transfuse Blood* step has a postrequisite (indicated by the fact that the arrowhead on the right of its step bar is colored in). We do not show the decomposition of this step, but the postrequisite defines the activities to be performed after this step's execution to determine whether there has been an adverse reaction to the blood transfusion. If so, this postrequisite will throw a *Reaction Suspected* exception, causing control to be transferred to the *Handle Transfusion Reaction* exception handler, which is another substep of the parent, *Single-Unit Transfusion Process*. *Handle Transfusion Reaction* is also elaborated by a structure of substeps, again not shown here for lack of space. But, as might be expected, this handler is of significant size and thus represented by a non-trivial structure. Note that the exception handler edge is annotated with the type of exception that is handled and with a right arrow icon indicating that execution continues as though the

step that threw the exception finished execution. Thus, the next step executed is *Post Transfusion Work.*

We note that the diagrams in Figures 2 and 3 do not contain all the information comprising a complete coordination specification. The Visual-JIL editor is used to create Little-JIL coordination diagrams, and it can elide much information in the interests of reducing visual clutter. In particular the step's agent and resource requirements are not shown in these diagrams, but are represented iconically by the circle above the step. Likewise, the artifacts that are arguments to the various steps must be specified on the edges of a Little-JIL diagram and as part of the information attached to the circle above each step. This information too is elided here for clarity.

While the process depicted in Figure 2 presents a straightforward top-level view of the transfusion process, this view is somewhat illusory. There is considerable additional complexity that must be defined in detail in order to capture salient issues in blood transfusion, thereby rendering them amenable to definitive analysis. To illustrate this, we decompose the first substep, *Bedside Checks*. This step, depicted in Figure 3, represents the checking that is to be done prior to a transfusion, and is thus of central importance in establishing a good basis for safety analysis.

Note that the hierarchical elaboration of *Bedside Checks* makes it clear that this step consists of two separate checks, one to assure the transfusion is being given to the right patient and one to assure the blood to be transfused is correct. The equal sign in the *Bedside Checks* step bar indicates that these two checks can be performed in any order, and indeed can be interleaved with each other. The details of the two checks are interesting and important, and also indicate the value of some of the semantic power of a language such as Little-JIL. Note, for example, that the first substep, *Check Patient ID*, consists of the execution of *Get Patient ID*, followed by the execution of *Check ID to Patient Match.* Each of these requires considerable further elaboration (not shown for lack of space), as they can be seriously complicated by various combinations of situations such as an unconscious patient, a patient who is bleeding profusely, and a patient who has no ID band. The full elaboration of these substeps deals with combinations of these situations, using language features such as exception handling. Of central concern to this step, however, is the possibility that the *Check ID to Patient Match* step might fail. This may happen for many different reasons, but here we indicate that it might happen as a consequence of the evaluation of this step's postrequisite, in which case this contingency is handled by throwing the *ID and Patient Don't Match* exception. The handling of this exception is done by recursively calling the *Check Patient ID* step. Here we note that, because Little-JIL steps are abstractions, and thus function very much like procedure calls, this recursive call of *Check Patient ID* occurs in the scope and context of the exception handler, thus making available to the step information that may be carried along as arguments to the recursive call. Thus, Little-JIL supports sending information about the reasons that the check has failed. This is a faithful representation of what would happen in the real-world situation, where this information would be used to guide the next execution of the *Get Patient ID* step (eg. gathering new information on the patient),and the next invocation of the *Check Patient ID* step. This shows the value of providing strong support for abstraction.

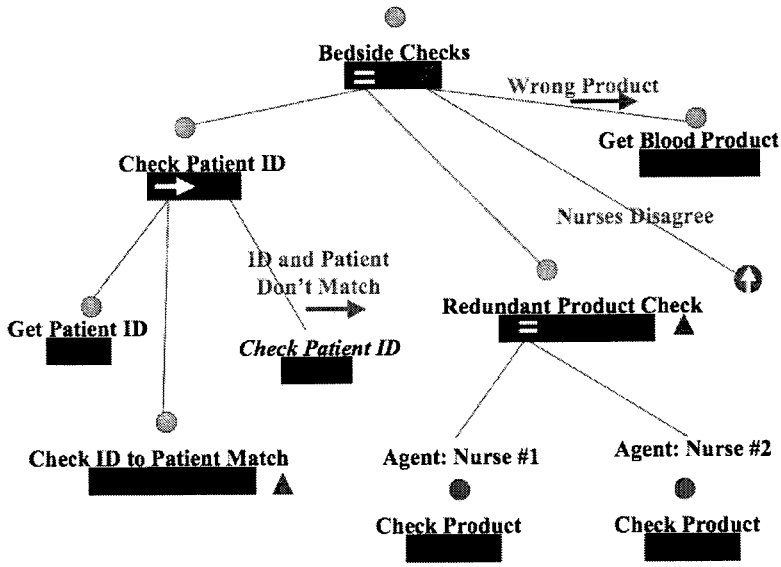

Figure 3: The hierarchical elaboration of the *Bedside Checks* step

The other checking step, *Redundant Product Check*, provides examples of the value of other Little-JIL language features. Here we note that this step consists of the parallel execution of two different instances of the *Check Product* step, not elaborated here for lack of space. But we have specified that the resources required as agents for the two steps are two different nurses (*Nurse #1* and *Nurse #2*),who are obliged to perform the identical check to be sure that the blood product is correct.

Redundant Product Check has a postrequisite, a comparison (not shown here) of the reports from the two nurses to make sure that both agree that the blood product is correct. We show two possible exceptions that can be thrown. If the two nurses disagree, a Nurses Disagree exception is thrown, and is handled by rethrowing (upward arrow) the exception to an ancestor step for resolution. If there is agreement that the blood product is incorrect, a *Wrong Product* exception is thrown, and is handled by the *Get Blood Product* step, which is a reinstantiation of the step defining how a blood product is requested from the blood bank. That step appeared previously in this process definition, but is not shown for lack of space. Again, note that the fact that *Get Blood Product* is called in the context of the handling of this exception means that the report from the nurses providing details about what was wrong with the blood product can now be transmitted to the blood bank.

2.3 Using Propel and FLAVERS Analysis to Look for Process Defects

We now provide a very brief and simplified example of how we applied finite-state verification to the blood transfusion process definition. Our approach to finite-

state verification is described in detail in [9]. In that paper we describe how FLAVERS performs exhaustive checks of all possible paths through a system in order to determine whether or not the execution of any path would cause a violation of a desired property. For our purposes, a property is a specification of the requirements for some aspect of the behavior of a system. As a requirement, the property is a specification against which a system is to be verified. For example, a property may specify that a certain event may not occur until another event has occurred. In our work we compare a process against such properties. In cases where the property is violated we modify the process (note, we ignore for the moment the possibility that the property may be incorrectly specified) and verify the modified process to the property again, continuing until the verification succeeds, thereby improving the process. For our analysis, properties are represented as finite-state automata and describe certain sequences of events that must (or must not) occur in every execution of the process. Figure 4 shows an example of one such property for our blood transfusion process. This automaton specifies that after executing the *Get Patient ID* step, executing the *Check ID to Patient Match* step moves the process into a state where *Transfuse Blood* is acceptable as the next step. The automaton also specifies, however, that *Transfuse Blood* is not acceptable if *Get Patient ID* or *Check ID to Patient Match* has not yet been executed. This would cause the automaton to be moved to the error state. Note also that if *Check ID to Patient Match* is followed by *Get Patient ID*, the automaton is moved back into the initial state, from which *Transfuse Blood* again causes a transition to the *ERROR State.* This event sequence occurs if *Check ID to Patient Match* is followed by the throwing of an exception, because the match has failed. The exception is handled by reinvoking the *Check Patient ID* step. Because its first substep is *Get Patient ID*, this repeated execution of *Get Patient ID* indicates that the *Check ID to Patient Match* step has failed and that *Transfuse Blood* is not acceptable now. Automata such as that indicated in Figure 4 were generated with the aid of our Propel system [14], which facilitates the generation of such automata by using a question tree to elicit specifics of the properties. Propel also features a natural language facility to describe the semantics of the automaton in natural English. Note, that for this example there are several other important properties that need to be verified, including one that states that *Check ID to Patient Match* must always be immediately preceded (e.g., no intermediate *Transfuse Blood* events) by *Get Patient ID.*

Once a process and an automaton are defined, using Little-JIL and Propel respectively, we use the FLAVERS finite-state verification system to determine whether any execution of the process could drive the automaton to the error state. While the verification may appear straightforward for this example, we note that even this small example poses serious challenges. The parallel step allows all possible interleavings of substep executions, and the recursive invocation adds further complexity. Finally, the sheer size of the final process (112 steps) makes the verification problem very large. A verifier such as FLAVERS, which employs a number of optimization techniques, is usually able to handle the verification of properties of modest-sized processes such as this one.

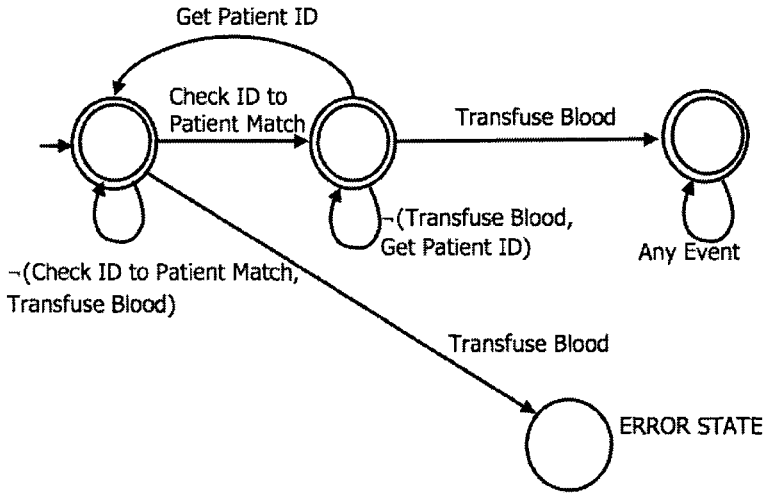

Figure 4: This finite-state automaton requires *Transfuse Blood* to happen only if *Check ID to Patient Match* has executed, but NOT been followed by *Get Patient ID.*

3 Experiences and Evaluation

Our experience in defining and analyzing the blood transfusion process suggests the value in this approach, as it has resulted in detection and correction of process defects. Some of our experiences were as expected, but many were unexpected.

3.1 Process Elicitation

Many process deficiencies were realized just in the interviewing that was necessary to elicit the complete, detailed process. We quickly found that the original process guidelines often did not use terms consistently. For example, we found that a word such as "check" sometimes was used in the same way as the word "verify", but sometimes it had different connotations. Careful elicitation of what was meant, by using Little-JIL to clarify the exact meanings, often led to the desired understandings. This led the medical professionals to examine their terms, to define them more carefully, and to use them more consistently. In doing so, the resulting process definitions left less room for confusion, misunderstanding, and ambiguity.

It was not uncommon for the process guidelines to leave responses to exceptions unspecified. For example, in some cases a process required a "check" for a condition, with the understanding that some alternative processing was necessary if the "check" fails. In many cases, however, the existing process description assumed

that check would always succeed and provided little or no guidance about what to do in case of a failure. Here again specifying details of the process quickly raised such issues and led the medical professionals to synthesize responses, thereby improving the process.

We note that the Little-JIL language itself was very helpful in this regard. We found that bundling resource specification, exception management, pre- and post-requisites, and artifact flow together in the definition of a step caused interviewers to ask about each of these issues each time the need for a new step was recognized. In asking such questions as "where is this exception handled?" and "what kind of agent is responsible for execution of this step?" important issues were raised, and significant process improvements were made. We have concluded that a language offering rich semantics can be important in suggesting the absence of important details from a process definition and in suggesting the need for elaboration.

The semantic features of Little-JIL were useful in this work. In particular we found that the facilities for handling exceptions were valuable and generally effective in representing exceptional behavior. The facilities for specifying agent types for each step were also useful and important. As we proceeded with the detailed elaboration of the blood transfusion process, the value of abstraction, scoping, and hierarchy became increasingly apparent. While this example gives only a hint of scaling issues, as our process became larger, the problems posed by increasing size became more apparent. Hierarchy is a well-established device for dealing with scaling issues, and its use in Little-JIL underscored that point. But hierarchy in Little-JIL also incorporates the use of abstraction. Thus, for example, specifying the same step in more than one place causes the elaboration of that step, but Little-JIL's use of scoping causes each elaboration to be done in the context of its enclosing scope(s). The previous example indicated how useful this can be.

Thus our experience suggested that a process definition language should offer facilities for abstraction, scoping, hierarchy, exception management, resource specification, and artifact specification—at the very least. This experience also suggested the value of other features not present in Little-JIL, for example transaction semantics and real-time specification.

Finally it seems important to note that the Little-JIL pictorial notation proved to be quite accessible to the medical professionals. Although we expected to find medical professionals unwilling to learn the semantics and iconography of Little-JIL we discovered that within an hour most were relatively comfortable with the language and were becoming increasingly adept at using its features skillfully.

3.2 Property Elicitation

Our work also indicated the importance of eliciting the properties that are required of the process being elicited. We were especially interested in properties that are stated at a high enough level to apply not just to the specific process we had elicited, but to other processes intended to achieve the same goals. In particular, we would like to use finite-state verification not just to detect possible problems with the existing process but also to evaluate proposed modifications to that process. Our experience demonstrated that property elicitation is valuable additionally as another

vehicle for drawing out important process details. We found that it was not uncommon for medical domain experts to specify the details of what they do without having a clear idea of what higher-level goals they are trying to achieve when they perform certain activities in certain ways. By using property specification as a way to place a focus on the goals, motivations, and desiderata for a process, we were often able to cause process performers to think about their processes in a new light, sometimes leading to realizations of possible improvements. In other cases we found that the careful specification of process desiderata, phrased in terms of required or forbidden sequences of steps, led quickly to a realization that some of the steps were missing from the process definition, were misnamed in the process definition, or were used incorrectly in the process definition. Thus, property elicitation also led to improvements in the process. It complemented the focus on "what do you do?" with "why do you do that?" or "what are you really trying to do here?".

We found that Propel was an important aid to the elicitation of precise property specifications. Experience with other projects had demonstrated that it is quite common to specify a property formally in terms of a finite-state automaton or some form of temporal logic, only then to find that important property details were not captured correctly. For example, the property, "A consent form must be signed prior to blood transfusion", leaves unanswered such questions as, "does one consent form suffice for multiple transfusions?" and "can the consent be revoked prior to transfusion?". Propel uses a question tree to automatically pose such questions, thus improving the likelihood that the specified property will correctly reflect the full intent of the person specifying the property. Propel's use on this project supported this conclusion.

3.3 Verification of the Process

Our work on this project is just beginning to employ the FLAVERS finite-state verifier to analyze the blood transfusion process for adherence to some properties. To date we have been able to verify adherence to a small number of properties, most of which have been relatively trivial. There have been numerous verification failures, but most have been due to errors in the process definition itself or the property definition. Although to date we have not yet uncovered serious defects in the process itself, we expect that process defects will start to appear once we begin to verify larger portions of the process and verify them against more stringent properties.

In analyzing larger portions of the process, however, it has become increasingly clear that it is important to employ the services of a reasoning system that can handle this scale. We note that processes, such as blood transfusion, that entail substantial amounts of concurrency and exception handling have accordingly very large execution state spaces, thus making scaling an important issue. Indeed the underlying graph structures that we generated from our process definitions and used as the basis for our finite-state verification often had tens of thousands of nodes and edges. The relative terseness of Little-JIL often serves to mask the size of this state space, but it is this state space that must be explored in order to verify properties.

Our experiences so far suggest that the performance of FLAVERS does seem to scale acceptably well.

4 Related Work

There has been some prior work in using process definition and analysis to improve medical processes. For example, the Protocure II project [15] has goals that are quite similar to ours, but uses a rather different, AI-based, linguistic paradigm for defining processes. Noumeir has also pursued similar goals, but using a notation like UML to define processes [16]. Others (eg. [17]), view medical processes as workflows and use a workflow-like language to define processes and drive their execution. But, we note that these projects seem to place less emphasis on analysis.

There have been other approaches to improving medical safety, as well, but much of the emphasis of this work has been targeted towards quality control measures [5,18], error reporting systems [19], and process automation in laboratory settings [20], such as those where blood products are prepared for administration. In other work, Bayesian belief networks have been used as the basis for discrete event simulations of medical scenarios and to guide treatment planning (eg. [21]).

We note that many languages and diagrammatic notations have been evaluated as vehicles for defining processes. It was suggested that processes be defined using a procedural language [22]. In MARVEL/Oz [23] processes were defined using rules. SLANG [24] used modified Petri Nets to define processes. More recently, the workflow [25] and electronic commerce [26] communities have pursued similar research. This work has shown that some notations aid process understanding, while others provide the semantic rigor needed to support verifying processes to varying degrees of certainty. None, however, seems able to support process definitions that are clear and precise enough. Main failings of these approaches include inadequate specification of exception handling, weak facilities for controlling concurrency, lack of resource management, and inadequate specification of artifact flows.

We also note that there has been a great deal of work on the analysis of software artifacts. Most of this work has been focused on analysis of code or models of systems. Finite-state verification, or model checking, techniques (eg. [8, 9, 27]), work by constructing a finite model that represents all possible executions of the system and then analyzing that model algorithmically to detect executions that violate a particular property specified by the analyst. As noted above one of the major concerns of these techniques is controlling the size of the state-space model, while maintaining precision in the analysis result. Our team has been involved in the analysis and evaluation of various finite-state verification approaches [9], and the development of verifiers such as FLAVERS [9] and INCA [28]. Our work seems to be among the first that has applied FSV approaches to process definitions [10].

5. Extensions of the Work

We have used the blood transfusion process definition to automatically generate a fault tree representation of the process and have used the fault tree to identify single points of failure. This shows the use of process definitions to improve the robustness of a process by identifying and removing single points of failure. Work with chemotherapy processes has confirmed most of the findings stated above. Work with patient flow in the Emergency Department, however, has led to realization of the centrality and complexity of issues pertaining to resources.

We have applied our process improvement approach to processes to a broad range of domains such as labor-management negotiation, elections, and scientific data processing. The work in each domain has shown the need for additional language facilities and a broader research focus, but has confirmed the general applicability of our approach, thus, pointing to the need for interesting complementary work.

In conclusion, we observe that this work has shown considerable promise and has suggested extensions in several directions. We propose to pursue further research in this domain. We expect that this research will lead to notable improvements in the quality of medical processes, and we also expect it to lead to better understandings of how process definition and analysis technology can become key components in the more effective engineering of methods in this critically important domain.

Acknowledgements

This research was supported by the US National Science Foundation under Award Nos. CCR-0204321 and CCR-0205575 and by the U. S. Department of Defense/Army Research Office under Award No. DAAD19-03-1-0133. 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 U.S. National Science Foundation, U. S. Department of Defense/Army Research Office, or the U.S. Government.

The authors gratefully acknowledge the work of Sandy Wise, who had major responsibility for the development of Little-JIL, as well as Barbara Lerner and Aaron Cass, who also made major contributions. Many students participated in the case studies described here, and major contributions were made by Irene Ros, Ethan Katz-Bassett, Huong Phan, M.S. Raunak, and Dave Miller.

References

1. L.T. Kohn, J.M. Corrigan, M.S. Donaldson (Eds). *To Err is Human: Building a Safer Health System*. Washington, DC: National Academy Press, 1999.

2. P.P. Reid, W.D. Compton, J.H. Grossman, G. Fanjiang (Eds). *Building a Better Delivery System: A new Engineering/Healthcare Partnership.* Nat. Academies Press, Washington. DC, 2005.
3. E.H. Henneman, R.L. Cobleigh, K. Frederick, E. Katz-Bassett, G.A. Avrunin, L.A. Clarke, L.J. Osterweil, C. Andrzejewski, K. Merrigan, P.L. Henneman, Increasing patient Safety and Efficiency in Transfusion Therapy using Formal Process Definitions, *Transfusion Medicine Reviews*, **21**, 1, pp. 49-57, January 2007
4. J.L. Callum, H.S. Kaplan, L.L. Merkley, et.al. Near-miss Event Reporting for Transfusion Medicine: Improving Transfusion Safety, *Transfusion*, **41**,1204-1211, 2001.
5. D. Voak, J.F. Chapman, P. Phillips, Quality of transfusion practice beyond the blood transfusion laboratory is essential to prevent ABO-incompatible death. *Transfusion Medicine* **10**: 95-96, 2000.
6. J. Burgmeier, Failure Mode and Effect Analysis: An Application in Reducing Risk in Blood Transfusion. *Quality Improvement* **28**, 331-339, 2002.
7. B. Chen, G.S. Avrunin, L.A. Clarke, L.J. Osterweil, Automatic Fault Tree Derivation from Little-JIL Process Definitions, SPW/PROSIM 2006, Shanghai, China, May 20-22, 2006, Springer-Verlag LNCS. **3966**, pp. 150-158.
8. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV vers. 2: An Open-Source Tool for Symbolic Model Checking, *Computer Aided Verification Conf.*, Springer-Verlag, 2002, 359-365.
9. M.B. Dwyer, L.A. Clarke, J.M. Cobleigh, G. Naumovich, Flow Analysis for Verifying Properties of Concurrent Software Systems. *ACM Trans. on Software Engineering and Methodology*, **13**(4) 359-430, 2004.
10. J.M. Cobleigh, L.A. Clarke, L.J. Osterweil, Verifying Properties of Process Definitions, *ACM SIGSOFT Intl. Symp. on Software Testing & Analysis*, Portland, OR, ACM Press, 2000:96-101
11. J.V. Linden, K. Wagner, A.E. Voytovich, et.al., Transfusion Errors in New York State: An Analysis of 10 Years' Experience, *Transfusion*, **40** (10), 1207-1213, 2000.
12. A.G. Cass, B.S. Lerner, E.K. McCall, et al. Little-JIL/Juliette: A Process Definition Language and Interpreter, *Intl Conf. on Software Engineering*. Limerick, Ireland, 754-758, 2000.
13. A. Wise, Little-JIL 1.5 Language Report, Lab. for Advanced SW Eng. Research (LASER). Dept. of Comp. Sci., UMass, Amherst, Tech. Report, 2006.
14. R.L. Smith, G.S. Avrunin, L.A. Clarke, L.J. Osterweil, PROPEL: An Approach To Supporting Property Elucidation, 24th Intl. Conf. on Software Engineering, Orlando, FL, 11-21, 2002.
15. A. ten Teije, M. Marcos, M. Balser, J. van Croonenborg, C. Duelli, F. van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, A. Seyfang, Improving Medical Protocols by Formal Methods. *Artificial Intell. in Medicine*, **36** (3), 193-209, 2006.
16. R. Noumeir, Radiology interpretation process modeling. *Journal of Biomedical Informatics* **39**(2) 103-114, 2006.
17. M. Ruffolo, R. Curio, L. Gallucci, Process Management in Health Care: A System for Preventing Risks and Medical Errors, *Business Process Mgmt.* 334-343 2005.
18. M.L. Foss, S.B. Moore, Evolution of Quality Management: Integration of Quality Assurance Functions Into Operations, or "Quality is Everyone's Responsibility". *Transfusion* **43** 1330-1336, 2003.
19. J.B. Battles, H.S. Kaplan, T.W. van der Schaaf, C.E. Shea, The Attributes of Medical Event Reporting Systems for Transfusion Medicine. *Arch Pathology Laboratory Medicine* **122**, 231-238, 1998.
20. S.A. Galel, C.A. Richards, Practical Approaches to Improve Laboratory Performance and Transfusion Safety, *Am. J. Clinical Pathology* **107** (Suppl 1):S43-S49, 1997.

21. L.C. van der Gaag, S. Renooji, C.L.M. Witteman, B.M.P. Aleman, B.G. Taal, Probabilities for a Probabilistic Network: A Case-Study in Oesophageal Cancer, *Artificial Intelligence in Medicine*, **25**(2), 123-148.
22. S.M. Sutton Jr., D.M. Heimbigner, L.J. Osterweil, APPL/A: A Language for Software-Process Programming, *ACM Trans. on Software Engineering and Methodology*, **4** (3), 221-286, 1995.
23. I.Z. Ben-Shaul, G. Kaiser, A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment, *16th Intl. Conference on Software Engineering*, 179-188, 1994.
24. S. Bandinelli, A. Fuggetta, C. Ghezzi, Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering* **19**(12) 1993.
25. S. Paul, E. Park, J. Chaar, RainMan: A Workflow System for the Internet, *Usenix Symposium on Internet Technologies and Systems*, 1997.
26. B. Grosof, Y. Labrou, H.Y. Chan, A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML, *ACM Conf. on Electronic Commerce (EC 99)*, Denver, CO, 68-77, 1999.
27. G. J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.
28. J.C. Corbett, G.S. Avrunin, Using Integer Programming to Verify General Safety and Liveness Properties, *Formal Methods in System Design*, **6**, 97-123, 1995.