Desiderata for Languages to be Used in the Definition of Reference Business Processes

Liming Zhu[1,2], Leon J. Osterweil[3], Mark Staples[1,2] Udo Kannengiesser[1,2]
Borislava I. Simidchieva[3]

[1]Empirical Software Engineering, NICTA, Australian Technology Park, NSW, Australia
[2]School of Computer Science and Engineering, University of New South Wales
[3]Laboratory for Advanced Software Engineering Research (LASER),
University of Massachusetts at Amherst, 140 Governors Drive, Amherst, MA 01003
[Liming.Zhu, Mark.Staples, Udo.Kannengiesser]@nicta.com.au, [ljo, bis]@cs.umass.edu

Abstract. In many modern enterprises, explicit business process definitions facilitate the pursuit of business goals in such ways as best practice reuse, process analysis, process efficiency improvement, and automation. Most real-world business processes are large and complex. Successfully capturing, analyzing, and automating these processes requires process definition languages that capture a variety of process aspects with a wealth of details. Most current process modeling languages, such as Business Process Modeling Notation (BPMN), focus on structural control flows among activities while providing inadequate support for other process definition needs. In this paper, we first illustrate these inadequacies through our experiences with a collection of real-world reference business processes from the Australian lending industry. We observe that the most significant inadequacies include lack of resource management, exception handling, process variation, and data flow integration. These identified shortcomings led us to consider the Little-JIL language as a vehicle for defining business processes. Little-JIL addresses the afore-mentioned inadequacies with a number of innovative features. Our investigation concludes that these innovative features are effective in addressing a number of key reference business process definition needs.

1 Introduction

As is the case with most other enterprises in modern society, business continually seeks to get its work done, "faster, better, and cheaper". In pursuit of these high-level abstract goals, the Lending Industry XML Initiative (LIXI) [1], a leading Australian e-Business industry standardization body that serves the lending industry in Australia, asked for help from the Australian National ICT Association (NICTA) in investigating ways in which process specifications might be useful [8, 10, 12]. The LIXI e-business standards are composed of XML-based business data models associated with message exchange patterns. NICTA's goal is to use these data models as the basis for defining business process specifications for LIXI that support highly interoperable and efficient e-Business transactions within the lending industry. It is expected that such models will be effective as the basis for improving the speed and

quality of real estate loan transactions, and in reducing their cost. It is expected that these business process specifications will eventually be mapped to software implementations through Web services or other technologies. Thus, this work also entails producing reference implementations using the Business Process Execution Language (BPEL)[14] and Web services.

While the completion of a real estate loan transaction may seem to be relatively straightforward, some reflection on it soon demonstrates that it is not. In order for real estate to change ownership, the buyer, the seller, the bank that will provide a loan, insurers, valuers, lawyers, real estate registries, and many others must all become involved. There must be much communication among these parties. Some of it must be confidential, some of it must be expedited. In addition a great deal of documentation must be generated, directed, tracked, and ultimately stored. Clearly the processes by which all of this is done are large and complex. But they become even more complex when one considers that there are many ways in which complications can arise. For example, a credit report might be late, or may contain some unwelcome information. The valuer of the property may be unavailable, may come up with a disappointing valuation, or may require a larger fee than originally agreed upon. Indeed, the fact that millions of such transactions take place every year around the world is remarkable, in view of the complexity of the processes entailed. But it is clear that there is room for improvement in speed, quality, and cost reduction.

We note, in particular, that these processes are prone to delays that can be frustrating, that the many parties involved sometimes have misunderstandings about what should be expected from each other, and the various parties (especially the buyer and seller) often lack clear insight into the state of the execution of these processes. LIXI intends to make substantial improvements in this situation, and believes that an approach to doing so can be built upon a set of e-business standards that are composed of XML-based business data models (associated with message exchange patterns), business process specifications, reference architectures, and implementations. A major focus of NICTA's work has been upon the LIXI business process specifications. These are intended to be used as the basis for addressing a number of goals. In particular, it is desired that LIXI process specifications be effective in:

- Capturing standard and best practices in the lending industry at different levels of abstraction, in order to help industry participants become better performers.
- Indicating the specific roles that are played by people, hardware, and software systems within the lending industry eco-system. This might be used to aid in the training of newcomers, or in order to suggest how automation might be injected into the LIXI processes.
- Describing how the various business objects, especially those identified in the industry data standards, are exchanged among process elements of the eco-system and transformed within those process elements.
- Serving as the high-level conceptual bases for the subsequent development of executable business processes, such as web service coordination models and executable models expressed in the Business Process Execution Language (BPEL).

- Serving as the basis for supporting the evaluation of standards compliance for different implementations of capabilities that might be incorporated into the standard processes.
- Suggesting how process steps and resources might be rearranged or reallocated in order to improve efficiency.
- Analyzing sequences of process steps and coordination in order to detect defects.
- Analyzing processes for vulnerabilities to fraud, and violations of privacy and security.

In creating the business process specifications that are needed to support pursuit of these goals, NICTA has encountered a set of process representation issues that present significant challenges. The purpose of this paper is to describe the challenges that NICTA faced, and what NICTA has learned about the characteristics that a process specification notation requires in order to meet those challenges successfully. Specifically, in section 2 we describe the nature of the challenges posed by the LIXI project by indicating how our attempts to specify processes in this domain using the Business Process Modeling Notation (BPMN) [16] have brought out specific deficiencies in BPMN. In section 3 we introduce the Little-JIL process definition language and provide examples of how features of Little-JIL seem to support successfully addressing many of the BPMN deficiencies that had been identified. Section 4 provides a summary and evaluation of our work. Section 5 indicates some related work in process definition.

2 Challenges for Workflow Process Specification

In creating the business process specifications that are needed to support pursuit of the goals of the LIXI project, NICTA has encountered a set of issues that present significant challenges. In particular, our experiences have highlighted the need for a process specification notation that offers facilities sufficient to support the specification of LIXI business processes. In this section we describe the nature of these process specification notation challenges using a specific example to emphasize and clarify many of our points. We then use these observations to indicate some key characteristics required of a process specification notation if it is to be effective in adequately specifying processes such as these.

2.1 An Example: Property Valuation

In this section we describe some details of our attempts to use BPMN to specify a particular LIXI process, namely property valuation.

During a loan application process, a lending organization needs to determine the market value of the loan-related property. This process is called “valuation” and is usually conducted by a valuation firm. The process is typically represented by a workflow specification that starts with an initial valuation request. The request

contains information such as the address of the property, the type of valuation (desktop or curb-side) and the urgency of the request. The request typically progresses through a number of processing stages. These stages are completed largely within the valuation firm, although many of them require services and support from parties outside of the valuation firm. Upon completion of the valuation process, the resulting valuation report is sent back to the requestor.

While this might sound like a rather straightforward process, our exploration of its details has demonstrated that it entails numerous additional complexities. Thus, for example, the process must be open to monitoring and observation as it proceeds. This is because the valuation request and the downstream activities during the valuation must be visible and accessible to other LIXI business processes. In addition, most of the valuation activities are long-lasting business transactions whose execution may require several days. During that time, backchannel messages regarding the progress of the valuation are exchanged upon requests or status changes. Indeed, during that time cancellations, and amending requests, such as requests for fee renegotiations can happen at any time. These activities are essentially subtransactions that are nested in various ways within the valuation transaction. Their cancellations entail compensation activities that can become quite intricate. They may result in variant execution paths and exceptions affecting the current subtransactions, the valuation transaction itself, and potentially the whole gamut of LIXI processes and involved organizations.

Many of the valuation activities involve human-aided interactions, but this may depend upon the automation sophistication of particular participants and nature of the particular activities. Intermediaries who act as proxies of lending institutions and valuation firms may also be involved. Some intermediaries are legal entities that act on behalf of multiple valuation firms or lending institutions. Other intermediaries are simply technical mediators who provide almost transparent infrastructure for transaction mediation. In all of these cases, however, it is essential that the process be able to determine that the performer designated for each activity have the inherent ability to carry out the activity. In case there is no performer qualified to carry out the activity, the process must devise an alternative, or proactively notify a cognizant process participant of the difficulty. We now present details of our efforts to use BPMN to define this process, and we indicate the specific shortcomings that we encountered in doing so.

Figure 1 shows a top-level view of a BPMN representation of the **Valuation** process, using a sub-process (indicated by a "+" symbol) **Check property value** as a black box activity associated with information artifacts representing input (**Prop ID**) and output (**Prop Value**) that are connected to processes by dotted arrows. The starting point of the process is a start event represented as a green circle, and the end point is an end event represented as a red circle. Both start and end events are connected to **Check property value** via solid arrows representing sequence flow. This sub-process has an intermediate event (circle with a thin, double line) **Fee increase requested** attached to its boundary. This event catches exceptions thrown by requests for fee increases that may occur at any time during **Check property value**, which results in the immediate interruption of this activity using the sequence flow connected to **Fee**

increase requested. The exception is subsequently handled through the sub-process **Handle fee increase request**, with specified input (**Fee Request**) and output (**Fee Response**) artifacts. After exception handling, control flows back into **Check property value** to continue with the valuation with either the same or an increased valuation fee. Both sub-processes are located within the pool **Valuation Firm**, which specifies the business entity executing the valuation process.

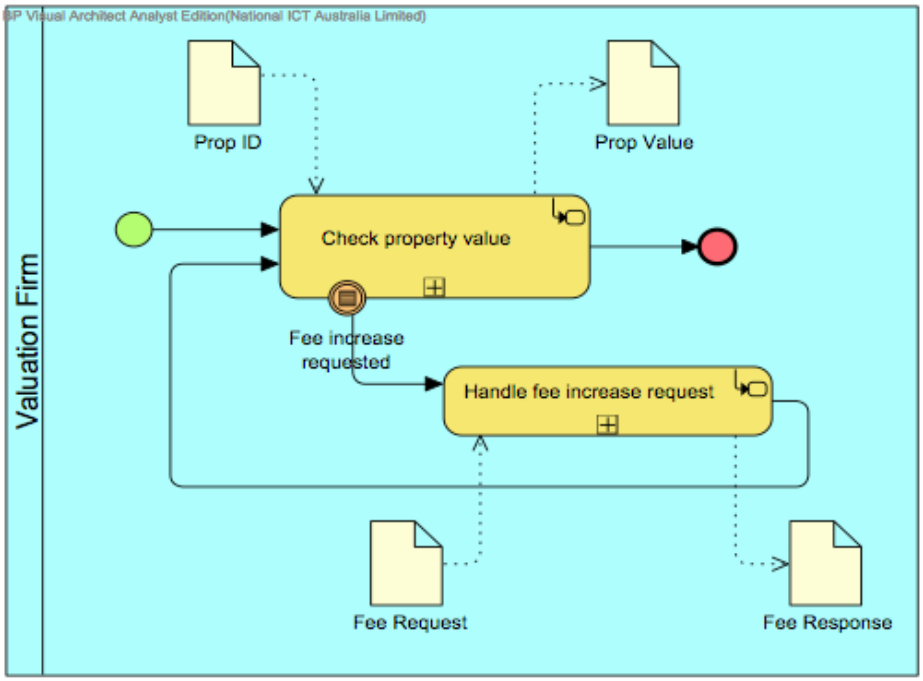

Figure 1. Valuation Process Modeled in BPMN (top-level view)

Figure 2 shows the activities comprising the sub-process **Check property value**. These activities are elementary tasks assigned to different internal roles, represented as different lanes within the pool **Valuation Firm**. The task **Assign valuer** is assigned to **Manager**, and the tasks **Perform inspection** and **Propose valuation response** are assigned to **Valuer**. The task **Propose valuation response** specifies an exception, labeled **Valuation contested**, which is thrown when this task produces a negative outcome in terms of a rejection of the valuation. In this case, the exception is handled by a repeated invocation of the sub-process **Check property value**. Upon completion of the sub-process **Check property value**, control returns to the parent process **Valuation**.

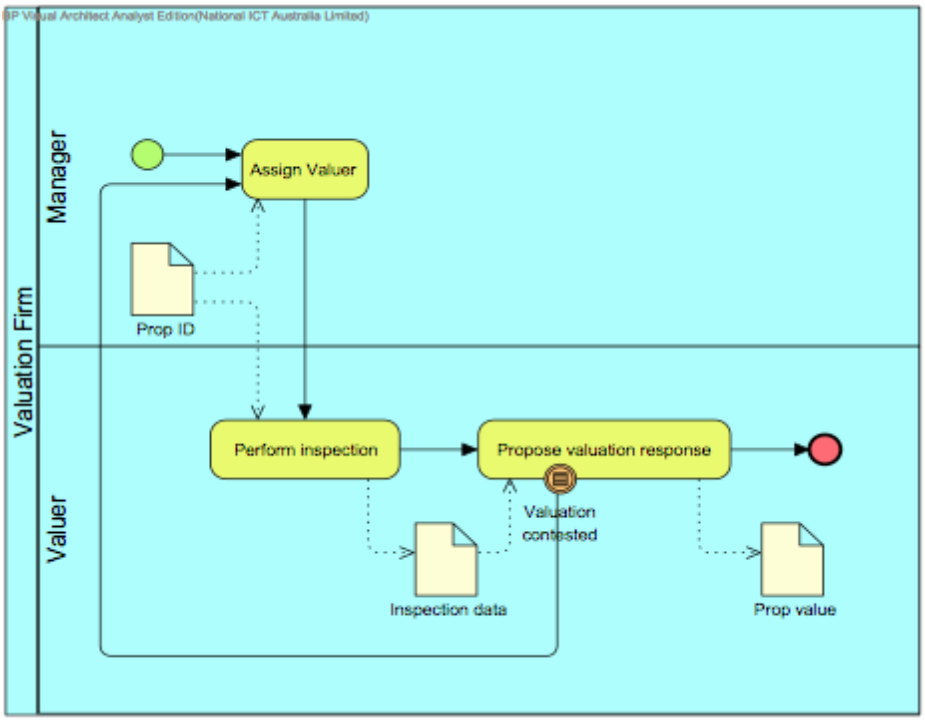

Figure 2. Sub-process "Check property value" modeled in BPMN

Figure 3 shows the details of the **Handle fee increase request** sub-process from Figure 1. It involves two business entities, represented by the pools **Valuation Firm** (with the lane **Manager**) and **Client**. It also involves message passing that is represented using a dashed arrow connecting the end event of the pool **Valuation Firm** with the start event of the pool **Client**. This is also shown through the "envelope" icons within these events. Upon completion of the sub-process **Handle fee increase request**, control returns to the parent process **Valuation**.

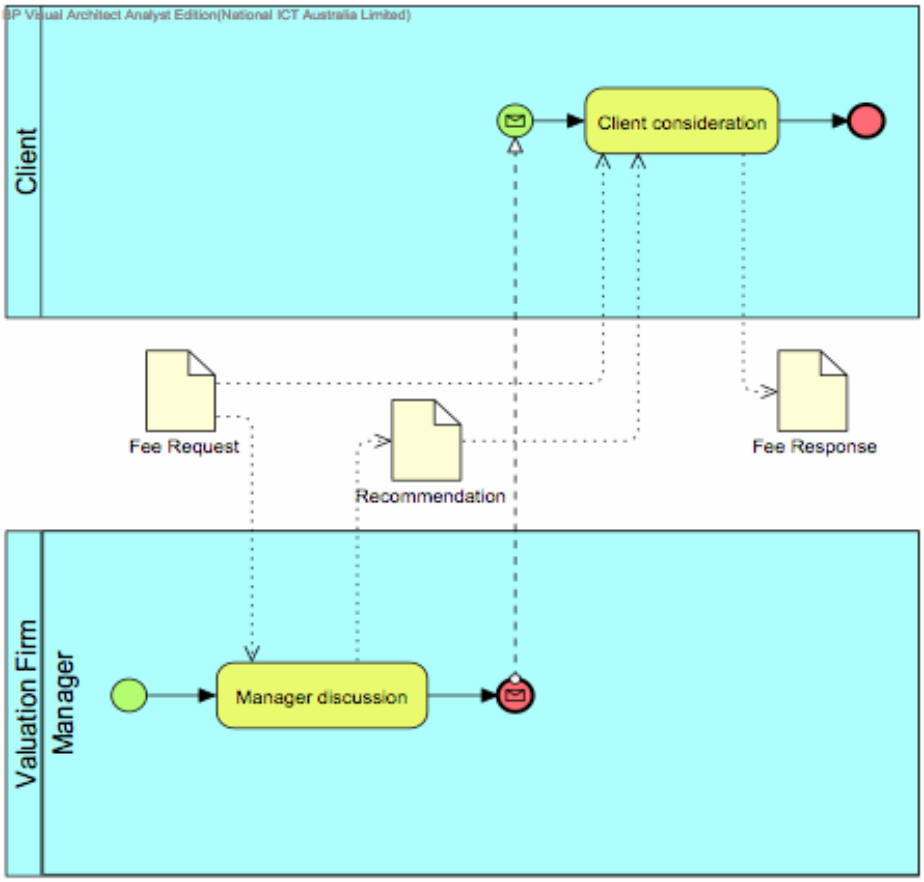

Figure 3. Sub-process "Handle fee increase request" modeled in BPMN

The diagrams shown here do indeed seem to provide a basic view of the valuation process. There are, however, a number of ways in which BPMN has restricted our ability to address many of the complexities that are inherent in this process. We address some of these complexities in the balance of this section.

2.2 Language Semantic Issues.

In defining processes such as **Property Valuation** we were struck by the fact that a wide range of semantic features are needed in order to specify such processes in suitable detail. Identifying the various activities that must be done, and the order in which they are to be done, is a most basic requirement. But far more issues must be addressed if processes such as these are to be specified in detail that is sufficient for them to be used as the basis for definitive reasoning about the presence or absence of defects, efficiency concerns, etc. The following is a brief list of the most salient of these issues.

Dealing with Parties and Agents: One major challenge has to do with the fact that the LIXI eco-system is so diverse that it is difficult to be precise about which kinds of entities are to perform the various activities within LIXI processes. To a considerable extent this is attributable to the fact that the scale of the companies involved varies widely. Some companies are large and have sophisticated information processing systems that can automate most tasks while others lack resources, and thus still rely on fax and manual processing. Because of this, the activities specified in e-Business standards can map to quite different mixes of systems and people. These differences may depend upon resource capabilities, availability, mobility, and rules of allocation of specific process performers to specific activities. These challenges have indicated the need for a flexible abstraction for defining the process performers, how they are to be allocated to activities, when they are to be allowed autonomy (and when they will not be allowed autonomy), and the ways in which they are mandated to participate in defined processes.

In BPMN this challenging set of representation requirements must be met by using fixed role-based swim lanes [23] to define the participation of parties in processes. As there is no other way to indicate the participation of parties in a process it is difficult or impossible to indicate how alternative parties might be selected to perform a task should a primary party be unavailable. Thus, in our example Figure 2 shows that the **Manager** has the task of assigning the **Valuation** task to the **Valuer**. Assuming that there is only one **Manager** and one **Valuer**, this specification may be sufficient. And assuming that both parties are available to carry out these tasks, no complications should be expected to arise. But these sorts of complications might well arise. In particular it is easy to see that there may be many parties capable of carrying out a **Valuation**, and the **Manager** may wish to use some criteria such as cost, skill, and availability to select from among them. BPMN offers weak facilities for specifying how such a selection might be carried out and how the binding of the selected performer is to be depicted. BPMN may be used, for example, to specify some sort of auction as the basis for such a selection, but the static nature of this language would restrict the bidders in such an auction to a predefined set of **Valuers**. This seems unduly restrictive. We believe that this example demonstrates the need for capability-based and other types of rule-based mechanisms for the selection from among a set of performers whose members may vary even as the performance of the process proceeds. This, in turn, relies upon the availability of late-binding semantics for binding the performer that is selected for binding to an activity.

Exception Management: The LIXI processes we have worked with are typically characterized by the need to deal effectively and carefully with a range of conditions that, while not unexpected, cause execution of the process to have to deviate from the normative flow used to handle trouble-free cases. These exceptional conditions may run the gamut from lack of needed resources, to unwelcome outcomes from prior step executions, to the violation of stringent timing constraints. In all such cases, it is necessary to clearly identify the exceptional condition, predict the place(s) where it may occur, and to specify precisely how the exceptional condition is to be handled. The need to do this has typically caused process specifications that were expected to be clear and terse to become large, complicated, and often hard to understand.

Capabilities such as scoping and dynamic exception handler determination seem to be useful in addressing these needs, but we have found BPMN's support of these capabilities to be inadequate. Similarly, the ability to specify arguments to exception handlers and how to proceed with further execution after an exception is handled seem useful in addressing the exception handling needs we have encountered. In BPMN, however, these capabilities are limited to structural control flows and some forms of resource re-allocation (still solely through the use of statically specified role-based capabilities [21]).

Thus, for example, in Figure 1, the exception caught by the intermediate event does not have any arguments associated with it. Although the diagram includes an input artifact connected to **Handle fee increase request**, it is not clear how this artifact relates to the exception. After handling the exception, there is no formal way of specifying that control is to return to the location within **Check property value** where it was thrown. These observations have focused our attention on the need for language features that can support the clear specification of such exceptions, as well as the clear specification of how they are to be dealt with. Such language capabilities seem essential to support the clear and precise specification of processes such as those arising in the LIXI context.

Elaboration and Abstraction: There are many difficulties in deciding the level at which LIXI processes should be defined. We found that a manager may have an idealized view of what is necessary for a process to be performed acceptably well, but that those charged with the actual performance of a task may understand the need to perform some tasks, and some sequences of tasks, differently. In such cases, this difficulty can be resolved by reaching agreements about goals for the performance of a part of a process, and then allowing performers of its activities to pursue the goals in their own individualized ways. This suggests the value of a language that can support the incremental provision of elaborative details of a subprocess, but can also hide those details as is done with programming language procedural abstractions.

In addition, as a business-to-business process standard, LIXI business processes are required to specify cross-boundary transactions and should thus leave private business processes behind organizational boundaries where they can provide their owners with competitive advantage. However, we found real difficulties in deciding where to draw those organizational boundaries. Indeed there is a subtle balance between promoting interoperability through prescriptive standards and allowing innovation through minimal prescription. Thus, this also argues for the use of a language that supports flexibility in decisions about how and where to add elaborative detail, and in supporting flexibility in deciding which process information is disclosed and exchanged.

Thus, levels of abstraction are of paramount importance in the specification of LIXI business processes. Abstraction specification capabilities must enable specification of references to internal private business processes and detailed process implementations as well as communications among different stakeholders. Every activity must be decomposable. BPMN offers abstraction only through sub-workflows, which do not

offer adequate abstraction and decomposition semantics. Thus, for example, sub-workflows do not incorporate semantics that enable the subprocess to be specialized or tailored in accordance with different contexts that may be established by the processes within which they are elaborated. In particular, BPMN does not incorporate parameter binding semantics that could be used to implement context-sensitive elaboration. The desirability of such process specification language features was clearly indicated by our work on specifying LIXI processes.

Reconciliation of Control and Data Flow. As noted above, LIXI seeks to develop both business process specifications, and data model standards that are based upon the consistent use of a controlled vocabulary and XML schemas. There is then a resultant need by LIXI managers, participants, and developers for facilities that assure that these representations are consistent with each other. At the least, LIXI requires support for generating cross-referencing of the data artifacts that are used to annotate process specification control flows with the data items defined in the LIXI data models. LIXI process representations should be able to express the precise data and artifacts used by each activity using names and notations that are consistent with those in the LIXI data models.

BPMN creates some obstacles to doing this clearly and completely. In particular, BPMN requires that the flow of artifacts between activities can be specified only as annotations on control flow edges, and can take place only within defined scopes. Thus, in cases (such as are not uncommon in the LIXI processes we have tried to specify) where data artifacts must flow between two activities that are distant from each other in terms of process control flow, it is necessary to show the artifacts moving through many activities that do not deal with these artifacts. In some cases, the artifacts should indeed not even be seen by these activities in order to preserve their confidentiality. Thus, for example, in our valuation example we note that there are situations in which the valuer may need to pass or receive a message from an entity that is not an active participant in the valuation process (eg. the valuer's manager). BPMN requires that such messages pass hierarchically, which requires the passing of such messages through activities that have no need for the information, and that may indeed violate privacy requirements by having access to the information. Clearly, it would be desirable to incorporate a more flexible specification mechanism to allow specifying that data artifacts must flow directly between activities, regardless of their data flow proximity.

In addition, we note that a LIXI data model will necessarily have to indicate the decomposition of larger data aggregates into smaller subunits. Indeed the consolidation and decomposition of such larger aggregates can be expected to take place through process control. Unfortunately, BPMN lacks facilities for showing how process activities perform such consolidation and decomposition, as it treats all data artifacts as being atomic. Thus, for example, in valuation it is usually the case that many factors are taken into consideration, and the eventual valuation is obtained by considering, consolidating, and structuring large amounts of information. The quantitative value of a property is typically supported by a large quantity of highly structured information including details of the property, its history, surroundings, and

market forces. Ideally BPMN would be useful in indicating how the various activities and subprocess structures are used to create those many items of information and structure them into the final artifact aggregates. But as BPMN supports only the annotation of atomic data artifacts, it is quite difficult to use BPMN to define the way in which structures of BPMN activities contribute to the creation of (potentially different) complex structures of artifacts.

Transaction Management: The LIXI business processes are large, complex and long lasting. A real estate loan process lasts weeks if not months during its application stage and requires servicing and refinancing spanning decades. The long lasting nature demands process definition capabilities for being precise about the constraints that must be met and about compensational policies that are needed when these constraints are not met. Indeed, many LIXI processes are best thought of as long transactions. These long-lasting transactions are often decomposed into sets of nested subtransactions that may themselves be long-lasting [22]. In such cases, the compensation policies may become particularly complex, and may vary depending upon contexts and scopes. This is complicated further by the fact that some LIXI transactions are performed by participants from different groups and organizations. This creates a more acute need for clear communication and understandings of such issues as rollback and compensation.

All of this points to the need for process language features that can support the specification of long transactions and nested transactions. The lack of such features as integral parts of BPMN required that we attempt to create them out of the semantic features that are available. The resulting process definitions are less clear, precise, and accurate because of this.

Variation: The variety of participants in these processes leads to still further difficulties in that we have found that these participants often disagree about both the nature of the process and their objectives for it. Thus, for example, it is not unusual for parties in LIXI to want complexity to reside in others' parts of the overall process, and to want information to be shared, but they often do not want to share their own information. Technical solution companies provide and favor intermediary gateways and custom-built applications, while smaller parties typically want commoditized applications and to remove intermediaries. This makes it very difficult for a LIXI reference business process to be defined by a single structured static process. Thus, attempts to arrive at one fixed all-encompassing process definition often seemed unrealistic, leading us to believe that instead it might be preferable to define a collection of variants on a core process, or a process family, in order to span an expectable spectrum of variation in processes that are caused by factors such as environmental constraints.

In BPMN, however, expressiveness of process variability is limited to static structural enumeration of possibilities. Some amount of variation can indeed be achieved through parameterization of the specification of activities, achieved through implicit data sharing. But this is a relatively weak capability. Far more preferable would be an approach that supports the specification of overall process properties or

pre/post/invariants for individual activities, and the use of such properties and invariants to generate variants. The absence of such features from BPMN limits its power to specify the breadth of process variation that our experiences in LIXI indicate to be necessary.

Observation and Monitoring: There are a number of reasons why LIXI processes must be readily observable. First, as noted above, the participants in the processes (especially the buyer and seller) have strong interests in seeing that the needed transactions are taking place in a timely fashion. In addition, however, as the real estate industry is subject to considerable amounts of governmental supervision, it is also important for regulators to have clear visibility into the transactions. For these reasons, it is important for LIXI processes to incorporate features that can facilitate such scrutiny. Features that support the checking of adherence to stated properties are useful in meeting such needs. The flexibility to insert such checks at specified strategic points in the process also supports such needs.

Timing: As has been noted, a key goal of this project is to improve the speed at which LIXI processes execute, thereby reducing the amount of time required to conclude real estate transactions. Thus it seems essential to include as part of the definitions of LIXI processes specifications about time. Specification of the time limits on executions of steps in the process seems important, and mechanisms for dealing with violations of those time limits also seem important. BPMN incorporates some support for specification of timing considerations. Our experience, however, is that the existing capabilities are not as powerful as would be desirable to support specification of LIXI processes.

Scaling to handle Size and Complexity: We have found that the processes to be specified for LIXI are invariably very large and very complex. Although we had expected the processes to be large, encompassing many steps, dealing with a wide range of artifacts, and entailing the coordination of many types of participants. We were surprised to find that the processes were larger still. Indeed much of the apparent size of these processes seems to stem from the large number of intricacies in dealing with the many complications that may arise in carrying out a process that seems quite straightforward, until examined in close detail. This observation underscores the importance of using a process specification language that can scale up to support the specification of large and complex processes.

2.3 Other Critical Issues

If, as previously suggested, the top-level goals of our work are to help the Australian real estate loan industry carry out its work “faster, better, and cheaper”, then the foregoing list of process specification semantic issues is only the beginning of a list of desiderata for specification notations. Incorporation of these semantic features into a suitable process specification language should enable us to specify LIXI processes acceptably well. But it is important to recognize that such a specification, while useful and important in its own right, must also serve as the basis for other activities, such as

analysis. The need to support these additional activities creates additional desiderata for the process specification language. In this section we address two such desiderata

Clarity: The preceding list of semantic features needed in order to support the adequate specification of LIXI processes suggests the desirability of a large and complex process specification language. But it is important to also recognize the need for such a language to be clear and transparent. Ultimately the processes that we define in this project must be made sufficiently accessible to loan industry representatives that they can certify that the specifications are accurate reflections of the actual processes. This establishes a requirement that the process specifications be readable and understandable by people who may lack a strong background in technology.

We have found that the elicitation of LIXI processes is an iterative process in which specifiers read documents and interview domain experts first, then create specifications of the processes, but then must validate that their specifications are accurate. Validation of the process specifications entails presenting them to the domain experts in order to elicit either confirmation of accuracy, or observation of inaccuracies. Either requires that the domain experts are able to grasp details of the process specifications. It is simply not possible for this to happen if the process specifications are in notations that cannot be understood by loan industry experts.

These observations strongly suggest the value of graphical or pictorial specifications. Indeed we have noted that many of the most commonly used process specification notations encompass at least some graphical notations.

Rigor: While the use of some sort of graphical notation seems important in order to support clarity and comprehensibility, this must not be at the expense of sufficient rigor to support reasoning. We note that quality and speed improvements are key goals of this project. Thus we must be able to subject process specifications to analyses whose objectives include determination of the presence or absence of defects and the presence or absence of such timing concerns as bottlenecks. There are a large number of analytic approaches that can be used to reason about process specifications, but if the results of such analytic reasoning are to be definitive, then the process specifications themselves must be stated in terms of a rigorous formalism[6, 7]. Lack of rigor in process specifications leads to lack of definitiveness in analytic results.

In this paper we advocate the use of process specifications that are stated through the use of a process specification notation that is well defined, and suitable for rigorous analyses that yield definitive results. We distinguish such process specifications by referring to them as process definitions, intending to emphasize that process definitions are process specifications that are stated in terms of a rigorously defined specification notation.

3 Little-JIL

In order to address the business process definition needs we have identified, we sought more powerful process definition languages from the research community. In this section we demonstrate the potential contribution of such a research process definition language. This section introduces the Little-JIL process definition language and then demonstrates its use in defining the valuation process. The section concludes with a summary of the ways in which this language offers advantages over BPMN in the areas noted in section 2.

3.1 An overview of Little-JIL:

Little-JIL is a language originally developed for defining the processes by which software is developed and maintained. Wise [28] provides full technical details of the language. A Little-JIL process is defined by specifying 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 that are available to support performance of the activities. The coordination specification ties these together by specifying precisely which agents, aided by which supplementary capabilities, will perform which activities upon which artifacts at which exact time(s). Because of its central role in specifying all of this, the coordination diagram is generally the central focus of a Little-JIL process definition.

A Little-JIL coordination diagram is depicted as a hierarchical decomposition of steps, although a Little-JIL step definition is best thought of as a procedural abstraction. Each step has a name and a set of badges to represent various of its features. Thus, for example, a step's interface badge represents the type of agent that is to be responsible for execution of the step. The interface badge also represents the flow of arguments between the step and its parent, and the step and its children. Other badges represent control flow among sub-steps, the exceptions the step 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 definition.

We have found that a number of features of Little-JIL make it rather unique, and are particularly useful in addressing the needs of the LIXI project. Among the key features of Little-JIL that distinguish it from most process specification languages are 1) its use of abstraction to support scalability and clarity, 2) its use of scoping to make the use of step parameterization clear, 3) its facilities for specifying both artifact and data flow in a single notation, 4) its extensive capabilities for defining how to handle exceptional conditions, and 5) the clarity with which iteration can be specified and controlled. These capabilities allow Little-JIL to address many of the previously enumerated process definition needs relatively more successfully. To make this clearer, we now show how Little-JIL can be used to define part of the valuation process that was introduced in the previous section of this paper.

3.2 A Little-JIL Definition of the Valuation Process:

The purpose of this process, as noted above, is to show how managers, valuers, and clients collaborate to arrive at a valuation of a property. As will be seen, this entails some iteration, and must take into account the participation of various agents, as well as the possibility of handling contingencies of various kinds. Figure 4 defines the high level of this process, which we refer to as **Check Property Value**. We define this as a hierarchical decomposition into three substeps, **Assign Valuer, Perform Inspection,** and **Propose Valuation Response**. The right arrow in the **Check Property Value** step bar indicates that these three substeps are to be executed in sequence. Little-JIL also supports specifying that substeps can be executed in parallel (denoted by two parallel lines in the step bar), and the **Perform Inspection** substep (elaborated in Figure 5) is an example. The substeps of a parallel step can be executed in any order, including any arbitrary interleaving.

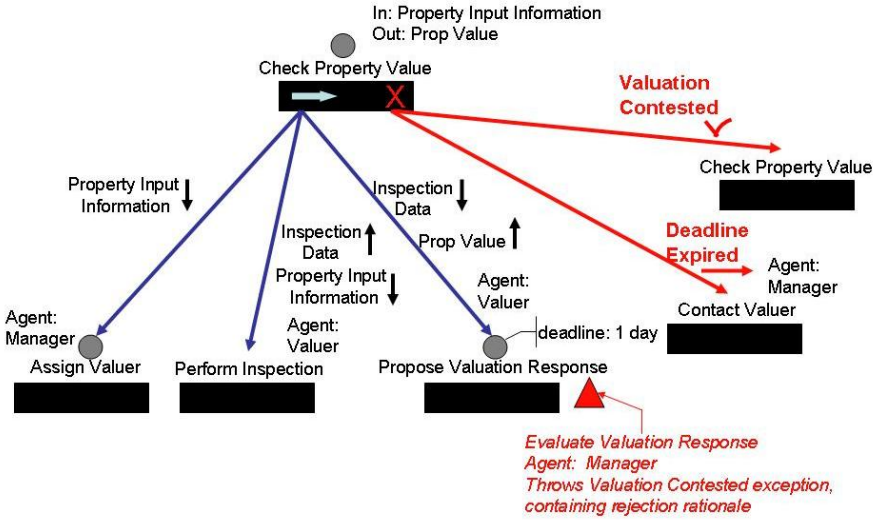

Figure 4. Valuation process modeled in Little-JIL

As is the case in BPMN, annotations on the edges (in the case of Little-JIL these edges are those that are between the parent step and its children) can be used to define the flow of process artifacts (but, note that the flow of artifacts can also be specified in other ways, as we shall see shortly). Thus, in particular, note that the process begins with the flow of an artifact, **Property Input Information,** from **Check Property Value** to **Assign Valuer** (the downward arrow indicates that this artifact is passed from parent to child). Execution of the next substep, **Perform Inspection,** also receives **Property Input Information** as an input, and in addition produces **Inspection Data** as an output (note the upward pointing arrow next to this artifact).

Note that the elaborated **Perform Inspection** step in Figure 5 also has a channel declaration, **channel** (denoted by the two-sided arrow), that carries artifacts that are of type **Data** (denoted by : **Data**). Channels allow for communication between steps that are not directly connected in the coordination hierarchy and are accessible by all steps within the scope of the step where they are declared, in this case the **Add Inspection Data Item** and **Compile Inspection Data** steps. The first substep, **Add Inspection Data Item**, writes to the channel, and the latter, **Compile Inspection Data**, reads from it (Little-JIL incorporates annotations to specify this, but these annotations are omitted from this diagram for the sake of avoiding clutter).

Thus, **Add Inspection Data Item** receives **Property Input Information** as input from its parent and produced a **datum** of type **Data** as output. However, instead of passing the output to its parent, **Add Inspection Data Item** writes the output to **channel**, from which **Compile Inspection Data** reads it as input. Note that this causes this data to flow directly between these two steps. This enables the process to keep the data from being accessible by the **Perform Inspection** step, which could entail a violation of privacy requirements. On the other hand, note that **Compile Inspection Data** produces **Inspection Data** as output, but in this case, the process definition specifies that this artifact is passed on to its parent step making use of hierarchical data flow specification semantics.

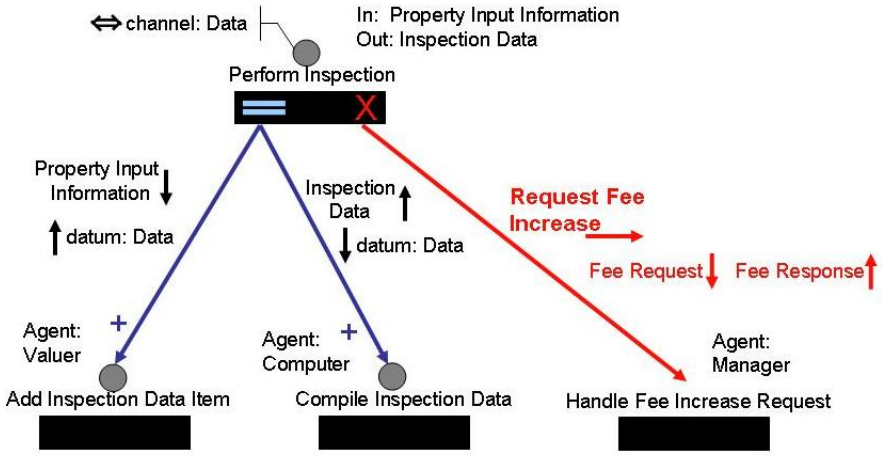

Figure 5. Perform Inspection step elaboration

Note also that the edges of both of these steps have an additional annotation, namely a plus sign. This indicates that each of these steps may be executed multiple times and must be executed at least once. Thus, **Perform Inspection** effectively consists of adding single inspection data items to a channel in the **Add Inspection Data Item** step and pulling these data from the **channel** and compiling them in the **Compile**

Inspection Data step. This is an example of how Little-JIL can be used to define the way in a higher-level data aggregate can be built by composing it out of lower-level artifacts. The structure of the higher-level aggregate in this example is not specified directly, but can be specified by elaboration of the details of the **Compile Inspection Data** step. Note that Little-JIL can be used to define the process for constructing complex aggregates whose structure might be arbitrarily complex, as defined in the artifact repository that is an integral part of a Little-JIL process definition.

Note also that the specification of artifact flows in Little-JIL is done both by annotating edges as shown, and also by specifying the input/output behavior of a step by attaching to the step's external interface icon (the round circle atop each step) an enumeration of the artifacts that are taken as input artifacts, and those that are produced as output artifacts. As noted above, it is useful to think of a Little-JIL step definition as an abstract procedural specification. Thus these annotations can be thought of as parameter specifications that, when instantiated and bound to the procedural abstraction transform it into a procedure invocation instance, with the artifact instances being taken as the arguments to the procedure instance. This is a principal abstraction mechanism in Little-JIL. As with typical abstraction mechanisms, it facilitates the reuse of procedural specification information, and serves to reduce the size of process definitions, thus supporting scalability.

Figure 4 also shows that each step is annotated with a specification of the type of agent that is required in order to perform the step. Thus, note that the agent for **Assign Valuer** is of type **Manager**, while the agent for **Perform Inspection** is of type **Valuer**. In all cases, these specifications indicate the type, rather than the specific instance, that is required to perform the step. The type specification is passed at runtime by the Little-JIL interpreter to a resource repository that uses the specification to select the specific agent that will be assigned to perform the step.

Both Figure 4 and 5 also show that each step incorporates the ability to handle exceptions. Thus, note that **Check Property Value** and **Perform Inspection** each has substeps, connected to the parent by red edges emanating from the red X on the right of the step bar. Each such edge is annotated (using bold face type) by the type of the exception that triggers execution of the step on its end.

Note, for example that an exception handler substep of the **Perform Inspection** step in Figure 5 is in place to handle the **Request Fee Increase** exception. In this case, the exception is raised by the execution of one of the substeps of **Perform Inspection** and arises when the agent decides that the fee offered for the valuation is inadequate. This exception is handled by another step, namely the **Handle Fee Increase Request** step, and in this case we see that its invocation has defined arguments, namely **Fee Request** (as an input) and **Fee Response** (as an output). Finally note that the specification of the type of the exception also includes an arrow that points to the right, which indicates that, once the exception has been handled, control returns to the location where the exception was raised. This is as needed, indicating that the outcome of the request was a response to the request for a fee increase, at which time execution must resume. This also demonstrates how Little-JIL makes use of scoping

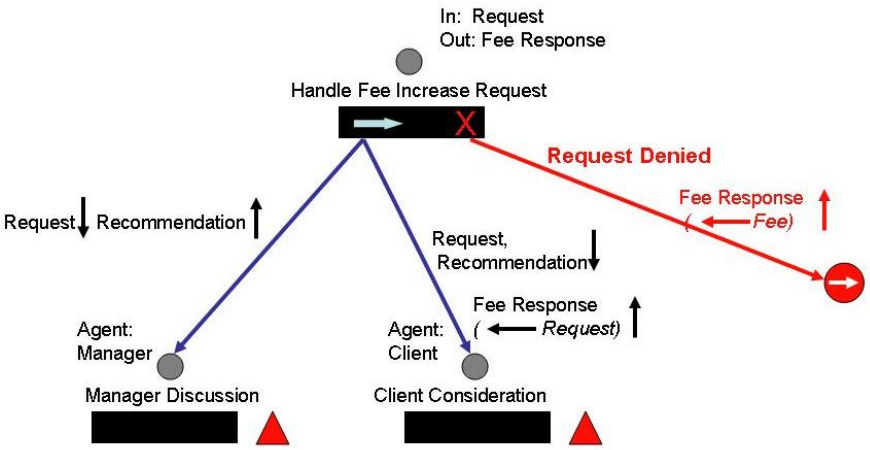

Figure 6. Handle Fee Increase Request modeled in Little-JIL

for precise and powerful exception handling—since the exception in thrown in the context of **Perform Inspection**, it is also handled within that scope instead of at the root step.

Note also that **Handle Fee Increase Request** is a step that is defined in further detail by the diagram shown in Figure 6, illustrating the use of hierarchical decomposition, but also indicating that the elaboration is indeed an invocation of a procedure, taking the indicated artifacts as its arguments. Thus, note that the annotations on the circular external interface icon show that the arguments are assigned as the values of the parameters, **Request** (an input), and **Fee Response** (an output). This step is decomposed into two substeps, and one exception handler substep. The first substep is **Manager Discussion**, and the second substep is **Client Consideration**. Each is labeled with an appropriate type of agent and the appropriate artifact flow specifications. Note that each has a postrequisite (denoted by a red triangle pointing upward) that specifies what is to be done in case the agent does not go along with the request. Each postrequisite throws the **Request Denied** exception, which is handled as the invocation of the procedure that is represented by the appropriately labeled exception handler. In this case, there is no step required to handle the exception, only the binding of the value of the **Fee** artifact to the **Fee Response** artifact (this is indicated by the documentation shown in italics), and by the right arrow at the end of the exception edge, which indicates that after this binding, control continues as though the parent step, **Handle Fee Increase Request**, has concluded, at which time control is returned to the part of the process depicted in Figure 5. Finally note that if neither substep raises an exception, then the second substep, **Client Consideration**, terminates by binding the value of the **Request** argument artifact as the value of the **Fee Response** artifact, properly reflecting that the request has been approved.

The diagram in Figure 4 demonstrates yet another important feature of Little-JIL, namely the ability to assign deadlines to individual steps in order to specify timing requirements and limitations. In this case, the **Propose Valuation Response** step has a deadline associated with it, stating that the step must complete within one day of the time at which execution of the step started. If the deadline expires and the step has not completed, a **Deadline Expired** exception is thrown. In this case, the exception is handled by having the **Manager** contact the **Valuer** to discuss the delay and then, at the discretion of the **Manager,** recursively invoking the **Propose Valuation Response** step. Note that such a recursive invocation will occur within a context that is set by the prior failure(s) of the step on previous invocation(s). In this case, the **Manager** is giving the **Valuer** another day to complete the **Valuation Response**. The process specifies as an alternative (note that this exception handler, which is not shown here due to lack of space, is defined as a choice step) that the **Manager** could also recursively invoke the **Check Property Value** step (i.e. thereby reassigning the valuation to a different **Valuer)**. In either case, the process specifies that all information about the **Valuer** agent who missed the deadline is appended to the **Property Input Information** artifact in order to provide more complete information about the context in which the reassignment is made. This example also illustrates the availability of context in Little-JIL's exception handling mechanism. Thus, for example, when the **Check Property Value** step is reinvoked, the information about the **Valuer** who missed the deadline is part of the invocation context, and so it is available to the **Manager.** Presumably when he or she executes the **Assign Valuer** step within the new context, this contextual information will be used to assure that the job is not reassigned to the original **Valuer**.

This availability of context is also apparent in the case of a **Valuation Contested** exception being thrown, in this case by executing the postrequisite (indicated by a red upward pointing triangle) of the **Propose Valuation Response** step and getting a negative outcome (i.e. that that valuation has been rejected). Figure 4 has attached to it an informal comment (indicated by the use of italics) noting that this postrequsite is in fact an entire step, **Evaluate Valuation Response**, and that it is to be executed by an agent of type **Manager**. When a **Valuation Contested** exception is thrown, information about the rejection is included in **rejection rationale**. The response to this outcome is the execution of the step at the end of the exception handling step bearing the **Valuation Contested** label. Note that this step is a recursive invocation of the **Check Property Value** step, which passes as input the original information contained in **Property Input Information**, appended with the **rejection rationale**. Thus, the step is reinvoked in the context of the rejection of the valuation, thereby enabling the agent assigned to perform the reevaluation to understand that this is being done in the context of this rejection. Finally note that there is a check mark under the **Valuation Contested** label, which indicates that once the exception has been handled, control continues as though the parent, **Check Property Value**, has completed. This also effects the rolling up of the nested recursion.

In this short example, we see that Little-JIL is capable of clearly and tersely capturing very complex control flow variation, and meshing it successfully with artifact flow. The handling of the request for a fee increase indicates this nicely, showing how a

variety of decisions will affect both data and control flow. It also shows how powerful exception handling helps a great deal. In this case we show how the exception can arise in different places, but is directed to the same exception handler, which itself has further exception handling capability. Despite this structure, it is clear how execution is guided back to the right location before proceeding.

3.3 Comparison to BPMN

The preceding discussion of the Little-JIL definition of the **Valuation** process has suggested some ways in which this language offers advantages. In this section we summarize and expand upon those advantages, addressing the issue areas raised in section 2.

Dealing with Parties and Agents: As noted above, BPMN does not have facilities for stating clear definitions of resources. Thus, its mechanism for specifying agents is very primitive. In particular, there is no way to specify dynamic agent allocation based on capabilities; rather, there is a static 'role' feature, achieved through pools (e.g. for different organizations) or swimlanes (e.g. for different departments within an organization). Agent specification is very important in our highly complex LIXI processes since dynamic allocation of agents based on their capabilities and the execution context facilitates resource management and optimization. By enabling substitutability of agents at run-time, the process can often proceed satisfactorily even if a specific agent is unavailable, as long as there is another agent who has the necessary capabilities to provide the services requested. This feature is very important for modelling real business processes since task assignment changes occur regularly in practice and should be considered as an integral part of the process.

Our example has shown that dynamic assignment of agents to steps in Little-JIL is an inherent part of this language. Each Little-JIL step must incorporate a specification of the capabilities required for its completion. One such specification is of the agent, namely the entity responsible for execution of the step. As this specification is a type, rather than an instance, flexibility is available at execution of the process. This flexibility resides in a resource manager whose job is to receive a request for a type of agent and then use that request to query its base of resource instances for one that is suitable for meeting the needs of the request. As this is done at runtime, the selection of the agent instance can reflect the state of the process execution, details of availability of agents, etc.

BPMN makes use of message passing to show artifact flow, but the addition of artifacts do not alter the process with respect to its meanings or analysis. Thus, additional event constructs have to be used to ascertain process properties relating to the dataflow. Artifact flow and resources are an integral part of a Little-JIL process definition and significantly alter the process meaning.

Exception Management. In the previous section we noted that LIXI processes, like most other processes, must often deviate from straightforward processing in order to deal with contingencies and abnormal situations. Doing so is non-trivial, involving at

least 1) the recognition of the need to deviate from normal processing, 2) a prescription for how to deal with the abnormal situation, and 3) specification of how to resume processing after handling of the abnormality. Our example contains only a small sample of such exceptional situations. We found that LIXI processes have a plethora of them. As noted above, BPMN has very few facilities for dealing with exceptions, thereby making it difficult to specify actual processes in the needed full detail.

Little-JIL provides powerful exception-handling mechanisms. As noted above any step may throw an exception. In addition each step may (optionally) be preceded by a pre-requisite and/or followed by a post-requisite. Each requisite can also throw an exception. Exceptions in Little-JIL are typed. Thus a step and/or requisite can potentially throw more than one type of exception. Exceptions are caught by exception handlers that are also typed in order to assure that the right handler handles the right exceptions. Exception handlers in Little-JIL are defined as substeps of a parent step and handle exceptions that arise in the scope that is defined by the parent step. Thus Little-JIL exception handlers are context-aware, and can provide more tailored and specific exception handling services. Parameters are also thrown when a Little-JIL exception occurs, thereby enabling the exception handler to be given details of the exceptional situation, and adding to its versatility and ability to provide tailored responses to the exceptional situation. In addition, Little-JIL offers a choice of four different continuation semantics (*complete*, *continue*, *rethrow*, and *restart*, two of which--*complete* and *continue*--are illustrated in Figure 4), allowing the specification of four different ways in which execution can continue after the exception has been handled.

Elaboration and Abstraction: BPMN provides good support for elaboration in the form of subprocesses, which are effective in allowing for the incremental provision of increasing levels of process detail. We note, however, that BPMN process activities and subprocesses are not true abstractions. Although it might be desirable to instantiate them in multiple usage contexts, in which their functioning would be adapted to the context, this is not possible in BPMN. BPMN subprocesses do not provide capabilities for scoping, parameter passing and control from parent processes. Ultimately, this impairs the readability of the flowchart format that BPMN uses.

Processes in Little-JIL are also hierarchical decompositions of steps. But Little-JIL steps function very much as procedures. Thus, a Little-JIL step definition can be reinvoked multiple times, and from different contexts. The step definition incorporates a specification of the input and output artifacts, which function very much as procedure parameters. Each step invocation establishes a context, and contexts may differ in a variety of ways. For example each step invocation provides a different collection of arguments that are to be passed into and out of the step. Different invocations also provide different sets of exception handlers, thereby supporting the specification of different reactions to a given exception type in different contexts. Different step invocations may also have different agents.

The availability of powerful abstraction capabilities in Little-JIL provides the expected advantages of comprehensibility and reusability. Reusability in turn leads to terseness without loss of clarity, and hence to scalability. Our experiences in using Little-JIL seem to have benefited from these advantages.

Reconciliation of Control and Data Flow. In the previous section we noted that BPMN has limited capabilities for integrating data flow specifications. Data flows can be represented only as annotations along control flow edges in BPMN. In Little-JIL, however, the artifacts that annotate edges are arguments to the procedures represented by the step at the bottom of the edge. This provides a strong basis for understanding just how the process activities are dealing with artifacts. Indeed, substeps are essentially subprocedures in Little-JIL, which makes it particularly convenient to represent the way in which lower level activities deal with the components of higher level artifacts that are aggregates. As noted above BPMN does not support such representation of how its activities deal with data aggregates and their components.

In addition, it has been noted that the complexity of some processes sometimes requires that artifacts be shared between activities that are not directly connected within a BPMN coordination diagram. Creating artificial scopes in order to allow for such sharing of data results in the data being available to intermediate activities, which may be undesirable, and be confusing to read. Instead, Little-JIL provides a channel mechanism to allow data flow to cut across the coordination hierarchy (as illustrated in Figure 4). Channels provide flexibility for defining large processes with complex data sharing protocols, while keeping the process definition clearer and easier to understand.

Transaction Management: We previously noted that many of the LIXI processes can be viewed as transactions, but that they are typically long transactions that often have subtransactions nested within them. Thus, for example, the valuation process can readily be viewed as a transaction, which terminates with a valuation being delivered and payment being provided to the valuer. Nested within that transaction are other subtransactions, one of which is modification of the amount of payment to the valuer. This subtransaction may be undertaken when the valuer is unhappy with the amount of payment. The subtransaction must terminate before the valuation transaction does. If the subtransaction ends without an agreement, then the valuation transactions containing it must abort. We have encountered a number of other examples of transactions, many of them nested, in the LIXI processes.

Neither BPMN nor Little-JIL has adequate facilities for specifying transactions. BPMN offers facilities for a small number of fixed responses (such as execution of a rollback exception handler) to contingencies, but lacks important dimensions of flexibility. Little-JIL offers more facilities that seem more useful as they provide for greater flexibility. Little-JIL's exception management facilities, for example, offer a way to provide for a wider variety of approaches to compensation and rollback that could be used to provide features of many kinds of transactions. In addition, Little-JIL's substep semantics, combined with exception management can be used to synthesize specific nested transaction capabilities. We note, however, that even in

Little-JIL these features must be specifically created. It seems far preferable to have transaction semantics incorporated as an integral well-defined part of a process definition language. Such semantics would be of considerable value in defining LIXI processes such as Valuation.

Variation. As noted earlier, our experience suggests that there are often situations in which different performers have different views of the same process, and indeed may wish to carry out the process differently. In the case of the valuation example, we have shown one specific way of performing this process, but acknowledge that there are indeed other ways in which it could be done. To some extent these differences can be concealed as elaborations of activities that are simply not shown. In other cases, however, such variations as differences in ordering of activities, ways of handling exceptions, and strategies for substituting for unavailable performers cannot (and indeed should not) be concealed. For these reasons it seems useful for a process notation to incorporate mechanisms that support the specification of process variation.

Little-JIL provides several mechanisms to accommodate design-time variations since static enumeration of the different variants is not only impractical but it also results in cumbersome processes that are very difficult to analyze. As previously mentioned, a Little-JIL process definition consists of a coordination specification, an artifact collection, and a resource repository. Since there is a clear separation of concerns, it is easy to achieve variation by making changes to each of these three elements, for example by making changes in agent behavior, artifact structure, and task elaborations [17][25].

Since every step in Little-JIL is assigned to an agent who will execute it, alternatives in the specification of the agent, or in the description of the agent's behavior will change the process thereby creating process variation. Likewise, the artifact collection is an integral part of a Little-JIL process definition and changing the structure of an artifact, also creates process variation. Artifacts can be modified by adding or removing different fields according to the context. For example, one valuation process may wish to accord anonymity to the valuer, and another may not. This type of variation can be achieved in Little-JIL simply by either excluding or including a field in the valuation report that contains the name of the valuer.

Both BPMN and Little-JIL can accommodate the desire for variation in the ways in which the lowest level activities in a process decomposition are defined. In BPMN the bottom subprocesses are to be performed in ways that are not specified, presumably leaving the performers latitude to create their own process variants. Similarly, leaf steps in Little-JIL are not decomposed further so when an agent is assigned to execute a leaf step, there are no restrictions and the agent has full freedom in deciding on how to execute the step. In both cases, creating variants by task elaboration is achieved by interchanging an elaborated subprocess for a bottom level subprocess.

Observation and Monitoring: In both BPMN and Little-JIL it is possible to observe the progress of process execution and thereby monitor the process. In either case this can be done by embedding in the process specific activities that report on status and

progress. Little-JIL has specific language features designed to facilitate this. In particular, each step may have a specified pre-requisite and post-requisite. Requisites are designed specifically to monitor for adherence to, or violation of, stated conditions. Thus they seem to strongly facilitate process execution monitoring. It is expected that requisites will embody executable checks for adherence to desired conditions, either those required for safe execution of a step, or those required as evidence of adequate performance of the step. Violation of a requisite activates another Little-JIL language feature designed to facilitate monitoring, namely the invocation of an exception. Thus requisite violation will cause the immediate performance of an exception handler that might send a message, or create some other auditable record that can provide a full description of the event. All of this remains possible in BPMN, but requires the use of lower level language features that make this more difficult and harder to read.

Timing: We note that every process has timing constraints of some sort. In the LIXI environment, these constraints can be severe, some of them reflecting legislative mandates for promptness and responsiveness. Neither BPMN nor Little-JIL provides adequate facilities for dealing with timing constraints in processes. Little-JIL provides the ability to state a deadline for the performance of a step. This seems useful for assuring that the lowest level activities are not allowed to languish. In case the deadline time expires, a timer exception is thrown which enables the process to proceed forward through the execution of the appropriate exception handler. Unfortunately, however, Little-JIL does not incorporate any facilities for interrupting the agent that is executing the step whose time limit has expired, nor for retrieving any resources that had been allocated to that step. Thus, Little-JIL offers some useful capabilities, but far more powerful and flexible timing specification and control features are needed.

Scaling to handle Size and Complexity: The brief example included in this paper is not a good vehicle for indicating the need for scaling, or effective approaches for supporting it. Valuation is only one small process within the far larger and more complex process for seeking, approving, extending, and then servicing a mortgage loan. Our work has explored many of other subprocesses of this large process, and this work has made it clear that these processes become very large very quickly, and that their size can be a significant obstacle to human comprehension. Thus mechanisms for supporting the scalability of process definitions are needed.

Earlier we have indicated that Little-JIL incorporates mechanisms for implementing abstraction and that they seem effective in supporting scalability. Certainly experience with programming language design strongly indicates that languages offering strong support for abstraction tend to also support scalability correspondingly strongly. Our early experiences in reuse of process steps in Little-JIL suggests that these lessons from programming languages are also applicable in the domain of process definition languages.

Clarity: BPMN uses a flowchart representation to depict processes. Flowcharts and much of the other BPMN notation are very intuitive to business users. The language

also incorporates a variety of constructs for different types of events, gateways, and activities, as well as different connectors for control and data flow. This results in pictorial depictions of processes that users find quite comfortable and easy to understand.

Little-JIL uses a pictorial notation that bears little resemblance to the more usual flowgraph, finite-state machine, or Petri Net notations. It uses a very small set of constructs, which are combined in different ways to define process steps and compose them into higher levels of process abstraction, thereby offering convenient ways to deal with higher levels of process complexity. The expressiveness of Little-JIL does not seem to us to be compromised by its smaller set of fundamental constructs. Moreover, this feature makes Little-JIL easy to learn so that a beginner developer would not be at a disadvantage because he or she is not familiar with all the constructs the language may offer. In addition, our observation has been that newcomers to this notation find it relatively easy to learn so that they are able to participate actively in reviews of Little-JIL process definitions usually within the first hour of their exposure to it.

Rigor: We have noted that a top-level goal of our project is to produce process definitions that help the Australian loan industry perform its work "faster, better, and cheaper". Our view is that this entails the use of process definitions as devices for the direct support of iterative incremental (loan industry) process improvement. The canonical process improvement loop begins with creation of a process, then continues with evaluation of the process aimed at identifying defects, efficiency obstacles, etc., then proceeds to removal of such defects and obstacles from the process, and reevaluation to assure that the removal has succeeded while nevertheless not introducing subsequent defects.

This summary of the process improvement process makes it clear that analysis of process representations is a central activity. This, in turn, emphasizes the importance of having process definitions that are defined precisely, because definitive analysis is possible only when the meaning of a process representation is precise and unambiguous.

We have found that BPMN is not sufficiently precisely defined, as it incorporates a number of constructs whose interpretation is ambiguous. As a consequence BPMN process definitions cannot be subjected to definitive analysis, and are correspondingly less useful in the sort of process improvement process that we have outlined.

Little-JIL, on the other hand, is a rigorously defined language. The central language abstraction, the step, is defined in terms of finite state automata. The automata define precisely the way in which a step sequences such activities as binding of arguments, execution of requisites, handling of exceptions, and sequencing of substeps. Accordingly a Little-JIL process definition can be used to create a variety of structures whose analysis can then cast direct light upon the presence or absence of defects from the Little-JIL process definition. For example, a Little-JIL process can be translated into the form of a (very large and complex) flowgraph that can then be

subjected to finite-state verification aimed at the detection of event sequence errors [7]. A Little-JIL process definition can also be used to automatically generate a fault-tree that can be used as the basis for Fault Tree Analysis (FTA) or Failure Mode Effects Analysis (FMEA) [6].

4 Discussion and Conclusions

Our brief example has illustrated a number of challenging desiderata for process definition languages. We summarize them very briefly in this section.

One such desideratum is the need to separate the specification of how an activity is to be performed from the specification of the agent who is to perform it. In particular, our example has shown the value of making clear that a specified capability is what is needed by a particular step, but leaving the identification of the agent to be accomplished by late binding at runtime. Thus we see that Little-JIL's capabilities for **Dealing with Parties and Agents** are a major improvement over the more static capabilities in BPMN. The example also indicates the complex **Exception Management** needs of LIXI processes, and likewise demonstrates that powerful exception management capabilities such as Little-JIL's are needed here. The example only begins to show the value of incorporating abstraction into a process language, indicating the value of thinking of an activity (represented in Little-JIL as a step) as a procedure. In this example we see the value of parameter passing and the use of scoping, and demonstrate that these are achieved clearly and naturally when an elaboration of an activity is viewed as a real abstraction, rather than simply as an elaboration that adds details. All of these features add greatly to the ability of Little-JIL to represent important forms of process **Variation** that are lacking in BPMN, and in most other process languages that we have investigated. These same features also seem to make the **Reconciliation of Process and Data Flow** views of our processes clearer. Thus we see how some of the key semantic features incorporated into Little-JIL seems effective in addressing some of the major areas of need identified early in this paper.

Indeed, our experience suggests that there is further value to be derived from Little-JIL language features. We note in particular that the use of abstraction in Little-JIL fosters reuse, clarity, and terseness of expression of complex process content. We have also found that the ability to specify concurrency is extremely important, as are other Little-JIL language semantic capabilities that we do not elaborate upon here due to the lack of space in this paper.

On the other hand, it is important to emphasize that there are other process definition features that are still not sufficiently provided by Little-JIL or other process languages that we have examined. In particular, we note that there seems to be benefit in thinking of many LIXI processes as transactions that may be nested in complex ways. It would be a great benefit to have a process definition language in which such processes can be defined as transactions, with the appropriate transaction semantics

provided by the language. Little-JIL can be thought of as providing a toolbox for constructing such transactions, but this is not nearly as satisfactory as having the semantics provided (and assured) by the language itself.

In addition, we note that few process definition language offer significant features for supporting the specification and enforcement of timing constraints. This is particularly surprising in view of the fact that virtually all processes have timing constraints. Such constraints are particularly important in LIXI processes, and we have been surprised that these needs are not well met by either BPMN or Little-JIL.

We will continue to use BPMN models in some aspects of our LIXI process definition work, mainly because of its status as an international standard and because of its abundant tooling support in graphical modeling. On the other hand, in order to meet all the needs that a process definition has to support, and to address unique challenges in the highly dynamic and variable nature of process definitions in domains such as LIXI (i.e. industry-wide reference processes or highly adaptable processes), a more sophisticated process definition language is needed. Our initial experience with Little-JIL has met nearly all of these needs. Further work will be done to fully explore the advanced features of Little-JIL.

In addition, we note that preliminary work indicates that Little-JIL's strong semantic basis renders processes defined in the language amenable to analysis by powerful finite state verification tools, as has been indicated in the previous section. The use of such tools seems to us to offer the clearest path towards effective use of process definitions to support making LIXI processes "faster, better, and cheaper". Thus we will also explore leveraging the use of these analysis tools by applying them to Little-JIL definitions of LIXI processes.

5 Related Work

Despite possible perceptions to the contrary, we must note here that BPMN actually seems relatively expressive in comparison to other workflow languages. Indeed a systematic comparison shows that traditional block-based workflow languages, such as UML activity diagrams[15], BPEL [14], and XML Process Definition Language (XPDL)[27], are more or less similar to BPMN. Indeed, because of the non-executable and informal nature of BPMN, it is often considered to be more expressive than these other notations. But, as the foregoing has indicated, BPMN still lacks key expressive capabilities that are needed in order to specify LIXI processes. This suggests the need for new languages capable of addressing increasing numbers of the demanding needs just summarized.

Indeed there is no shortage of other workflow languages. Among those that have been proposed are those by [3, 9, 11, 20, 24]. One very recent effort in this area is YAWL[2] and NewYAWL[22]. NewYAWL incorporates more resource and exception handling capabilities, and incorporates a number of other important features

that are missing from most workflow languages. In addition E-services frameworks BPEL 1.1[14] [13] [20] have also been used as the basis for defining processes centered on marshalling internet-accessible resources to do work or produce products in pursuit of goals that are also quite similar to those of our project. This should not be surprising as the real estate loan industry is indeed an s-services industry.

We note that the languages used in these domains bear some striking similarities to software process languages. In particular, a great many of these languages are based upon flowgraph representations, most are effective in defining artifact flows, and most make some provision for hierarchical decomposition. Studying these similarities seems to yield interesting and important insights into the nature of processes, and basic process language desiderata. The weaknesses of these languages are no less important and illuminating, however. Thus, we note that most of these languages lack effective mechanisms for representing exceptions and their handling, for defining resources and their utilization, and for representing real-time behaviors and constraints. Ultimately, consideration of the needs for such features should lead to languages that should be more effective in supporting pursuit of the goals in these domains [17-19].

That pursuit has led us to consider process definition languages, many of which have emerged from work on the rigorous definition of the processes by which software is developed. There is a long history of work in this area. and we note that many languages and diagrammatic notations have been evaluated as vehicles for defining software processes. It was suggested that processes be defined using a procedural language [26]. In MARVEL/Oz [5]processes were defined using rules. SLANG [4] used modified Petri Nets to define processes.

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. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML), 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. Other approaches involve the creation and analysis of Fault Trees [6, 7]to study the ripple effects of failures to perform activities correctly, and the use of discrete event simulations as the basis for evaluating possible changes in resource allocation.

6 Acknowledgements

NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

This research was also supported by the US 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 U.S. National Science Foundation, or the U.S. Government.

7 References

[1] "Lending Industry XML Initiative (LIXI)," http://www.lixi.org.au
[2] W. M. P. v. d. Aalst and A. H. M. t. Hofstede, "YAWL: Yet Another Workflow Language," Information Systems, vol. 30(4), pp. 245-275, 2005.
[3] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, A. E. Abbadi, and C. Mohan, "Exotica/FMDC: A Workflow Management System for Mobile and Disconnected Clients," Distributed and Parallel Databases, vol. 4(3), pp. 229-247, 1996.
[4] S. Bandinelli, A. Fuggetta, and C. Ghezzi, "Model Evolution in the SPADE Environment," IEEE Transaction on Software Engineering, vol. 19(12), pp. 1128-1144 1993.
[5] I. Z. Ben-Shaul and G. E. Kaiser, "A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment," in 16th International Conference on Software Engineering, 1994 pp. 179-188.
[6] B. Chen, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Automatic Fault Tree Derivation from Little-JIL Process Definitions," in SPW/PROSIM Shanghai, China, 2006,Springer-Verlag, LNCS 3966, pp. 150-158.
[7] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "Verifying Properties of Process Definitions," in ACM SIGSOFT International Symposium on Software Testing and Analysis, Portland, 2000,ACM Press, pp. 96-101.
[8] H.-E. Eriksson and M. Penker, Business modeling with UML : Business Patterns at Work. New York: John Wiley & Sons, 2000.
[9] L. Fischer, "Workflow handbook 2006," 2006.
[10] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," Distributed and Parallel Database, (3), pp. 119-153, 1995.
[11] C. Hagen and G. Alonso, "Exception Handling in Workflow Management Systems," IEEE Transaction on Software Engineering, vol. 26(10), pp. 943-958, 2000.
[12] M. Laguna and J. Marklund, Business process modeling, simulation, and design. Upper Saddle River, NJ: Pearson/Prentice Hall, 2004.
[13] A. Lazcano and G. Alonso, "Process Based E-services," in Second International Workshop Electronic Commerce(WELCOM), 2001 pp. 1-10.
[14] OASIS, "Business Process Execution Language (BPEL) 1.1," http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
[15] OMG, "Unified Modelling Language 2.1.1," http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML
[16] OMG, "Business Process Modeling Notation Specification (version 1.0 Final Adopted Version)," 2006.
[17] L. Osterweil, "Software Processes Are Software, Too, Revisited," in 19th International Conference on Software Engineering (ICSE), Boston, MA, 1997 pp. 540-558.

[18] L. Osterweil, N. K. Sondheimer, L. A. Clarke, E. Katsh, and D. Rainey, "Using Process Definitions to Facilitate the Specifications of Requirements. ," Department of Computer Science, University of Massachusetts, Amherst, MA 2006.
[19] C. Rolland, "A comprehensive View of Process Engineering," in 10th International Conference on Advanced Information Systems Engineering (CAiSE'98), 1998.
[20] M. Rusinkiewicz and D. Georgakopoulos, "From Coordination of Workflow and Group Activities to Composition and Management of Virtual Enterprises," in International Symposium on Database Applications in Non-Traditional Environments (DANTE), 1999 pp. 3-15.
[21] N. Russell, W. M. P. v. d. Aalst, and A. H. M. t. Hofstede, "Exception Handling Patterns in Process-Aware Information Systems," BPM Center Report BPM-06-04, 2006.
[22] N. Russell, A. H. M. t. Hofstede, D. Edmond, and W. M. P. v. d. Aalst, "newYAWL: Achieving Comprehensive Patterns Support in Workflow for the Control-Flow, Data and Resource Perspectives," BPMcenter.org 2007.
[23] N. Russell, A. H. M. t. Hofstede, D. Edmond, and W.M.P. van der Aalst, "Workflow Resource Patterns," Eindhoven University of Technology BETA Working Paper Series, WP 127, 2004.
[24] A. P. Sheth, D. Georgakopoulos, S. Joosten, M. Rusinkiewicz, W. Scacchi, J. C. Wileden, and A. L. Wolf, "Report from the NSF Workshop on Workflow and Process Automation in Information Systems.," SIGMOD Record, vol. 25(4), pp. 55-67, 1996.
[25] B. I. Simidchieva, L. A. Clarke, and L. J. Osterweil, "Representing Process Variation with a Process Family," in International Conference on Software Process (ICSP), 2007.
[26] S. M. Sutton, D. Heimbigner, and L. J. Osterweil, "APPL/A: a language for software process programming," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 4(3), pp. 221-286, 1995.
[27] WfMC, " XML Process Definition Language (XPDL) 2.0," http://www.wfmc.org/standards/XPDL.htm
[28] A. Wise, "Little-JIL 1.5 Language Report," Department of Computer Science, University of Massachusetts, Amherst, MA 2006.