Flexible Static Semantic Checking Using First-Order Logic

Shimon Rura and Barbara Lerner

Williams College, Computer Science Department,
Williamstown, MA 01267 USA
{03sr, lerner}@cs.williams.edu

Abstract. Static analysis of software processes is important in assessing the correctness of processes, particularly since their long duration and distributed execution make them difficult to test. We describe a novel approach to building a static analyzer that can detect programming errors and anomalies in processes written in Little-JIL. We describe semantic rules declaratively in first-order logic and use xlinkit, a constraint checker, to check the processes. We have used this approach to develop a checker that can find simple syntactic errors as well as more complex control and data flow anomalies.

1 Introduction

Formal descriptions of software processes can help us understand, analyze, and execute complex processes. But the value of process descriptions depends on their correctness: an invalid process description is unusable, and an improperly specified process description can be misleading. To help process developers verify the validity of their processes and catch some common mistakes, we developed a static semantic checker which checks a process against a set of rules expressed in first-order logic. Our approach quickly yielded a tool that catches simple language errors. More surprising, however, is that this simple approach has scaled to more challenging static analysis problems and has the potential to provide a lightweight yet effective checking framework for many phases of process development and analysis.

The semantic analysis that we describe was built for Little-JIL [9], a process language being designed and developed by researchers in the Laboratory for Advanced Software Engineering Research at the University of Massachusetts, Amherst and at Williams College. It has a simple graphical syntax, which allows for an intuitive visual representation of process structure, and well-defined semantics designed to be rich enough to allow analysis and execution of complex processes. Static analysis is particularly valuable for process developers, because the prolonged and distributed nature of most processes can make testing, as well as more formal dynamic analyses, prohibitively expensive. To this end, we are pursuing several projects that enable static analysis of Little-JIL processes. In this paper, we describe a novel approach to static analysis. Using xlinkit [6], a COTS constraint checker, we have developed a checker that examines Little-JIL processes and detects a variety of anomalies, including language violations, race conditions, lost data, and infinite recursion. The semantics to be checked are expressed as a set of rules written in first-order logic. This approach has made it easy to turn assertions about program

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 143–153, 2003.
© Springer-Verlag Berlin Heidelberg 2003

structure into the parts of a working checker. The checker's architecture, which keeps semantic rule declarations separate from verification and reporting procedures, makes it easy to extend the checker with new anomaly detection rules and to adjust existing rules if language features change.

2 Overview

A Little-JIL process is represented as a hierarchy of steps, expressed in a simple graphical syntax. The Little-JIL interpreter uses this description to coordinate the flow of tasks and information between agents, which may be human or automated and interact with the interpreter via an API or GUI. Steps are connected with a number of different edges: substep edges, prerequisite and postrequisite edges, reaction and exception handling edges. The characteristics of steps and edges determine how data and control flow between agents. (The Little-JIL Language Report [8] describes the language in detail.)

A developer creates a process using a specialized Little-JIL editor by creating and manipulating steps, represented graphically, and links between steps, represented as lines. Each step has an icon (the *sequencing badge*) that describes the control flow for its substeps. For example, a step with the sequential badge completes when all of its substeps have completed, while a choice step completes when exactly one of its substeps is completed. (The choice of which substep to perform is deferred until runtime). A leaf step, which cannot have substeps, is dispatched to an external agent which notifies the Little-JIL interpreter via its API when the step is completed.

The process programmer provides the details of a step's interface and the data passed between steps by selecting a step or edge and editing its properties in a separate form. The visual representation of a process displays step names, sequencing badges, and the edges that connect steps to their substeps and exception handling steps, but suppresses additional information about step interfaces, prerequisites and postrequisites, and data flow between steps. Thus while the process diagram makes it easy to check that substeps are connected as intended, it is more cumbersome to verify other important properties, such as that all input parameters to a step are bound.

The editor enforces proper syntax and performs some static semantic checks. The editor generally prevents the introduction of constructs that would violate language semantics. For example, it will not allow the process programmer to create a parameter binding that results in a typing error. The editor's incremental checking is best suited for preventing the creation of inconsistent processes; it is not well suited to ensuring that a process description is complete. For example, while it guarantees that all parameter bindings are well-typed (a consistency check), it does not check whether all the input parameters of a step are bound to some value (a completeness check). Since Little-JIL is an interpreted language, there is no compiler to detect these errors. As a result, without an additional semantic checker, many violations of language semantics can go undetected until runtime.

In addition to linguistic errors such as these, a programmer might use the language in a way that is technically correct but suggests that the process is incomplete. For example, sequential and choice steps are expected to have substeps, but the language defines the behavior of these steps when they have no substeps. If a sequential step

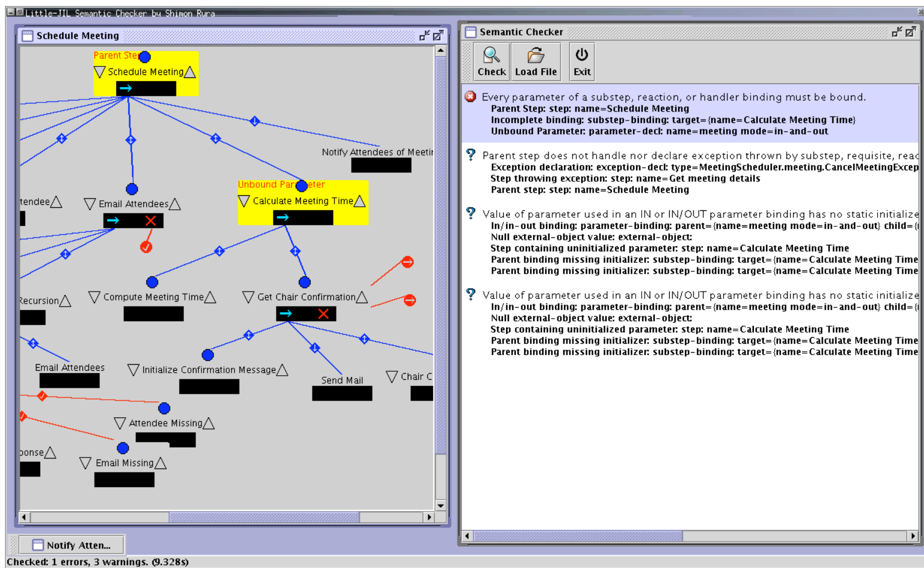

Fig. 1. Reporting errors in the Little-JIL Semantic Checker

with no substeps executes, it will complete successfully since all of its (zero) children complete successfully. If a choice step with no substeps executes, it will always fail because a choice step requires exactly one substep to complete successfully. The use of non-leaf sequencing badges when a step has no substeps suggests a programming error. This type of anomaly is not detected by the editor, but merits a warning before execution.

In considering how to address these issues of checking static semantics, we decided to use a novel approach that would allow us to express the language semantics declaratively rather than use Java to extend the editor with a semantic checker written in a more traditional style. The benefits of this approach stem from the more concise and encapsulated representation of the semantic rules. Rather than needing to extend multiple parts of the editor to support each semantic check, we can express each semantic rule purely in terms of program structure. This makes semantic checks easier to develop, understand, and maintain.

Figure 1 shows an example of the semantic checker in operation. The top panel displays the Little-JIL process as it appears in the editor, while the bottom panel displays the error and warning messages. When the user clicks on an error message, the steps of the process that are related to the error message are highlighted as shown.

In the remainder of this paper, we present more details on this mechanism of describing the semantic rules declaratively and our experiences with the semantic checker.

3 Encoding Rules in First-Order Logic

The Little-JIL static semantic checker makes use of xlinkit [6], a commercial constraint checking tool. Given one or more documents and a set of consistency rules expressed

```
<consistencyrule id="warnNonLeafNoChildren">
   <forall var="nonLeaf" in="//step[@kind != 'leaf']">
      <exists var="sub" in="$nonLeaf/substeps/substep-binding"/>
   </forall>
</consistencyrule>
```

Fig. 2. xlinkit Consistency Rule to Find Non-Leaf Steps with No Substeps

as first-order logic predicates relating document elements, xlinkit attempts to satisfy the rules across the documents. If a rule cannot be satisfied, xlinkit reports an inconsistency, identifying the document elements for which the rule cannot be satisfied.

To use xlinkit, we generate an XML representation of the Little-JIL process and define our semantic rules using the consistency rule schema provided by xlinkit. The consistency rule schema allows us to express first-order logic expressions using XML syntax. Within the rules, XPath [11] expressions are used to select sets of elements or values from the document. In our case, the document elements correspond to Little-JIL language elements, and the XML document structure mirrors the process structure. Thus the encoding of many conditions is quite straightforward. To report errors to the user, we map the XML document elements involved in an inconsistency back into the steps of the process so that they can be highlighted as we display the corresponding textual error message as shown in Figure 1.

For example, the logical rule that asserts that each step with a non-leaf sequencing badge should have one or more substeps can be expressed in first-order logic as:

$$\text{let } NonLeaves = \{s|s \in Steps \wedge kind(s) \neq Leaf\} \text{ in}$$
$$\forall\, nonLeaf \in NonLeaves, |substeps\,(nonLeaf)| > 0$$

In this rule $Steps$ denotes a set containing all the steps in the process, the function $kind$ reports the kind of sequencing badge the step has, and the function $substeps$ reports the set of substeps that a step has.

To encode this rule in xlinkit's consistency rule schema, we must translate the first-order logic syntax into XML[10]. Within the rules, we use XPath[11] expressions to denote the set of non-leaf steps and the sets of substeps of these non-leaves. The XML encoding of this rule is shown in Figure 2. `//step[@kind != 'leaf']` is an XPath expression that denotes the set of all steps not designated as leaves and `$nonLeaf/substeps/substep-binding` is the set of all substep edges coming from a step $nonLeaf$. To verify this rule, xlinkit will attempt to associate each non-leaf step with a substep binding. These associations are called *links* and when a link cannot be completed, xlinkit reports it as inconsistent. In this case, xlinkit will report an inconsistency for each non-leaf step that has no substep bindings. For each inconsistency, it will identify the non-leaf that violates the consistency rule.

Figure 3 shows the checker output when run on a process where one non-leaf step (the root) has substeps, while another (named `Bad NonLeaf`) does not. The error panel provides a textual message and identifies the step violating the rule by name. The left

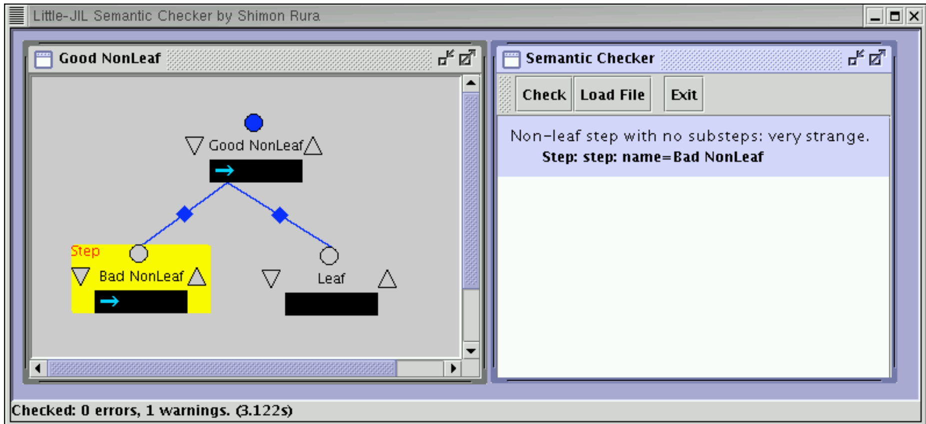

Fig. 3. Violation of the Rule that Non-Leaves must have substeps

```
<?xml version="1.0"?>
<!DOCTYPE program
  SYSTEM "http://www.cs.williams.edu/~lerner/littlejil.dtd">
<program root="Diagram1">
  <diagram root="Step1" id="Diagram1">

    <step name="Good NonLeaf" kind="sequential" id="Step1">
      <substeps>
        <substep-binding target="Step2"></substep-binding>
        <substep-binding target="Step3"></substep-binding>
      </substeps>
    </step>

    <step name="Bad NonLeaf" kind="sequential" id="Step2">
    </step>

    <step name="Leaf" kind="leaf" id="Step3">
    </step>

  </diagram>
</program>
```

Fig. 4. XML Representation of a Simple Little-JIL Process

panel highlights the step visually. The XML that corresponds to this process is shown in Figure 4.

When the rule is applied to the XML representation of the process, there are two steps that satisfy the forall clause: the root and the step named `Bad NonLeaf`. The step named `Leaf` does not satisfy the forall clause because its $kind$ attribute has the value $leaf$. When evaluating the root, the XML definition shows that it contains two substep-bindings, thereby satisfying the rule. Looking at the XML for `Bad NonLeaf`, however, we see that it fails to satisfy the rule and is therefore flagged as being inconsistent.

The rule shown above is among the simplest rules. The semantic checker currently supports the following rules:

Language semantic rules:

- Prerequisites and postrequisites should not have out parameters. (They should behave as pure boolean functions.)
- All input parameters to a step should be initialized.
- A step should handle or explicitly throw all exceptions its substeps can throw.
- The root step should have an agent.
- The agent of a non-root step should either be acquired locally or passed in from the parent.
- Deadlines should only be attached to leaves.

Programming anomalies:

- All steps with non-leaf sequencing badges should have substeps.
- All steps should be connected to the process graph.
- All steps should be reachable.
- A non-leaf step should only have handlers for exceptions that its substeps throw.
- A non-leaf step should only declare that it throws exceptions that its substeps throw.
- Multiple substeps of a parallel step should not modify the same parameter. (This is a possible race condition.)
- All parameters to a step should be bound.
- The values of all output parameters should be propagated to other steps.
- Recursion should terminate.
- All items in an interface should have unique names.

While it is not surprising that first-order logic can be used to check local properties on a step, it is more surprising that it can also be used to check for fairly complicated data flow and control flow anomalies. As an example of a more complicated rule, we present details on the rule that checks whether an output parameter is propagated to other steps in the process. First, we offer an explanation for why we have this rule. All real work is performed by the agents when they are assigned a leaf step. Part of what they may do is to produce a data result that can then be passed up to its parent and from there distributed to other parts of the process. Since a leaf step can appear in multiple contexts in a process, it is possible that it will have output parameters that are only needed in some contexts and can be ignored in other contexts. The proper thing to do when ignoring output data is to not provide a parameter binding to its parent for this output data. If a parameter binding is provided, then we would expect to see this data being propagated from the

parent to another step, such as another child, an exception handler, a postrequisite, or up to its own parent. The consistency rule that we describe next checks whether this propagation occurs.

$$
\begin{array}{l}
\text{let } NonLeaves = \{s | s \in Steps \wedge kind(s) \neq Leaf\} \text{ in} \\
\quad \forall\, nonLeaf \in NonLeaves, \\
\quad \forall\, substep \in substeps(nonLeaf), \\
\quad \forall\, outParamBinding \in outParams(substep) \\
\quad \text{let } paramName = nameInParent(outParamBinding) \text{ in} \\
\qquad ((kind(nonLeaf) == Sequential \vee kind(nonLeaf) == Try) \wedge \\
\qquad\quad \exists\, laterSubstep \in laterSubsteps(subStep), \\
\qquad\quad \exists\, inParamBinding \in inParams(laterSubstep), \\
\qquad\qquad paramName == nameInParent(inParamBinding)) \\
\qquad \vee \exists\, parentOutParamBinding \in outParams(nonLeaf), \\
\qquad\quad nameInChild(parentOutParamBinding) == paramName))
\end{array}
$$

In this rule, $outParams$ identifies the set of parameter bindings where a step outputs a value, while $inParams$ identifies the set of parameter bindings where a step inputs a value. $nameInParent$ is a function that returns the name of a parameter in a parent step involved in a parameter binding to a child, while $nameInChild$ returns the name of the parameter in the child step involved in a parameter binding to a parent. $laterSubsteps$ returns the set of substeps that are the sibling of a given step and follow that step. The rule first finds all parameter bindings where a child outputs a parameter value to a parent. It then checks that the parameter is used by checking that the parameter is input to a subsequent child or is itself output to its parent's parent.[1] Because of the flexibility of XPath and the structure of the XML representations of Little-JIL processes, each of these functions is easy to express. The XML representation of this rule, shown in Figure 5, is more lengthy but essentially expresses the same condition.

4 Checker Architecture

The checker is implemented in Java, as is xlinkit. The architecture of the checker is shown in Figure 6. The checker consists of three major components: the user interface, the LJILChecker component, and the LJILFetcher component. The user provides a filename containing a Little-JIL process to be checked via the user interface. This is passed through the LJILChecker to xlinkit which uses a custom xlinkit *fetcher* which automatically invokes a Little-JIL to XML translator when xlinkit is invoked on a Little-JIL file. Xlinkit performs the consistency checks using the rules stored in the rule file. Xlinkit returns a set of links to the LJILChecker identifying the document elements that violate consistency rules along with the rules violated. LJILChecker uses metadata from the violated rules to translate the links into LJILError objects. Each `LJILError` object contains a message and a set of `LJILErrorOperand` objects which carry titles and identifying information

[1] The complete rule also checks whether the parameter is passed to an exception handler, reaction, or postrequisite. These tests are similar but have been omitted for brevity.

```
<1-- For all non-leaves -->
<forall var="nonLeaf" in="//step[@kind != 'leaf']">

 <!-- For all out parameters of children of a non-leaf -->
  <forall var="outParamBinding"
    in="$nonLeaf/substeps/substep-binding/parameter-binding
      [@mode = 'copy-out' or @mode = 'copy-in-and-out']">

    <!-- Bind substep to the child involved in the parameter
         binding -->
    <forall var="substep" in="id($outParamBinding/../@target)">

    <or>
      <or>
        <and>
          <!-- The step has sequential or try control flow -->
          <or>
            <equal op1="$nonLeaf/@kind" op2="'sequential'"/>
            <equal op1="$nonLeaf/@kind" op2="'try'"/>
          </or>

          <!-- And the same parameter is passed as an in-parameter
               to a later substep -->
          <exists var="inParamBinding"
           in="$outParamBinding/../following-sibling::substep-binding
            /parameter-binding[
            @in-parent = $outParamBinding/@in-parent and
            @mode != 'copy-out']"/>
        </and>

        <!-- Or, no matter what step kind, the parameter is passed
             up to the parent of the non-leaf -->
        <exists var="parentOutParamBinding"
          in="//substep-binding[@target = $nonLeaf/@id]
          /parameter-binding[@in-child = $outParamBinding/@in-parent
          and @mode != 'copy-in']"/>
      </or>

      <!-- Conditions to check for passing to exception handlers,
           reactions, and postrequisite omitted -->
     </or>
  </forall>
  </forall>
</forall>
```

Fig. 5. Rule to Check for Propagation of Output Values

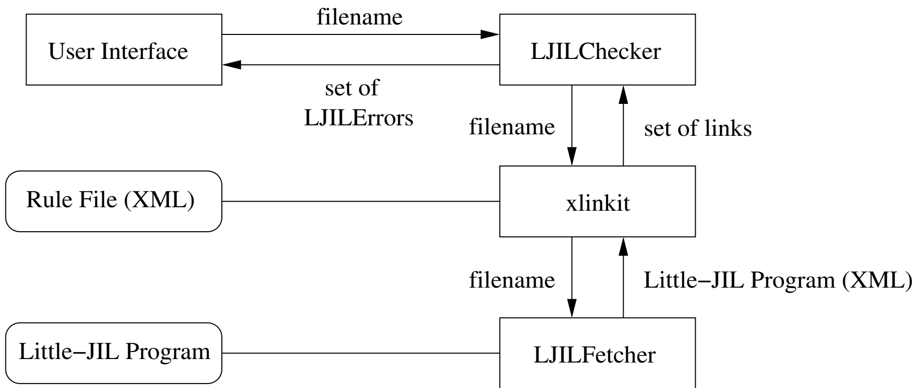

Fig. 6. Architecture of the Little-JIL Semantic Checker

for program elements referenced in the error message. The checking can be invoked either from a command-line printing interface, or from a GUI which uses `LJILErrorOperand` information to locate and visually highlight components of error messages (Figure 1).

The checker is designed to be easy to extend whenever new rules are formalized, whether due to language changes or improving knowledge of programming practice. So that rules can be added or changed by editing only the rule file, the error messages are specified with metadata contained in each rule. For example, in the rule in Figure 2 for detecting non-leaf steps with no substeps, we use the following `header` element:

```
<header>
  <description>
    Non-leaf step with no substeps: very strange.
  </description>

  <meta:msg mode="warning"/>
  <meta:operand seq="1" title="Step"/>
</header>
```

The error or warning message itself is provided in the header's description element. For further information, we take advantage of xlinkit's metadata facility, which allows free-form content within elements in the `meta` namespace. Thus the `meta:msg` tag describes a `mode`, either *warning* or *error*, indicating whether violations of the rule should be flagged as warnings or errors.

For each rule violation, xlinkit returns a *link*, a list of what document elements were bound to quantifiers in the rule before it was violated. In Rule 2, a consistent link would match a `step` element with a `substep-binding` element; an inconsistent link would contain only a `step` and result in a warning message. For useful reporting of these error elements, the `meta:operand` tag associates a title with an operand sequence number. Here, a warning message will give the title "Step" to its first operand as shown in the right panel of Figure 3. Though a title seems superfluous in this case, titles can be crucial to interpreting messages that refer to multiple language elements.

5 Future Work

Xlinkit was created to check the consistency of information encoded in several related XML documents rather than just within a single document as we use it here. A complete Little-JIL process consists of the coordination process discussed in this paper as well as several external entities: a resource model describing the resources available in the environment in which the process is executed and agents that carry out the activities at the leaf steps. Xlinkit offers a good foundation for building a tool to check whether a resource model and a collection of agents can provide the services required by the process. A tool like this is critical in determining whether a process will be able to succeed prior to its execution.

Another avenue to explore is whether xlinkit could be used to verify process-specific properties in addition to language semantics and general coding guidelines. Ideally, we would want to allow people writing these specifications to use a more convenient syntax than XML and to be able to write the specifications without needing to know the XML structure of Little-JIL processes. This therefore requires development of a specification language that can be translated automatically to the syntax required by xlinkit.

We are also exploring other avenues of static analysis of Little-JIL processes. Jamieson Cobleigh did some initial evaluation of applying the FLAVERS data flow analysis tool [3] to Little-JIL processes [1]. His approach required manual translation of Little-JIL into a model that FLAVERS could analyze. We are currently investigating automating this translation. We are also pursuing work toward the use of LTSA [5] to perform model checking of Little-JIL processes, and have had some encouraging initial results in this effort.

6 Related Work

The novelty of this work lies in its use of first-order logic to define the semantic analyzer. Formal notations are commonly used to define the semantics of programming languages. In particular, operational semantics and denotational semantics allow for fairly natural mappings to interpreters, typically written in functional programming languages. It is less clear how to derive static semantic checkers or coding style conformance tools from semantics written in these notations.

Verification tools, such as model checkers, data flow analyzers and theorem provers, often make use of formal notations to specify desired properties to be proven. Unlike our checker, however, these tools usually operate on a specialized semantic model derived from a program rather than the program's syntax itself. The development of these models may be very difficult, depending on the language to be checked.

LOOP [7] is a tool that translates Java code, annotated with formal specifications, into semantics in higher-order logic for analysis with a theorem prover. ESC/Java [4], which also uses theorem proving technology, is based on axiomatic semantics. Bandera [2] is a front-end for static analysis tools that builds models from Java source code with guidance from a software analyst. These tools focus on proving desirable runtime properties, rather than enforcing language semantics and coding guidelines.

Acknowledgements. Systemwire, the makers of xlinkit, provided us with excellent software that made this project possible. Christian Nentwich provided prompt and helpful technical support without which we might still be wading through the XML jungle.

The Little-JIL team at the University of Massachusetts, Amherst, particularly Aaron Cass and Sandy Wise, were helpful in discussing and suggesting rules to check. Sandy Wise also provided the code to produce XML representations of Little-JIL processes and to display Little-JIL processes in a form suitable for annotation with error messages.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9988254. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. Verifying properties of process definitions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 96–101, Portland, Oregon, August 2000.
2. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
3. Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 62–75, December 1994.
4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, Berlin, June 2002.
5. Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.
6. C. Nentwich, W. Emmerich, and A. Finkelstein. Static consistency checking for distributed specifications. In *International Conference on Automated Software Engineering (ASE)*, Coronado Bay, CA, 2001.
7. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proceedings of the 7th International. Conference On Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001.
8. Alexander Wise. Little-JIL 1.0 language report. Technical Report TR 98-24, University of Massachusetts, Department of Computer Science, 1998. Available at ftp://ftp.cs.umass.edu/pub/techrept/techreport/1998/UM-CS-1998-024.ps.
9. Alexander Wise, Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton Jr. Using Little-JIL to coordinate agents in software engineering. In *Proceedings of the Automated Software Engineering Conference (ASE 2000)*, pages 155–164, Grenoble, France, September 2000.
10. World Wide Web Consortium. Extensible markup language (XML). http://www.w3.org/XML/.
11. World Wide Web Consortium. XML path language (XPath). W3C Recommendation, November 16, 1999. Version 1.0. http://www.w3.org/TR/xpath.