

Lightweight Analysis of Operational Specifications Using Inference Graphs

Laura K. Dillon, R. E. Kurt Stirewalt
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI, USA
+1 517 353 3148

ldillon@cse.msu.edu, stire@cse.msu.edu

Abstract

The Amalia framework generates lightweight components that automate the analysis of operational specifications and designs [16]. A key concept is the step analyzer, which enables Amalia to automatically tailor high-level analyses, such as behavior simulation and model checking, to different specification languages and representations. A step analyzer uses a new abstraction, called an inference graph, for the analysis. It creates and evaluates an inference graph on-the-fly during a top-down traversal of a specification to deduce the specification's local behaviors (called steps). The nodes of an inference graph directly reify the rules in an operational semantics, enabling Amalia to automatically generate a step analyzer from an operational description of a notation's semantics. Inference graphs are a clean abstraction that can be formally defined. The paper provides a detailed, but informal, introduction to inference graphs. It uses example specifications written in LOTOS for purposes of illustration.

Keywords Testing, analysis, and verification; Patterns and frameworks; Formal methods; Lightweight analysis components; Operational specifications; Automated software generators.

1. Introduction

Automated software-engineering environments (ASEs) create and manipulate representations of specifications, designs, and programs (hereafter called *system descriptions*). We recently developed a software generator called Amalia¹,

¹Named after Amalia Freud, the mother of Sigmund Freud, an allusion to our framework being a “generator” of “analyzers”.

which generates software that analyzes the behavior of system descriptions. Amalia-generated analyzers are packaged as *lightweight components*, so called because they analyze a system description *in memory*, without first translating the description into another representation.² In a companion paper, we discuss the benefits of lightweight analysis components over stand-alone analysis tools [16]. We also show that analysis algorithms can be made lightweight by implementing the analysis procedure using the visitor pattern [10] from object-oriented design. A key component in Amalia is a tool called the LWA generator, which generates visitor objects (called *step analyzers*) that compute local behaviors (called *steps*) by traversing the in-memory representation of a system description. Our use of visitors in Amalia represents a tradeoff of simplicity to achieve efficiency. The current paper explores this tradeoff and presents a theoretical framework we developed to deal with the resulting complexity.

Efficient step analyzers are difficult to generate automatically. We assume that a system description is formalized using *structural operational semantics* [15], according to which a step is computed by a formal derivation based on the syntactic structure of the system description. These semantics comprise a set of axioms and inference rules, which form the basis for constructing the formal derivations. A generator must translate these axioms and rules into a step analyzer. We require the translation to be a *reification*, which is to say that there must be a coherent implementation entity associated with each axiom and each inference rule. In the CENTAUR environment [5], rules are reified into Horn clauses, and a Prolog engine is employed to perform derivations. This solution exploits a powerful mecha-

²This meaning of “light-weight” is not standard. A tool may also be called light-weight if it operates on small and easy to write specifications.

nism, unification in Prolog, to simplify generation and synthesis, but at an efficiency cost. In Amalia, the efficiency constraints are tighter than in CENTAUR: We must reify rules into an efficient synthesis framework, and we must use a visitor object to orchestrate the synthesis.

We devised a synthesis framework that uses ideas from data-flow analysis frameworks [1, Ch. 10]. An *inference graph* is a data-flow mechanism that is constructed from a system description and that, when evaluated, computes the set of steps for that description. In this framework, nodes that initiate and transform steps reify axioms and rules, respectively. These nodes are linked together to form a flow graph, and the graph itself is constructed and evaluated by a visitor object as a side effect of visiting a system description. Our solution views the synthesis of steps as a data-flow problem, and the axiom/rule reifications as transfer functions.

Inference graphs enable the generation of correct and efficient step analyzers. We validated the generation claim by building a generator, which we used to construct step analyzers for two notations, pure LOTOS and linear-time temporal logic. To establish correctness, we formalized the flow framework and the reifications of axioms and inference rules. (Our treatment here is informal due to space limitations.) Our efficiency claim is based on several optimizations that we engineered into the design of the reifications. (See Section 6.) This paper provides a detailed introduction to inference graphs and explains how a step analyzer creates and evaluates an inference graph as it visits a system description.

The remainder of the paper is structured as follows. We first present necessary background in operational notations and describe the notations supported by Amalia (Section 2). We then describe the components that Amalia generates from an operational description of a notation’s semantics (Section 3), and explain how a step analyzer assembles these components into an inference graph while traversing an AST (Section 4). The steps of a system description are produced by evaluating the inference graph generated from the description’s AST (Section 5). We conclude with some discussion of issues raised by our work (Section 6).

2. Background

Before delving into the details of step analyzers and inference graphs, we recap the key ideas that underlie operationally defined notations. An operational semantics formalizes the notion of a step in an execution of a system. Amalia currently supports a restricted form of operational semantics suitable for notations that describe synchronization of actions in behaviors of a system, such as LOTOS [4] Esterel [3], or propositional linear-time temporal logic [14]. Because many of these ideas are abstract, we use concrete

example specifications written in a subset of the LOTOS notation throughout the paper. We now briefly describe the LOTOS subset and its operational semantics. We then introduce the assumptions we make about system representations. Finally, we recast the key ideas in the LOTOS semantics in a more general form, which we use to describe the design of a general step analyzer and to precisely define the variety of operational semantics that Amalia supports.

2.1. Lotos subset

LOTOS is a rich language for specifying event-driven behavior. A LOTOS specification comprises one or more *processes*, each of which is a computational entity whose internal structure can only be discovered by observing how it interacts with its environment. Amalia supports all of *pure* LOTOS, which deals with process synchronization, but not with data exchange. Due to space limitations, this paper considers a much smaller subset of LOTOS. This subset was chosen to illustrate different types of semantic rules, not to be representative of the LOTOS notation.

Briefly, a process can perform an *action*, which is an atomic event that other processes can observe, and thereby transform itself into another process. Notationally, we distinguish place holders for processes using uppercase letters from the middle of the alphabet (P or Q) and place holders for actions using lowercase letters from the beginning of the alphabet (a or b). We use multi-letter names to denote actual processes and actual actions, with the former written entirely in uppercase and the latter written entirely in lowercase. One exception is the termination of a process—an observable action in LOTOS—which is denoted by the Greek letter δ . If P denotes a LOTOS process, a step of P is a pair (a, P') , where a designates an action that P can perform and P' specifies how P is transformed when it performs a . Following convention, we write $P \xrightarrow{a} P'$ to mean (a, P') is a possible step of P ; in this case, P' is called a *derivative* of P . We refer to $P \xrightarrow{a} P'$ as a *step assertion*.

We will use two primitive LOTOS processes:

EXIT: can be observed only to terminate; doing so causes it to become the process *STOP*.

STOP: (also called “deadlock”) cannot engage in any actions.

In addition to the two primitive processes, we use three LOTOS operators:

Prefix: The process “ a prefix P ,” denoted $a;P$, performs a and then becomes P .

Disable: The process “ P disabled by Q ,” denoted $P \triangleright Q$, behaves like P unless it is interrupted by Q .

Parallel composition: Given a set of actions A , the process “ P and Q synchronizing on A ,” denoted $P \mid_A Q$,

$$\begin{array}{c}
\frac{}{EXIT \xrightarrow{\delta} STOP} \text{ [exit]} \\
\frac{P \xrightarrow{a} P', (a \neq \delta)}{P [> Q \xrightarrow{a} P'] [> Q]} \text{ [disL1]} \quad \frac{P \xrightarrow{a} P', (a = \delta)}{P [> Q \xrightarrow{a} P']} \text{ [disL2]} \quad \frac{Q \xrightarrow{a} Q'}{P [> Q \xrightarrow{a} Q']} \text{ [disR]} \\
\frac{P \xrightarrow{a} P', a \notin A \cup \{\delta\}}{P | A | Q \xrightarrow{a} P' | A | Q} \text{ [parL]} \quad \frac{Q \xrightarrow{a} Q', a \notin A \cup \{\delta\}}{P | A | Q \xrightarrow{a} P | A | Q'} \text{ [parR]} \quad \frac{P \xrightarrow{a} P', Q \xrightarrow{b} Q', (a \in A \cup \{\delta\}) \wedge (a = b)}{P | A | Q \xrightarrow{a} P' | A | Q'} \text{ [parD]}
\end{array}$$

Table 1. Semantic rules for some Lotos operators

behaves like P and Q running independently except on actions from A and on termination, when P and Q must synchronize.

An operational semantics for the LOTOS subset codifies *semantic rules* that prescribe how to derive step assertions for a composite process from the step assertions derivable from its parts (Table 1). Semantic rules are named for reference purposes; the names appear in brackets beside the rules. Step assertions in the *numerator* of a rule make up the rule’s *premises*. A rule with no premises is called an *axiom*; a rule with a single premise is called a *singular rule*; and a rule with two premises is called a *dual rule*. Thus, [exit] and [pref] are axioms, [parD] is a dual rule, and the remaining rules in Table 1 are singular rules. We refer to the step in a premise as a *premise step*. The numerator of a rule may also specify a *side-condition*, i.e., a boolean expression that must be satisfied for the rule to be applicable. For example, [disL1] applies only if the premise step, (a, P') , does not represent termination, as is indicated by the side-condition, $(a \neq \delta)$.

The *denominator* of a rule specifies a step assertion, called the *conclusion*, that is inferred using the rule. The LOTOS expression on the left-hand side of the conclusion is called the *subject* of the rule, and the step in the conclusion is called the *conclusion step*.³ As a mnemonic device, the names of our rules indicate the major operator of the rule’s subject and whether the rule is a dual rule (signified by an ending “D”), a singular rule whose premise step refers to the subject’s left operand (signified by an ending “L”), a singular rule whose premise step refers to the subject’s right operand (signified by an ending “R”), or an axiom (signified by the absence of a final uppercase letter). A rule is applied by instantiating the place holders in the rule with processes and actions that satisfy the rule’s premises and side-condition, if any. The rule then justifies the (instantiated) conclusion.

³The “premise,” “side-condition,” and “conclusion” terminology is standard, and the “numerator,” “denominator,” and “subject” terminology is borrowed from [2].

We illustrate the application of rules using the following process definitions, in which *ping* and *ctrlc* are assumed to denote atomic actions.⁴

$$\begin{array}{l}
PROC \hat{=} PDCC \mid [ctrlc] \mid EDCC \\
PDCC \hat{=} PING [> CTRLC] \\
PING \hat{=} ping; EXIT \\
CTRLC \hat{=} ctrlc; EXIT \\
EDCC \hat{=} EXIT [> CTRLC]
\end{array} \quad (1)$$

We can derive a step for *PROC* by the following reasoning. In [pref], we can instantiate a with *ping* and P with *EXIT* to conclude

$$PING \xrightarrow{ping} EXIT \quad (2)$$

Then, we can use (2) and [disL1] to justify the conclusion

$$PDCC \xrightarrow{ping} EXIT [> CTRLC] \quad (3)$$

Finally, (3) and [parL] justify the conclusion

$$PROC \xrightarrow{ping} (EXIT [> CTRLC] \mid [ctrlc] \mid EDCC) \quad (4)$$

By this reasoning, we conclude that *PROC* can perform the step $(ping, ((EXIT [> CTRLC] \mid [ctrlc] \mid EDCC))$.

2.2. System representations supported by Amalia

Amalia assumes that a system is represented by an abstract syntax tree whose structure is governed by a *class model*. A class model is a design abstraction that relates a collection of classes by *generalization* (or “kind-of”) and *aggregation* (or “part-of”) associations. When used to specify the abstract syntax of a notation, a class model contains: (1) a concrete class for each operator in the notation; (2) aggregation associations that relate composite classes to classes that represent their arguments; and (3) abstract

⁴“*PDCC*” for “*ping disabled by ctrl-c*” and “*EDCC*” for “*exit disabled by ctrl-c*.” For brevity, we use the abbreviations *PROC*, *PDCC*, *PING*, *CTRLC*, and *EDCC* in examples; they must be replaced by their definitions when applying rules or analyzing a process.

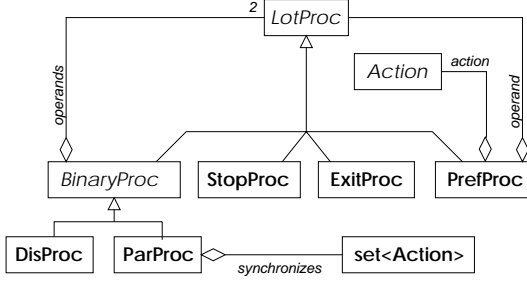


Figure 1. Class model for the Lotos subset

classes that factor and clarify aggregation associations and provide a most general class. Figure 1 presents a class model that describes the abstract syntax of our LOTOS subset. LOTOS processes are generalized by the abstract class *LotProc*. Concrete classes—*ExitProc*, *StopProc*, *PrefProc*, *DisProc*, and *ParProc*—represent operators. In the figure, an open arrow signifies generalization and an open diamond signifies aggregation.

Each operator class must provide a constructor with which to create instances of the operator, and access methods with which to retrieve its parts. For brevity, we elide these operations in graphical depictions of a class model, replacing them with the more concise aggregation associations. For example, the aggregations *action* and *operand* show that the constructor for *PrefProc* takes two arguments, one designating an *Action* AST and the other a *LotProc* AST. Constructors for binary operators—*DisProc* and *ParProc*—take two *LotProc* arguments. Moreover, the constructor for *ParProc* takes a third argument, which designates a set of synchronizing actions.

Notationally, we use OP to denote a generic operator, i.e., a concrete sub-class of the most general AST class, and $OP(\bar{X})$ to denote an OP -valued expression, where \bar{X} stands for the (possibly empty) list of arguments passed to the constructor of the concrete class associated with OP . We also distinguish special arguments to an operator class’ constructor, called *operands*: An argument $X_i \in \bar{X}$ is an operand of $OP(\bar{X})$ if X_i is an instance of the most general AST type. Every operand is an argument, but the converse is not true. For example, an instance of type *PrefProc* has a single operand, even though its constructor takes two arguments (the *Action* argument is not an operand). Consequently, we refer to prefix as a unary operator. We treat “primitive” systems, like *EXIT* and *STOP* in LOTOS, as nullary operators. Amalia currently supports nullary, unary, and binary operators.

An AST is an instance of a class model in the following sense: Each node is an instance of a concrete class, and children correspond to aggregation links that conform to the class model. Figure 2 shows the AST for the LOTOS process *PROC*, which is defined in (1). Nodes that

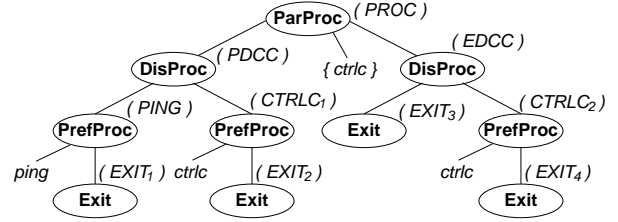


Figure 2. Example Abstract Syntax Tree

Axioms:

$$\frac{C_{\text{axiom}}(\bar{X})}{OP(\bar{X}) \rightarrow F_{\text{axiom}}(\bar{X})} \text{ [axiom]}$$

Singular Rules:

$$\frac{X_i \rightarrow S \quad C_{\text{sing}}(\bar{X}, S)}{OP(\bar{X}) \rightarrow F_{\text{sing}}(\bar{X}, S)} \text{ [sing]}$$

Dual Rules:

$$\frac{X_i \rightarrow S_1 \quad X_j \rightarrow S_2 \quad C_{\text{dual}}(\bar{X}, S_1, S_2)}{OP(\bar{X}) \rightarrow F_{\text{dual}}(\bar{X}, S_1, S_2)} \text{ [dual]}$$

Figure 3. Semantic rule types

represent operands are drawn as ovals and annotated with their class names; to simplify the drawing, we use concrete syntax to depict nodes that represent other arguments (e.g., action prefixes and sets of synchronizing actions). We provide abbreviations (in parentheses beside nodes) to refer to nodes in our examples.

The current implementation of Amalia assumes that steps are represented as ordered pairs, but a future version will allow any type to be used for representing steps (Section 6). For this reason, we place no assumptions on the representation of steps, other than that one is defined. If S denotes a step, we write $OP(\bar{X}) \rightarrow S$ to signify that S is a possible next step of $OP(\bar{X})$. We regard the LOTOS step assertion $P \xrightarrow{a} P'$ as an alternative way of writing $P \rightarrow (a, P')$.

2.3. Semantic rules supported by Amalia

An operational semantics consists of a set of inference rules, which specify how step assertions of operator expressions are derived from step assertions of their operands. Amalia currently supports three types of rules (Figure 3). In these rule types $C_{\text{axiom}}(\bar{X})$, $C_{\text{sing}}(\bar{X}, S)$, and $C_{\text{dual}}(\bar{X}, S_1, S_2)$ are boolean-valued expressions; $F_{\text{axiom}}(\bar{X})$, $F_{\text{sing}}(\bar{X}, S)$, and $F_{\text{dual}}(\bar{X}, S_1, S_2)$ are step-valued expressions; \bar{X} is the set of arguments to the subject expression; S , S_1 , and S_2 are premise steps; and $X_i, X_j \in \bar{X}$ are distinct operands of the subject expres-

sion. We use the constant boolean-valued expression *True* in cases where a rule has no side-condition.

For the LOTOS rules in Table 1, we have

$$\begin{aligned}
C_{\text{exit}}() &\hat{=} \text{True} & F_{\text{exit}}() &\hat{=} (\delta, \text{STOP}) \\
C_{\text{pref}}(a, P) &\hat{=} \text{True} & F_{\text{pref}}(a, P) &\hat{=} (a, P) \\
C_{\text{disL1}}(P, Q, (a, P')) &\hat{=} (a \neq \delta) \\
F_{\text{disL1}}(P, Q, (a, P')) &\hat{=} (a, P' [> Q]) \\
C_{\text{disL2}}(P, Q, (a, P')) &\hat{=} (a = \delta) \\
F_{\text{disL2}}(P, Q, (a, P')) &\hat{=} (a, P') \\
C_{\text{disR}}(P, Q, (a, Q')) &\hat{=} \text{True} \\
F_{\text{disR}}(P, Q, (a, Q')) &\hat{=} (a, Q') \\
C_{\text{parL}}(P, A, Q, (a, P')) &\hat{=} (a \notin A \cup \{\delta\}) \\
F_{\text{parL}}(P, A, Q, (a, P')) &\hat{=} (a, P' |A| Q) \\
C_{\text{parR}}(P, A, Q, (a, Q')) &\hat{=} (a \notin A \cup \{\delta\}) \\
F_{\text{parR}}(P, A, Q, (a, Q')) &\hat{=} (a, P |A| Q') \\
C_{\text{parD}}(P, A, Q, (a, P'), (b, Q')) &\hat{=} \\
&\quad (a \in A \cup \{\delta\}) \wedge (a = b) \\
F_{\text{parD}}(P, A, Q, (a, P'), (b, Q')) &\hat{=} (a, P' |A| Q')
\end{aligned}$$

When an operator appears in the subject of multiple rules, we require that the rules are *disjoint*—that is, at most one rule can apply to any (instantiated) premise step. This assumption is justified provided that the premises of different rules for the same operator involve different operands or, if two or more rules involve the same operand, provided that the side-conditions of these rules are contradictory. For example, [disL1] and [disR] are disjoint because they involve different operands, whereas [disL1] and [disL2] are disjoint because their side-conditions are contradictory.⁵

3. Inference Graphs

An inference graph is created by a step analyzer, for a particular expression, in order to compute the expression’s steps. The graph is an acyclic linked object structure whose objects (nodes) reify axioms and inference rules. It is *evaluated* by *firing* the nodes with no incoming edge, which causes a propagation of steps that flow out of nodes, along edges, and into other nodes, eventually resulting in the propagation of steps to the output of the graph. As a practical matter, a step analyzer assembles and evaluates an inference graph on-the-fly, as it traverses an expression’s AST representation. To simplify the presentation, however, this section explains only the structure of the graph.

Rule instances. The primitive concept in an inference graph is a class that reifies a semantic rule by providing an

⁵In practice, we have not encountered any formal notations whose semantic rules violate this assumption. A different formulation of inference graphs would allow us to relax this assumption, but at a performance penalty.

operation to simulate *firing* the rule to infer a step. Amalia compiles a rule named [op], with subject $OP(\bar{X})$, into a *rule class*, denoted RC_{op} . The constructor for RC_{op} is invoked with the arguments of $OP(\bar{X})$ to instantiate a *rule instance*, denoted $\text{RC}_{\text{op}}(\bar{X})$. The distinction is subtle: A rule class reifies a rule, which can apply to an arbitrary subject $OP(\bar{X})$; a rule instance reifies an instantiation of a rule over a particular expression.

Constructing a rule instance requires information about the particular expression. Table 2 shows the rule instances that a step analyzer creates when it traverses the AST of Figure 2. The first column indicates the subexpressions that engender the rule instances. For convenience, we define abbreviations for rule instances that indicate the type of the rule instance (“A” for “axiom,” “S” for “singular,” and “D” for “dual”) and, in the case of singular instances, the operand that provides the premise step (“L” for “left” and “R” for “right”); names are subscripted when necessary to make them unique.

Expressions	Rule instances
<i>PROC</i>	$S_1^L \hat{=} \text{RC}_{\text{parL}}(\text{PDCC}, \{\text{ctrlc}\}, \text{EDCC})$ $S_1^R \hat{=} \text{RC}_{\text{parR}}(\text{PDCC}, \{\text{ctrlc}\}, \text{EDCC})$ $D_1 \hat{=} \text{RC}_{\text{parD}}(\text{PDCC}, \{\text{ctrlc}\}, \text{EDCC})$
<i>PDCC</i>	$S_2^L \hat{=} \text{RC}_{\text{disL1}}(\text{PING}, \text{CTRLC}_1)$ $S_3^L \hat{=} \text{RC}_{\text{disL2}}(\text{PING}, \text{CTRLC}_1)$ $S_2^R \hat{=} \text{RC}_{\text{disR}}(\text{PING}, \text{CTRLC}_1)$
<i>PING</i>	$A_1 \hat{=} \text{RC}_{\text{pref}}(\text{ping}, \text{EXIT}_1)$
<i>CTRLC₁</i>	$A_2 \hat{=} \text{RC}_{\text{pref}}(\text{ctrlc}, \text{EXIT}_2)$
<i>EDCC</i>	$S_4^L \hat{=} \text{RC}_{\text{disL1}}(\text{EXIT}_3, \text{CTRLC}_2)$ $S_5^L \hat{=} \text{RC}_{\text{disL2}}(\text{EXIT}_3, \text{CTRLC}_2)$ $S_3^R \hat{=} \text{RC}_{\text{disR}}(\text{EXIT}_3, \text{CTRLC}_2)$
<i>EXIT₃</i>	$A_3 \hat{=} \text{RC}_{\text{exit}}()$
<i>CTRLC₂</i>	$A_4 \hat{=} \text{RC}_{\text{pref}}(\text{ctrlc}, \text{EXIT}_4)$

Table 2. Rule instances

The firing operation expects one step parameter for each operand that is referenced in the premise of the corresponding rule. A rule instance cannot be fired until the premise steps required by its firing operation become available. These premise steps arrive as the result of firing other rule instances. To formalize this behavior, we view rule instances as components in a data-flow framework. Specifically, a rule instance provides an input port, or an *in-flow*, and up to two special-purpose output ports, designated as

the *exit out-flow* and the *shunt out-flow* (Figure 4 (a–b)). The exit out-flow is used to forward conclusion steps, when the premise steps satisfy the corresponding side condition. Conversely, the shunt out-flow is used when the premise steps fail to satisfy the side condition. The pseudo-code in Figure 5 describes the logic of firing a rule instance using the terminology of Figure 3.⁶ Shunt out-flows enable daisy chaining of rule-instances with disjoint side conditions that apply to the same set of premise steps.

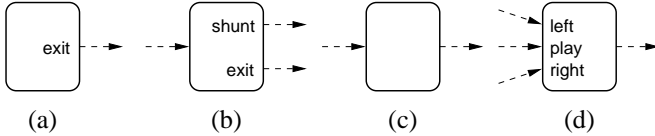


Figure 4. Flow interfaces of (a) axiom instances, (b) singular and dual instances, (c) s-adaptors, and (d) d-adaptors

- To fire an axiom instance, $RC_{\text{axiom}}(\bar{X})$:
If $C_{\text{axiom}}(\bar{X})$ then
send $F_{\text{axiom}}(\bar{X})$ to the exit out-flow
- To fire a singular instance, $RC_{\text{sing}}(\bar{X})$:
Receive a premise step S on the in-flow;
If $C_{\text{sing}}(\bar{X}, S)$ then
send $F_{\text{sing}}(\bar{X}, S)$ to the exit out-flow
else
send S to the shunt out-flow
- To fire a dual instance, $RC_{\text{dual}}(\bar{X})$:
Receive a pair of premise steps (S_1, S_2) on the in-flow;
If $C_{\text{dual}}(\bar{X}, S_1, S_2)$ then
send $F_{\text{dual}}(\bar{X}, S_1, S_2)$ to the exit out-flow
else
send (S_1, S_2) to the shunt out-flow

Figure 5. The firing logic of rule instances

Adaptors. Rule instances have well-defined connection interfaces. Still, they cannot be simply linked together to form an inference graph. Whereas all rule instances, regardless of type, propagate a single step along the exit out-flow, a dual instance expects a pair of steps over its in-flow. This disparity makes it impossible to connect the exit out-flow of a rule instance to the in-flow of a dual rule instance. We address this disparity by adapting the firing interface of a dual rule instance to that of a singular rule instance.

An *adaptor* wraps a rule instance with a firing interface that expects one step: *s-adaptors* are used with singular rule

⁶Amalia implements shunting in C++ using mixins and propagation of conclusion steps by dereferencing a target parameter, but such details are beyond the scope of this paper.

instances, which require only a single premise step, and *d-adaptors* are used with dual rule instances, which require an ordered pair of premise steps (Figure 4 (c–d)). An adaptor *plays* steps by sending them to its out-flow. An s-adaptor plays a premise step immediately upon receiving it, since singular instances require only a single premise step. However, a d-adaptor cannot play steps as it receives them, since it receives all steps of the left operand before it receives any steps of the right operand, and thus it cannot form an argument for a dual instance until after it has received all left-operand steps.

A d-adaptor has two modes of operation: *recording* and *playing* modes. While in recording mode, the d-adaptor assumes all incoming steps are left-operand steps; whereas in playing mode, it assumes that incoming steps are right-operand steps. In recording mode, the d-adaptor stores incoming steps in a local buffer. In playing mode, it pairs an incoming step with each step in this buffer and then marshals each pair to the dual rule instance, one after another. It switches from recording mode to playing mode on receiving a signal on the play in-flow.

Inference nodes. Essentially, the nodes in an inference graph are either *flow initiators* or *inference nodes*.⁷ A flow initiator consists of an axiom instance that is to be fired, such as A_1 in Table 2. An inference node, on the other hand, derives and then propagates conclusion steps in reaction to receiving premise steps. Rule instances and adaptors are used to assemble inference nodes.

The inference nodes for an expression are determined as follows. The set of non-axiom rule instances for an expression are partitioned into subsets based on the operand(s) that supply the premise steps. For example, the three rule instances for *PROC* (Table 2) are partitioned into three singleton subsets, as the premise steps for all rules are supplied by different sets of operands. In contrast, the three rule instances for *PDCC* are partitioned into two subsets—since the left operand supplies the premise steps for both S_2^L and S_3^L , these rule instances form one subset of the partition and, since the right operand supplies the premise step for S_2^R , this latter rule instance forms the other.

An inference node comprises a stack of non-axiom rule instances, wrapped by an adaptor. In order to produce one inference node for each subset in the partition, we stack the rule instances that are contained in the same subset. Stacking entails connecting the shunt out-flow of all but the last rule instance to the in-flow of the next one and joining all of the exit out-flows. Such a stack is then fitted with an adaptor by connecting the adaptor’s out-flow to the in-flow of the first rule instance in the stack. Figure 6 shows the dual node for *PROC* (top) and the singular node for *PDCC* that operates on left-operand steps (middle). It also shows a

⁷There is a third kind of node, called a pivot, which we introduce in Section 4.

shunt node (bottom), which is what we call a singular node that sends everything it receives to its shunt out-flow. Shunt nodes are needed if an expression engenders a dual node, but does not produce both a singular node that operates on left-operand steps and a singular node that operates on right-operand steps. As described below, the singular nodes are needed to shunt unhandled premise steps to the dual node.

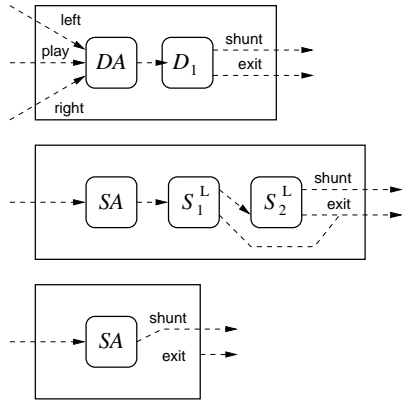


Figure 6. Example inference nodes

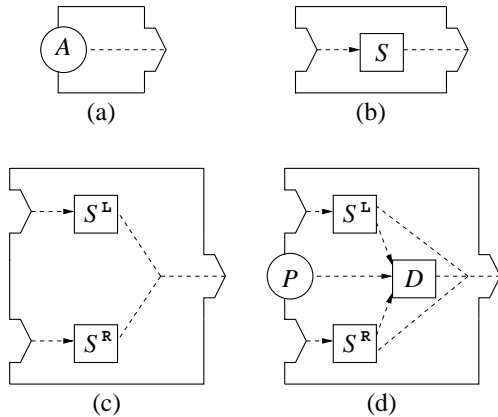


Figure 7. Layouts of some boxes

4. Assembly of inference graphs

A step analyzer assembles an inference graph by traversing the AST of an expression. The step analyzer is a *visitor class*, which means that for each operator OP , it provides a `visitOP` method that expects to be passed an AST whose root has type OP . The assembly process is a recursive elaboration of the inference graph, which requires a correspondence between the inference graph and the operand structure of an AST. We achieve this correspondence with a concept called a *box*, which aggregates the flow initiators and

inference nodes engendered by a subexpression and, if the box includes a dual node, a *pivot*. A pivot behaves like a switch. When fired, it causes the dual node to transition into playing mode.

Boxes. We define a box in terms of its purpose and its contents. Boxes induce a correspondence between the operand structure of an expression and the inference graph that computes the steps of the expression. A box contains all of the nodes in the graph that share the same exit out-flow. Additionally, if a box contains a dual node, it also contains a pivot.

The layout of a box depends on the number and types of nodes. Figure 7 shows layouts for some typical boxes. Circles represent the different types of leaf nodes—flow initiators are labeled “A” (because they reify axioms) and pivots are labeled “P”. Rectangles represent inference nodes; the names of inference nodes follow the naming conventions for rule instances. An arrow-shaped cut-out represents an in-flow of a box and an arrow-shaped push-out represents an out-flow. To reduce clutter, we have not labeled in-flows and out-flows of inference nodes, distinguishing them instead by notational conventions. Specifically, the in-flows of a dual node are drawn with the left in-flow on top, the play in-flow in the middle, and the right in-flow on the bottom. The exit out-flow of an inference node is represented by a dashed line connecting the node to the out-flow of the box. The shunt out-flow of a node either forwards premise steps to another node in the same box or connects to a *ground*, which just discards its input. In the former case, the shunt out-flow is shown as a dashed arrow connecting the two inference nodes; in the latter case, it (and the ground) is (are) not shown.

We assemble an operator box, B , for an expression $OP(\bar{X})$ as follows:

- For each axiom instance, A : Install A as a flow initiator, joining the out-flow of A to the out-flow of B .
- For a singular node, S : Create an in-flow of B to provide an in-flow for S ; join the exit out-flow of S to the out-flow of B ; and connect the shunt out-flow of S as follows:
 - If S operates on left-operand steps and $OP(\bar{X})$ engenders a dual node D , connect the shunt out-flow of S to the left in-flow of D .
 - If S operates on right-operand steps and $OP(\bar{X})$ engenders a dual node D , connect the shunt out-flow of S to the right in-flow of D .
 - Otherwise, connect the shunt out-flow of S to a ground.
- For a dual node, D : Connect the left and right in-flows of D to the shunt out-flows of singular nodes that oper-

ate on, respectively, left- and right-operand steps; connect a pivot to the play in-flow of D ; join the exit-outflow of D to the exit out-flow of B ; and ground the shunt out-flow of D .

For example, the box for $PING$ is assembled as shown in Figure 7 (a), with a flow initiator (A) that fires the axiom instance A_1 . (See Table 2 for definitions of the rule instances.) The box for $PDCC$ is assembled as shown in Figure 7 (c), with a singular node (S^L) containing S_2^L and S_3^L , and a singular node (S^R) containing S_2^R . Similarly, the box for $PROC$ is assembled as shown in Figure 7 (d), with a pivot (P); a singular node (S^L) containing S_1^L ; a singular node (S^R) containing S_1^R ; and a dual node (D) containing D_1 .

Putting it all together. Before being used to visit an expression, a step analyzer is supplied a box in-flow, called a *target*, in which to connect the inference graph produced during the visit. When visiting a composite expression, a box is created for the root of the expression, and new step analyzers are instantiated to assemble inference subgraphs from the expression’s operands. Prior to visiting these operands, each new step analyzer is supplied an in-flow of the box that corresponds to the root of the expression. This recursive elaboration of the inference graph as a hierarchy of boxes continues until we visit the leaves of the composite expression. Each leaf corresponds to a box that contains only flow initiators.

More specifically, when invoked to visit an expression $OP(\bar{X})$, if this expression is the subject of any semantic rules, the `visitOP` method assembles a box B for $OP(\bar{X})$, as previously described. Then, for each operand $X_i \in X$, starting with the leftmost operand, if B has an in-flow I that operates on steps of X_i , the `visitOP` method instantiates a new `StepAnalyzer` visitor with target I and invokes a visit of X_i to generate an inference graph for X_i that connects to B on I . Alternately, if $OP(\bar{X})$ is not the subject of any semantic rules, then it has no steps. In this case, therefore, the `visitOP` method simply returns without assembling an inference graph, as the inference graph for $OP(\bar{X})$ is empty. Note that a step analyzer visits a subexpression only if some inference node operates on steps of the subexpression. Thus, not all subexpressions are necessarily visited.

For example, the inference graph for $PROC$ contains seven boxes (Figure 8). Each box is produced by visiting an object listed in the first column of Table 2. We label inference nodes with the rule instances that they contain. The `visitParProc` method of the “root” visitor creates the root (rightmost) box, which contains a node for the dual rule instance, D_1 ; a node for the singular rule instance, S_1^L , that operates on left-operand steps; a pivot, P_1 ; and a singular rule instance, S_1^R , that operates on right-operand steps. It then creates a “child” visitor, passing the in-flow of the singular node for S_1^L as the target, and invokes a visit of its left

operand, $PDCC$, by the new visitor. The `visitDisProc` method of this new visitor creates the box for $PDCC$, containing nodes for S_2^L and S_3^L , and also for S_2^R . It then instantiates two additional child visitors and recursively invokes visits of, first $PING$, and then $CTRLC_1$; it passes the in-flow of the inference node for S_2^L and S_3^L as the target for the first of these visitors and the in-flow of the inference node for S_2^R as the target for the other. When the child visitors all return, the root visitor proceeds to instantiate another child visitor and to invoke a visit of the right operand, $EDCC$; this visit is similar to the visit of $PDCC$ by the first child visitor.

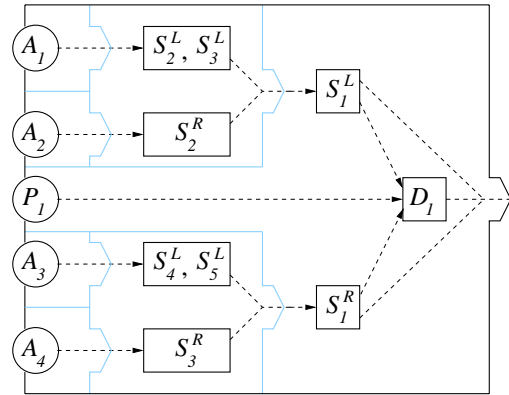


Figure 8. Inference graph for $PROC$

5. Evaluation of inference graphs

A step analyzer evaluates the inference graph created from an expression to compute the steps that the expression can perform. The graph is evaluated in a sequence of flows, which is produced by firing flow initiators and pivots, in the order of their creation. Firing a flow initiator starts a flow; when this flow terminates, the next flow initiator or pivot is fired. Firing a pivot switches a dual node into playing mode, which affects the flows produced by firing subsequent flow initiators. For example, the inference graph in Figure 8 is evaluated in four flows, produced by firing the nodes on its left side, from top to bottom. The first flow is produced by firing A_1 and the second by firing A_2 . Then, P_1 is fired, switching D_1 into playing mode. Finally, the third flow is produced by firing A_3 and the fourth by firing A_4 . We show below that the first and last flows propagate steps of $PROC$ to the out-flow of the inference graph; the other flows terminate without generating any steps of $PROC$.

Once started, a flow proceeds by sequentially firing inference nodes, where each node that is fired is *enabled* by firing the previous node. A singular node is enabled when a step propagates to its in-flow. Similarly, a dual node is enabled when a step propagates to one of its in-flows; how-

ever, in playing mode, a dual node is also enabled when no step is propagated if (1) it has not already played every pair produced by pairing one of its buffered left-operand steps with its most recently received right-operand step, and (2) no other node on the path from the dual node to the root is enabled. Firing an inference node *disables* it and either enables another inference node or terminates the flow. From these definitions, it follows that a flow terminates when a step is not propagated if all dual nodes either are in recording mode or have already played all the argument pairs that they can form. A flow also terminates when a step propagates to the out-flow of the inference graph if no dual node is enabled. We now explain how nodes are fired.

A flow initiator fires the associated axiom instance. Thus, if the instantiated side-condition is true, it propagates a step; otherwise, it has no effect. For example, firing A_1 propagates the step ($ping, EXIT_1$), thereby enabling the singular node containing S_2^L and S_3^L (Figure 8).

A singular node, S , fires when a premise step propagates to its in-flow, which is the in-flow of the s-adaptor at the front of S . The s-adaptor plays the premise step to its out-flow, which causes the first rule in S to fire. If the rule's side-condition is satisfied, this rule instance transforms the premise step into a conclusion step, which it propagates to the exit out-flow of S ; otherwise, it shunts the step to the next rule instance in S (Figure 6, middle). This next rule instance, in turn, will either propagate a conclusion step or shunt the premise step. If the premise step does not satisfy the side-conditions of any of the rule instances in S , it is eventually shunted to the shunt out-flow of S , which connects either to a dual node or to a ground. Thus, firing a singular node, S , produces one of three possible outcomes:

- If some rule instance in S applies to the premise step, S infers a conclusion step and propagates it to the exit out-flow
- If no rule object in S applies to the premise step and the shunt out-flow of S connects to the in-flow of a dual node, S shunts the unhandled premise step to the dual node.
- If no rule object in S applies to the premise step and the shunt out-flow of S connects to a ground, S shunts the premise step to the ground.

For example, when A_1 propagates ($ping, EXIT_1$), the rule instance S_2^L uses it to infer the conclusion step ($ping, (EXIT_1 [> CTRLC_1])$). This latter step is then propagated to the in-flow of S_1^L , which infers and propagates ($ping, ((EXIT_1 [> CTRLC_1] | ctrlc | EDCC)$) to the out-flow of the inference graph. In both of these cases, a rule instance in a singular node applies. Similarly, in the second flow, ($ctrlc, EXIT_2$) is propagated first by A_2 and then by

S_2^R . In this case, however, S_1^L does not apply, and so it shunts the step to the dual node.

While in recording mode, a dual node is enabled only by a premise step that propagates to its left in-flow, which is the in-flow of a d-adaptor. In this case, the d-adaptor adds the premise step to its local buffer of left-operand steps; it must wait for right-operand steps to pair with the buffered steps. Thus, a dual node does not propagate a step when fired in recording mode. For example, the dual node in Figure 8 buffers the left-operand step ($ctrlc, EXIT_2$) that is shunted by S_1^L , terminating the second flow.

While in playing mode, a dual node, D , may be enabled by a step that propagates to its right in-flow or by an inference node that does not propagate a step (under the circumstances discussed previously). If enabled by a step that propagates to its right-inflow and if the buffer of left-operand steps in its d-adaptor is empty, D does not propagate a step. Otherwise, the d-adaptor forms an argument pair that has yet to be played through the stack of dual rule instances in D and it plays the new pair. This pair flows through the stack in D like a single-step argument flows through a stack of singular rule instances in a singular node. By construction, the shunt out-flow of D is grounded. Thus, if some rule instance in D applies to the pair, D propagates a conclusion step to its out-flow; otherwise, it does not propagate a step. For example, in the third flow produced by the inference graph of Figure 8, ($\delta, EXIT$) is propagated by A_3 , shunted by S_4^L , propagated by S_5^L , and shunted by S_1^R . The dual node then pairs the stored step with this latter step and plays it to D_1 , which shunts the pair to a ground. Similarly, in the fourth flow, ($ctrlc, EXIT_4$) is propagated by A_4 and S_3^R , shunted by S_1^R , and paired with the stored step; then the pair is played to D_1 . This time, however, D_1 infers the step ($ctrlc, (EXIT_2 | ctrlc | EXIT_4)$), and propagates it to the inference graph's out-flow.

As noted previously, a step analyzer does not sequentially generate and then evaluate an inference graph; rather, it dynamically builds and evaluates the graph on-the-fly, as it visits an AST. On-the-fly evaluation is possible because a subexpression's inference graph is no longer needed when the visit of the subexpression returns. Thus, at any point during analysis of an expression, the inference graph contains boxes for only those subexpressions that are currently being visited. For example, while $EXIT_1$ of $PROC$ is being visited, the inference graph contains boxes for only $EXIT_1$, $PING$, $PDCC$, and $PROC$. When the visits of $EXIT_1$ and $PING$ return, their boxes are deallocated, so that, while $EXIT_2$ is being visited, the inference graph contains boxes for only $EXIT_2$, $CTRLC_1$, $PDCC$, and $PROC$.

6. Discussion

A step analyzer comprises one level of an Amalia generated framework. The next level uses step analyzers to create labeled transition systems—a common abstraction used to reason about behavior. Other types of analyses, such as model checking, can be similarly refined down to operations on ASTs by layering them on labeled transitions systems. By this technique, Amalia’s step analyzers allow higher-level verification tools to be packaged as lightweight components [16]. In this section, we discuss some issues raised by this work and mention some on-going research.

6.1. Efficiency

The ability to operate directly on ASTs allows a lightweight component to be more efficiently integrated into a larger environment than a stand-alone tool. It forestalls the need to transform ASTs, which is a common internal form used in software engineering environments, into a form suitable for processing by an external analysis tool and to parse and interpret the output of the tool for processing by other tools in the environment. Such translations exchange a lot of data through input/output streams and complicate the architecture of a software engineering environment. Moreover, a stand-alone tool must either be invoked as a separate process while other tools wait or be running all the time on a dedicated server.

Our earlier research used a step analyzer for LOTOS that did not build an inference graph, but synthesized steps for an expression in a strictly bottom-up fashion. Briefly, when visiting a composite expression, it invoked visits of all subexpressions that needed to supply premise steps and stored the premise steps in local buffers. It then iterated through these buffers, instantiating rules with premise steps to infer steps of the composite expression. While conceptually simple, this bottom-up approach creates ancillary data structures, which are used once and then destroyed. In contrast, an inference graph is created and evaluated on-the-fly in a demand-driven fashion. Only dual nodes buffer premise steps, and they buffer only those premise steps of their left operands that are not handled by a singular node.

6.2. Correctness

For efficiency, a bottom-up step analyzer must interleave knowledge about different rules for an operator with custom algorithms that manipulate the data structures containing the premise steps. This interleaving obscures the inferring logic and makes it difficult to reason about whether the step analyzer is correct. Moreover, much of the logic of an algorithm for automatically generating bottom-up step analyzers would entail checking semantic rules for special

properties in order to determine how to optimize the analyzer. It would be difficult to prove (or even rigorously argue) that the generation algorithm was correct. Another avenue to explore for achieving efficiency without sacrificing transparency is to use aspect-oriented programming [12] or aspectual components [13] to weave the evaluation of rules into custom algorithms that manipulate data structures containing the premise steps. These approaches facilitate transparency but lack a formal basis in which to prove correctness.

Our formalization of inference graphs allows a proof that this abstraction is consistent with an operational semantics and also allows reasoning about specific inference graphs to justify some optimizations that Amalia performs. Intuitively, we regard a *well-formed* inference graph as denoting a sequence of steps, which models the steps produced on the graph’s out-flow during evaluation. This sequence is defined compositionally, in terms of the sequences denoted by the inference sub-graphs that supply premise steps to the root node of an inference graph. Provided that the nodes of an inference graph *reify* certain semantic rules, an induction argument can show that an inference graph produces a step if and only if the step is derivable, where the induction is performed on the length of a derivation. Precise definitions for what it means for an inference graph to be well-formed and for nodes to reify rules are based on our classification of semantic rules (Figure 3). In on-going research, we are using transformation techniques to show that Amalia-generated step analyzers produce inference graphs that are well-formed and that reify the necessary semantic rules.

6.3. Automated analyzer generators

Other researchers have looked at automatically generating analyzers from formal-semantics descriptions. For example, CENTAUR [5] maps specifications in natural semantics into Horn clauses in Prolog. Another example is SPARE [17], which synthesizes analysis algorithms from denotational-semantic specifications. However, these generators produce stand-alone analysis tools; additionally, the CENTAUR environment requires an online Prolog engine in order to run.

The Concurrency Workbench of North Carolina (CWB-NC) is a powerful toolkit for analyzing operational specifications [7]. It provides a Process Algebra Compiler (PAC) to enable use of the CWB-NC with different design notations. Like Amalia, the PAC takes as input definitions of a notation’s abstract syntax and operational semantics (as SOS rules). From these definitions, the PAC generates semantic routines (the main routine being a step analyzer) that a user can insert into the CWB-NC. Thus, whereas Amalia automatically tailors high-level analyses directly to an ab-

stract syntax, the PAC generates a front end that interfaces with the CWB-NC.

Finally, many general-purpose theorem provers, e.g., Isabelle [11] and HOL [6], allow inference rules to be represented declaratively. In particular, a framework developed by Day and Joyce [9] uses an embedding of a notation's semantics in HOL to define how a specification determines a "model," a HOL type by which the framework formalizes a next-step relation. Of course, theorem provers are themselves heavy-weight tools. The framework, however, uses symbolic functional evaluation to allow certain analyses (e.g., consistency and completeness checking and model checking) without using a theorem prover [8].

6.4. Validation studies

We have validated Amalia using a prototype implementation, written in C++. This prototype was used to build lightweight analyzers for two notations, pure LOTOS and linear-time temporal logic; the latter case study is described in [16]. Step analyzers for the notations were automatically generated from semantic rules and used to refine two higher level analysis components. Two additional case studies are in progress to validate extended prototypes, which will generate the equivalent of step analyzers for more general relations. The first prototype will be used to generate a lightweight analyzer that determines the free variables in a program from an inductive definition of the free-variable relation. The second will be used to generate a lightweight tool that analyzes programs to determine potential communication patterns. The step analyzers generated by this prototype will simultaneously derive steps for multiple inter-related relations. These new prototypes will generalize the notion of a "step" for an inductively defined relation.

Acknowledgements. The authors wish to thank Hamid Alavi, Grant Birchmeier, and the anonymous referees for careful reading of and helpful comments on earlier drafts of this paper. This work was partially supported by NSF grants CCR-9896190, CCR-9901017, and EIA-0000433.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, MA, 1987.
- [2] I. Attali and D. Parigot. Integrating natural semantics and attribute grammars: The Minotaur system. Tech. Rep. 2339, INRIA Sophia Antipolis, 1994.
- [3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language; design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
- [4] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comp. Netw. ISDN Sys.*, 14(1), 1987.
- [5] P. Borras et al. Centaur: The system. In *Proc. SIGSOFT'88, Third Symp. Software Development Environments*, Boston, 1988.
- [6] A. J. Camilleri. Mechanizing CSP trace theory in Higher Order Logic. *IEEE Trans. Softw. Eng.*, 16(9):993–1004, September 1990.
- [7] R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. To appear in *Science of Computer Programming*, 1999.
- [8] Nancy A. Day and Jeffrey J. Joyce. Symbolic functional evaluation. In *12th Internat. Conf. Theorem Proving in Higher Order Logics*, LNCS 1690, pp. 341–358, Springer, 1999.
- [9] Nancy A. Day and Jeffrey J. Joyce. A framework for multi-notation specification and analysis. In *Fourth IEEE Internat. Conf. Requirements Engineering*, June 2000. To appear.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
- [11] B. Gramlich and F. Pfenning. Strategic principles in the design of Isabelle. In *Work. Strategies in Automated Deduction*, pp. 11–16, Lindau, Germany, July 1998.
- [12] G. Kiczales et al. Aspect oriented programming. In *European Conf. Object-Oriented Programming*, 1997.
- [13] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.
- [14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.
- [15] G. D. Plotkin. A structural approach to operational semantics. Tech. Rep. DAIMI bFN-19, Computer Science Department, Aarhus University, 1981.
- [16] R. E. K. Stirewalt and L. K. Dillon. A component-based approach to building formal analysis tools. In *Proc. 2001 Internat. Conf. Software Engineering*, 2001. To appear.
- [17] G. A. Venkatesh and C. N. Fischer. Spare: A development environment for program analysis algorithms. *IEEE Trans. Softw. Eng.*, 18(4):304–318, April 1992.