Flexible Static Semantic Checking Using First-Order Logic

Shimon Rura and Barbara Lerner
Williams College

January 10, 2003

1 Introduction

Many errors in programs can be detected statically with the right tools. Compilers usually report some basic errors, and a variety of analysis tools can reason about programs based on static information such as data flow graphs. For many interpreted or experimental languages, however, these tools are lacking. Using traditional techniques for developing analysis tools, the cost of developing good tools may not be deemed worthwhile for languages still under development or languages with a small user base.

This poster presents a static semantic checker implemented using xlinkit [2], a COTS constraint checker, and a custom set of semantic rules expressed in first-order logic. This approach has made it easy to turn assertions about program structure into the parts of a working checker. Our checker is for Little-JIL, a maturing graphical process programming language. The checker features visual error reporting and can effectively detect and report over twenty conditions including linguistic restrictions, potential race conditions, lost data, and infinite recursion. 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

Little-JIL is a process programming language. A process is represented as a hierarchy of steps, expressed in a simple graphical syntax using a special editor. 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, and exception handling edges. The characteristics of steps and edges determine how data and control flows among agents.

As a process is developed, the editor enforces proper syntax and performs some static semantic checks. The editor tries to prevent the introduction of semantic errors: for example, it will not allow the process programmer to create an assignment 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 verifying properties of programs once they are thought to be complete and ready for execution. 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).

The semantic checker described here checks a Little-JIL process against rules, such as this one, that are required to hold in a complete program. In addition, it checks for conformance to coding guidelines and indicates warnings in these cases. The Little-JIL semantic checker provides clear and valuable information ranging from detection of common errors to complicated data flow anomalies.

Figure 1 shows an example of the semantic checker in operation. The left panel displays the Little-JIL process, while the right 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.

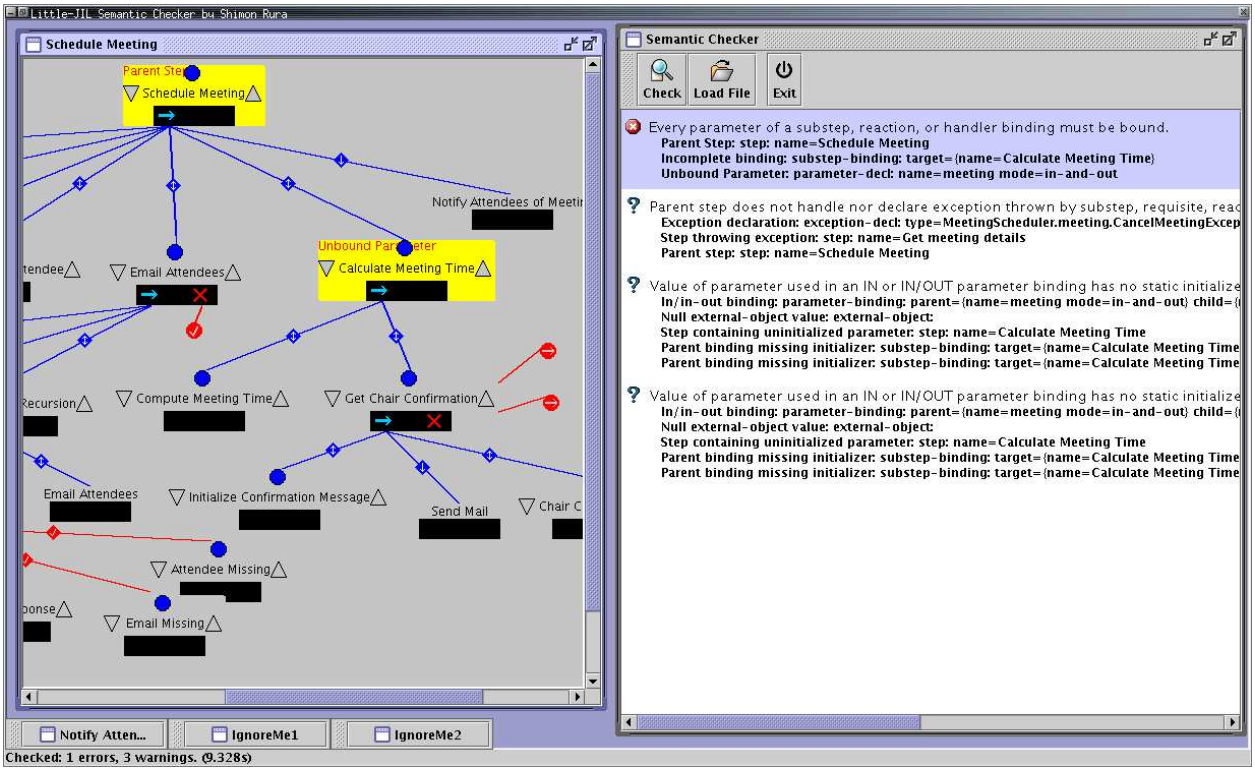

Figure 1: Reporting errors in the Little-JIL Semantic Checker

3 Encoding Rules in First-Order Logic

The Little-JIL static semantic checker makes use of xlinkit [2], a commercial constraint checking tool. Given one or more documents and a set of consistency rules expressed as first-order logic predicates relating document elements, xlinkit attempts to satisfy the rules across the documents.

The documents and ruleset are encoded in XML [4]. Within the rules, XPath [5] 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.

For example, each step has a designator that describes how control flows among its substeps in order to complete that step: sequentially from left to right, in parallel, etc. Steps with no substeps have a special *leaf* designator; their execution depends on an external agent. If a step is not designated as a *leaf* but has no substeps, it will exhibit strange and useless behavior such as instant success or failure. To warn programmers of this condition, we make use of the following logical assertion. Using XPath-like notation, //`step`[@`kind` $\neq$ "leaf"] denotes the set of all steps not designated as leaves and s/`substeps`/`substep`—`binding` is the set of all substep edges from a step s:

$$\forall\ s \in \text{//}\texttt{step}[\texttt{@kind} \neq \text{“leaf”}] \quad \exists\ sub \in s/\texttt{substeps}/\texttt{substep—binding}$$

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 actual xlinkit syntax, we write this rule as:

```
<consistencyrule id="warnNonLeafNoChildren">
   <forall var="s" in="//step[@kind != 'leaf']">
```

```
      <exists var="sub" in="$s/substeps/substep-binding"/>
   </forall>
</consistencyrule>
```

4 Checker Architecture

The checker is implemented in Java, as is xlinkit. Because Little-JIL programs are not normally stored as XML, we make use of a custom xlinkit *fetcher* which automatically invokes a Little-JIL to XML translator when xlinkit is invoked on a Little-JIL file. All core checking functionality is encapsulated in the `LJILChecker` class. Given a filename, the `LJILChecker` uses xlinkit to check that file. Using the links returned by xlinkit, along with metadata specified in the rules, it returns a set of `LJILError` objects corresponding to specific error or warning conditions. Each `LJILError` object contains a message and a set of `LJILErrorOperand` objects which carry titles and identifying information 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).

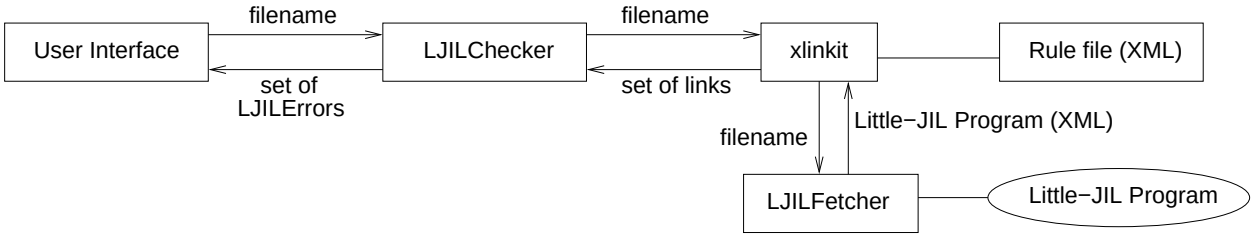

Figure 2: Architecture of the Little-JIL Semantic Checker

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. Rules can be added or changed by editing only the rule file; this is made possible by the use of metadata contained in each rule that controls how it will be reported. For example, in the rule 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 errors or warnings.

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 our example rule, 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 a sequence number. Here, a warning message will give the title "Step" to its first operand. Though a title seems superfluous in this case, titles can be crucial to interpreting messages that refer to multiple language elements.

5 Related Work

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 [3] 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 [1], which also uses theorem proving technology, is based on axiomatic semantics. These tools focus on proving desirable runtime properties, rather than enforcing language semantics and coding guidelines.

6 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] 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.

[2] 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.

[3] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proc. TACAS*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001.

[4] World Wide Web Consortium. Extensible markup language (XML). http://www.w3.org/XML/.

[5] World Wide Web Consortium. XML path language (XPath). W3C Recommendation, November 16, 1999. Version 1.0.