IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 29, NO. 2, FEBRUARY 2003 133
http://dx.doi.org/10.1109/TSE.2003.1178052.

Inference Graphs: A Computational Structure Supporting Generation of Customizable and Correct Analysis Components

Laura K. Dillon, *Member*, *IEEE*, and R.E. Kurt Stirewalt, *Member*, *IEEE Computer Society*

Abstract—Amalia is a generator framework for constructing analyzers for operationally defined formal notations. These generated analyzers are components that are designed for customization and integration into a larger environment. The customizability and efficiency of Amalia analyzers owe to a computational structure called an inference graph. This paper describes this structure, how inference graphs enable Amalia to generate analyzers for operational specifications, and how we build in assurance. On another level, this paper illustrates how to balance the need for assurance, which typically implies a formal proof obligation, against other design concerns, whose solutions leverage design techniques that are not (yet) accompanied by mature proof methods. We require Amalia-generated designs to be transparent with respect to the formal semantic models upon which they are based. Inference graphs are complex structures that incorporate many design optimizations. While not formally verifiable, their fidelity with respect to a formal operational semantics can be discharged by inspection.

Index Terms—Amalia, analysis software, engineering trade-offs, inference graphs, operational semantics, program transformations, proofs of correctness, transparent design.

1 Introduction

Automated software-engineering environments (ASEs) create and manipulate representations of specifications, designs, and programs (hereafter called *system descriptions*). We recently developed the Amalia generator,[1] which generates software that analyzes the behavior of system descriptions that are defined via an operational semantics. Amalia-generated analyzers are not stand-alone tools, but rather are components that can be customized for integration into an ASE [1]. Consequently, these analyzers must satisfy design concerns that do not arise in stand-alone analysis/verification tools. To address these concerns requires the use of advanced object-oriented design techniques and technologies, for which mature proof methods are not yet available. This paper investigates how to assure the correctness of Amalia-generated analyzers without violating or unduly constraining these other design concerns.

We designed Amalia to generate analysis and verification software subject to three concerns. First, analysis software should support *extension* and *contraction*—in the sense suggested by Parnas [2]—so that it can be easily assembled into useful subsets. An important class of contractions enables a tool designer to incorporate one or more simplifying assumptions about the specifications that the tool will be used to analyze, such as that the specifications are known to be deterministic or finite state. Second, analysis software should be designed to facilitate its integration into a larger ASE [1]. In short, this means that generated analyzers should interoperate with environment-specific internal representations and should do so without unduly constraining the design of these representations. Third, it must be possible to demonstrate that the analyzer itself correctly implements its intended analysis. This paper explains how Amalia-generated software satisfies these concerns. Specifically, we extend the results of [1], [3], which address concerns 1 and 2, with an analysis of the correctness of the generated software, thereby also addressing concern 3.

What makes this problem difficult is that each of the aforementioned concerns constrains the underlying design of the generated analyzer. For example, we could provide assurance by requiring a formal proof of correctness. However, to satisfy thfis requirement would mean limiting the use of design techniques and language constructs to those for which mature proof techniques are available. Such a limitation is unacceptable in the context of the other concerns. For example, in [1], we argue that the use of the visitor pattern [4] is key to integration and that GenVoca generators [5] are a key enabling technology for supporting extension and contraction. Later in this paper, we argue for the use of mixin classes [6] in order to improve efficiency. None of these advanced design methods are supported by mature proof techniques.

Because our discipline currently lacks a rigorous engineering paradigm for simultaneously optimizing concerns such as these, we must make design trade-offs. This paper demonstrates how we use designs that are *transparent* with respect to the formal semantic models upon which

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

• *The authors are with the Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824.*
E-mail: {ldillon, stire}@cse.msu.edu.

Manuscript received Feb. 2002; revised July 2002; accepted Sept. 2002.
Recommended for acceptance by M.J. Harrold.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117519.

they are based to accommodate the needs of integration and extension/contraction without conceding assurance. Our thoughts on transparency follow the ideas of Boyle et al. who claim that to "build trust" into compilers requires viewing the verification of compiler code as a sequence of design transformations, each of which is "correct by inspection" [7]. In this paper, we demonstrate that a key component in Amalia-generated analyzers is a transparent implementation of an operational semantics. To adequately address the integration and extension/contraction concerns and also argue transparency, we integrate ideas and methods from different subdisciplines, such as formal verification, compiler-construction theory, domain analysis and reuse, and object-oriented design.

In [1], we argued that extension/contraction can be addressed by decomposing analysis algorithms into a sequence of *reusable refinements* in the GenVoca style [5]. In the domain of analysis software, one such refinement is called a *step analyzer*, which is used to compute a *labeled transition system (LTS)* representing the behaviors of an operationally-defined specification. The specification corresponds to a state in the LTS. Given a state, the step analyzer computes its immediate successors. Labeled transition systems are the basis for a variety of different analysis tools (e.g., model checkers, test oracles, etc.).

By exploiting ideas from advanced object-oriented design (specifically the visitor pattern [4]), we can generate step analyzers for a given internal representation without imposing undue constraints on the design of that representation. Moreover, by virtue of being able to generate step analyzers, a designer can automatically refine analysis algorithms within this domain to operate on a given internal representation. While a case study integrating an Amalia generated analyzer with a second party tool has yet to be completed, generators and design patterns represent the current best practice for addressing integration and extension.

The step analyzer is thus a crucial component in Amalia-generated analyzers. These generated step analyzers operate by creating and executing a data-flow structure called an *inference graph* as a side effect of traversing an internal representation [3]. Inference graphs balance the need to efficiently compute the transitions leaving a state in an LTS against the need to demonstrate the correctness of the analysis with respect to the operational semantics of the language used to describe the state (specification).

The computational results that a step analyzer must compute are specified using an operational semantics in a form that admits parameterization by a particular abstract syntax notation (Section 2). We first recast the analysis that a step analyzer must perform denotationally, as a function over specifications, and prove that this function correctly implements an operational semantics (Section 3). We then present the inference graph (Section 4) and demonstrate how the graph is a transparent implementation of the denotation function (Section 5). In addition to assurance, other concerns that we address include integration and efficiency of step analysis. We conclude the paper with a discussion of lessons learned and related and future research (Section 6).

2 Background

A step analyzer traverses the internal representation of an operational specification. This internal representation can be thought of as an *abstract syntax tree (AST)*, but it might be implemented in a variety of different ways, subject to the concerns of a larger environment and also to the whims of the original designer. Because integration into an existing environment is one of the driving concerns in Amalia, the step analyzer should impose only minimal constraints on this representation, as every such constraint could violate an assumption made elsewhere in the environment. Additionally, assurance dictates that we demonstrate that the step analyzer computes results according to the operational semantics of the specification being analyzed.

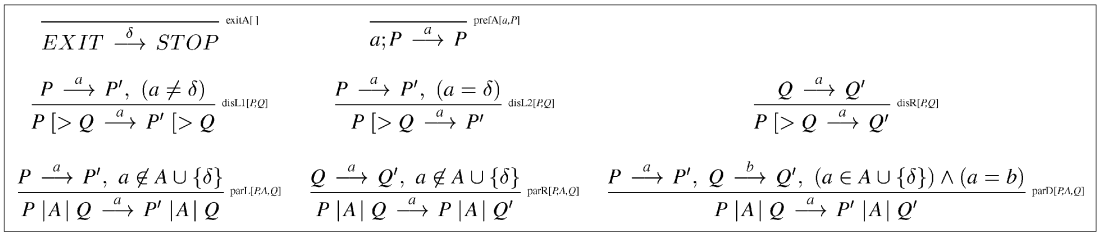
This section provides necessary background for understanding these problems and the techniques that we use to address them. We begin by describing operational semantics, which formalizes the notion of a step in an execution of a system. Because many of these ideas are abstract, we use concrete example specifications written in a subset of the LOTOS notation (Section 2.1). We then introduce a textual metanotation, which uses expressions for constructing and navigating "abstract" ASTs without reference to the details of a particular internal representation (Section 2.2). This metanotation permits us to give a generic description of the procedure by which a step analyzer assembles an inference graph. To describe how we instantiate this procedure with a specific internal representation for LOTOS, we provide a *class model*, which articulates the abstract syntax in terms of implementation classes and operations (Section 2.3). We then describe how visitors allow us to implement a step analyzer so as not to unduly constrain the design of an existing internal representation (Section 2.4). Finally, we briefly introduce the Amalia toolsuite and identify where step analyzers fit (Section 2.5).

2.1 Operational Semantics and the Lotos Subset

Amalia currently supports a restricted form of operational semantics, called *structural operational semantics* [8], or *SOS*, suitable for notations that describe synchronization of actions in behaviors of a system, such as LOTOS [9], or propositional linear-time temporal logic [10]. LOTOS is a rich language for specifying event-driven behavior. A LOTOS specification describes one or more concurrent *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 *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

TABLE 1
Semantic Rules for Some LOTOS Operators



beginning of the alphabet (a or b). We use multiletter 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.

We use two primitive LOTOS processes:

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

STOP: This process cannot engage in any actions.

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

Prefix: The process $a;P$, pronounced "a prefix P," performs the action a and then continues to behave like the process P.

Disable: The process $P\ [>\ Q$, pronounced "P disabled by Q," behaves like process P unless it is interrupted by process Q, at which point it will continue to behave like Q.

Parallel composition: The process $P \mid A \mid Q$, pronounced "P and Q synchronizing on A," behaves like process P and process Q running independently, except on actions from the set A and on termination, when P and Q must synchronize.

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 by performing action 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*.

A SOS specification for the LOTOS subset consists of *semantic rules* that prescribe how to derive step assertions for a composite process from the step assertions derivable from its parts (Table 1). Step assertions in the *numerator* of a rule make up the rule's *premises*. Each semantic rule is named for reference purposes; we show the (parameterized) name beside the rule. A rule with no premises (e.g., exitA[ ] and $\mathrm{prefA}[a, P]$) is called an *axiom*. 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, $\mathrm{disL1}[P, Q]$ applies only if the action a in the premise step (a, P') is not 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*.[2]

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. A sequence of conclusions produced by applying rules in an iterative manner is called a *derivation*. For example, consider the following process definitions, in which $ping$ and $ctrlc$ denote atomic actions.[3]

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

The following example derivation produces one step for $PROC$:

$$\begin{array}{lcl} PING & \xrightarrow{ping} & EXIT \\ PDCC & \xrightarrow{ping} & EXIT\ [>\ CTRLC \\ PROC & \xrightarrow{ping} & (EXIT\ [>\ CTRLC) \\ & & \mid [ctrlc] \mid EDCC \end{array}$$

This sequence of conclusions is justified by applying, in succession, the rules: $\mathrm{prefA}[ping, EXIT]$, $\mathrm{disL1}[PING, CTRLC]$, and $\mathrm{parL}[PDCC, [ctrlc], EDCC]$. This derivation shows that $PROC$ can perform the step

$$(ping,\ ((EXIT\ [>\ CTRLC) \mid [ctrlc] \mid EDCC)).$$

A similar derivation produces a second step for $PROC$, namely, $(ctrlc,\ (EXIT \mid [ctrlc] \mid EXIT))$.

2.2 Semantic Rules Supported by Amalia

To represent generic semantic rules, we use a metanotation that refers to operators in the abstract syntax of a specification language. Let OP denote an operator, or type of expression, in the language. Then, $OP[\overline{X}]$ denotes an instance of a specification, also called an abstract syntax tree. Here, $\overline{X}$ stands for the (possibly empty) list of arguments, most of which will be subexpression ASTs, called *operands*.

2. "premise," "side-condition," and "conclusion" terminology is standard, and the "numerator," "denominator," and "subject" terminology is borrowed from [11].

3. "$PDCC$" 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.

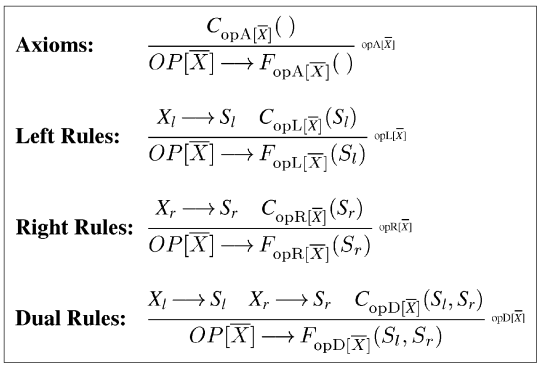


Fig. 1. Semantic rule types.

Currently, Amalia supports only *binary*, *unary*, and *nullary* operators. For example, in LOTOS, disable ($[>$) and parallel composition ($|\cdots|$) are binary operators; whereas action prefix (;) is a unary operator and "primitive" processes, such as $EXIT$ and $STOP$, are nullary operators. As a notational convention, when the major operator OP for an AST $OP[\overline{X}]$ is a binary operator, we use X_l to denote the first operand (the "left" operand) and X_r to denote the second operand (the "right" operand), and we assume $X_l, X_r \in \overline{X}$; if the major operator is a unary operator, we use X_l to denote the only operand and again we assume $X_l \in \overline{X}$. In general, $\overline{X}$ may contain arguments that are not operands, such as the set of actions A (in Table 1) when OP denotes the parallel composition operator.

Each rule is classified according to the number of step assertions in its premise and to the operand(s) that the premise steps refer to. A rule with no premise steps is called an *axiom*; a rule with a single premise step is called a *singular rule*, which we further distinguish as a *left rule* or a *right rule*, depending on whether the premise step refers to the left operand or to the right operand of the rule's subject; and a rule with two premise steps is called a *dual rule*. Thus, $\mathrm{exitA}[\ ]$ and $\mathrm{prefA}[a, P]$ are axioms; $\mathrm{disL1}[P, Q]$, $\mathrm{disL2}[P, Q]$, $\mathrm{parL}[P, Q]$, $\mathrm{disR}[P, Q]$, and $\mathrm{parR}[P, A, Q]$ are singular rules, with the first three examples of left rules and the last two examples of right rules; and $\mathrm{parD}[P, A, Q]$ is a dual rule.

Steps of a LOTOS process are represented as ordered pairs. However, in general steps may have some other structure, dependent on the semantic basis that underlies the specific notation. When referring to a generic SOS, therefore, we use S to denote the domain from which steps are drawn, and we distinguish elements of S using subscripts, i.e., $S_i, S_j \in S$. Moreover, a generic step assertion is written $OP[\overline{X}] \longrightarrow S_i$; we regard the LOTOS step assertion $P \xrightarrow{a} P'$ as an alternate way of writing $P \longrightarrow (a, P')$.

Amalia currently supports four types of rules (Fig. 1). In these rule types, $C_{opA[\overline{X}]}(\ )$, $C_{opL[\overline{X}]}(S_l)$, $C_{opR[\overline{X}]}(S_r)$, and $C_{opD[\overline{X}]}(S_l, S_r)$ are Boolean-valued expressions; $F_{opA[\overline{X}]}(\ )$, $F_{opL[\overline{X}]}(S_l)$, $F_{opR[\overline{X}]}(S_r)$, and $F_{opD[\overline{X}]}(S_l, S_r)$ are step-valued expressions; $\overline{X}$ is the set of arguments to the subject expression; $S_l, S_r \in S$ are premise steps; $X_l \in \overline{X}$ is the left operand of the subject expression; and $X_r \in \overline{X}$ is the right operand. We use the constant Boolean-valued expression *True* in cases where a rule has no side-condition.

For example, to codify the rules from Table 1 in this form we define:

$$
\begin{aligned}
C_{\mathrm{exitA}[\ ]}(\,) &\ \widehat{=}\ True \\
F_{exitA[\ ]}(\,) &\ \widehat{=}\ (\delta,\ STOP)
\end{aligned}
$$

$$
\begin{aligned}
C_{\mathrm{prefA}[a,\,P]}(\,) &\ \widehat{=}\ True \\
F_{\mathrm{prefA}[a,\,P]}(\,) &\ \widehat{=}\ (a, P)
\end{aligned}
$$

$$
\begin{aligned}
C_{\mathrm{disL1}[P,\,Q]}((a,\ P')) &\ \widehat{=}\ (a \neq \delta) \\
F_{\mathrm{disL1}[P,\,Q]}((a,\ P')) &\ \widehat{=}\ (a,\ P' \ [> Q)
\end{aligned}
$$

$$
\begin{aligned}
C_{\mathrm{disL2}[P,\,Q]}((a,\ P')) &\ \widehat{=}\ (a = \delta) \\
F_{\mathrm{disL2}[P,\,Q]}((a,\ P')) &\ \widehat{=}\ (a,\ P')
\end{aligned}
$$

$$
\begin{aligned}
C_{\mathrm{disR}[P,\,Q]}((a,\ Q')) &\ \widehat{=}\ True \\
F_{\mathrm{disR}[P,\,Q]}((a,\ Q')) &\ \widehat{=}\ (a,\ Q')
\end{aligned}
$$

$$
\begin{aligned}
C_{\mathrm{parL}[P,\,A,\,Q]}((a,\ P')) &\ \widehat{=}\ (a \notin A \cup \{\delta\}) \\
F_{\mathrm{parL}[P,\,A,\,Q]}((a,\ P')) &\ \widehat{=}\ (a,\ P' \mid A \mid Q)
\end{aligned}
$$

$$
\begin{aligned}
C_{\mathrm{parR}[P,\,A,\,Q]}((a,\ Q')) &\ \widehat{=}\ (a \notin A \cup \{\delta\}) \\
F_{\mathrm{parR}[P,\,A,\,Q]}((a,\ Q')) &\ \widehat{=}\ (a,\ P \mid A \mid Q')
\end{aligned}
$$

$$
\begin{aligned}
C_{\mathrm{parD}[P,\,A,\,Q]}((a,\ P'),\ (b,\ Q')) &\ \widehat{=}\ (a \in A \cup \{\delta\}) \wedge (a = b) \\
F_{\mathrm{parD}[P,\,A,\,Q]}((a,\ P'),\ (b,\ Q')) &\ \widehat{=}\ (a,\ P' \mid A \mid Q').
\end{aligned}
$$

Amalia imposes one additional requirement on the rules in an SOS. When an operator appears in the subject of multiple rules, the rules must be *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, $\mathrm{disL1}[P, Q]$ and $\mathrm{disR}[P, Q]$ are disjoint because they involve different operands, whereas $\mathrm{disL1}[P, Q]$ and $\mathrm{disL2}[P, Q]$ are disjoint because their side-conditions are contradictory.[4]

2.3 Internal Representations Supported by Amalia

A major goal of Amalia is to enable integration of analysis capability into larger environments [1]. For this reason, Amalia must tailor a step analyzer to operate on a particular internal representation. To satisfy this requirement, we design step analyzers to operate over ASTs in the metanotation defined in Section 2.2. We then use a code generator to specialize this meta design to the details of a particular internal representation. These details are conveniently described by a *class model*.

A class model is a design abstraction that relates a collection of classes by *generalization* (or "kind-of") and

4. A different formulation of inference graphs would allow us to relax this assumption, but at a performance penalty for the "normal" case.

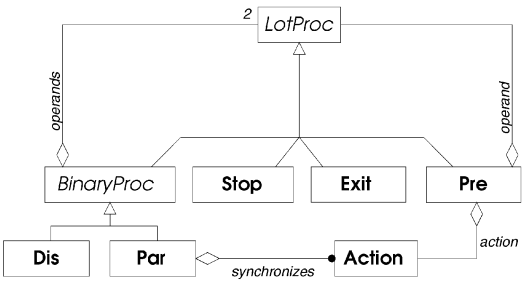


Fig. 2. Class model for the Lotos subset.

aggregation (or "part-of") associations. When used to specify the abstract syntax of a notation, a class model contains: a concrete class for each operator in the notation, aggregation associations that relate composite classes to classes that represent their parts, and abstract classes that factor and clarify aggregation associations and provide a most general class. Fig. 2 presents a class model that articulates the abstract syntax of our LOTOS subset in terms of classes and operations of a specific internal representation of LOTOS specifications. LOTOS processes are generalized by the abstract class `LotProc`. Concrete classes—`Exit`, `Stop`, `Pre`, `Dis`, and `Par`—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 an *accessor operation*, with which to retrieve any parts (i.e., references to operand ASTs or other syntactic attributes). For brevity, we elide these operations in Fig. 2, replacing them with the more concise aggregation associations. For example, the aggregations `action` and `operand` show that the constructor for `Pre` takes two arguments, one designating an `Action` AST and the other a `LotProc` AST. Constructors for binary operators—`Dis` and `Par`—take two `LotProc` arguments, and the constructor for `Par` takes a third argument, which designates a set of synchronization actions.

We treat a class model as a refinement of our metanotation as follows: First, we treat OP as a generic operator class, i.e., a concrete subclass of the most general AST class, and $OP[\overline{X}]$ as an object of class OP instantiated with the (possibly empty) list of arguments $\overline{X}$ that are passed to the constructor of the concrete class associated with OP. Second, commensurate with our metanotation, we distinguish operand arguments to a class' constructor from nonoperand arguments. Here, an operand $X_i \in \overline{X}$ must be an instance of the most general AST type. For example, an instance of type `Pre` has a single operand, even though its constructor takes two arguments (the `Action` argument is not an operand). 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 (Section 4, Fig. 6, top).

2.4 Implementing a Step Analyzer as a Visitor

Refining general AST accessor and constructor operations into operations over a specific internal form addresses only part of the integration concern. In addition, the analysis function itself must not impose constraints on the design of the internal representation, as such constraints might conflict with decisions already made. To address this concern, step analyzers implement the *visitor pattern* [4], which allows an operation over a linked object structure, such as an AST, to be separated from the design of the objects being traversed and encapsulated into a single class, called a *visitor class*.

In general, the visitor pattern accommodates algorithms that traverse a composite object structure, dispatch a function on an object in the structure (based on the class of the object) when the object is traversed, and exploit object orientation to collect information derived from the objects during the traversal. This second property—dispatching a function on an object based on the class of the object—illustrates that the operation implemented by a given visitor is polymorphic over its argument. Moreover, because a visitor class provides a `visitOp` method for each AST operator `Op`, the visitor class can be viewed as implementing a function that is designed by cases. Thus, the visitor pattern affords a degree of transparency for implementing inductive, polymorphic functions that are defined by cases. In Section 3.2, we formalize the derivation of steps for an AST as a polymorphic function of the AST and, indeed, most formally defined semantic functions are of this form. Thus, many interesting analyses can be implemented transparently using visitors, and designs based on this pattern are amenable to integration.

A visitor class `ASTVisitor` provides a `visitOp` operation for each distinct AST operator `Op`, and each such operation expects to be invoked with a single parameter of type `Op`. Furthermore, each AST-operator class `Op` provides an operation called `Accept`, which takes a single parameter of type `ASTVisitor`. To traverse a particular AST, you must allocate an object of class `ASTVisitor` and then invoke the `Accept` operation of the root of the AST, passing the `ASTVisitor` object as a parameter. Each AST class `Op` defines an `Accept` method that invokes the appropriate `visitOp` operation on its (`ASTVisitor`) parameter, passing itself (i.e., the AST node) as a parameter. This "double dispatch" operation is the key to decoupling the design of functions that operate by traversing an AST from the design of the AST classes themselves.

2.5 Amalia Tool Suite

Amalia combines principles from the GenVoca model of component-based application generators [5] with our results in visitor-based implementation of formal analyses to produce analyzers for formal notations. The step analyzer is a critical component to a variety of formal analysis capabilities, such as model checking, simulation, and test verification using oracles. These analysis capabilities can be designed as GenVoca components that can be instantiated with a particular step analyzer to work for a particular formal language whose AST is implemented using a particular internal representation [1]. Moreover, step analyzers are not written by programmers, but are generated automatically from SOS rules, expressed in the metanotation of Fig. 1 and from a description of how operators and operations in the SOS rules map to classes and navigation operations of a particular internal representation.

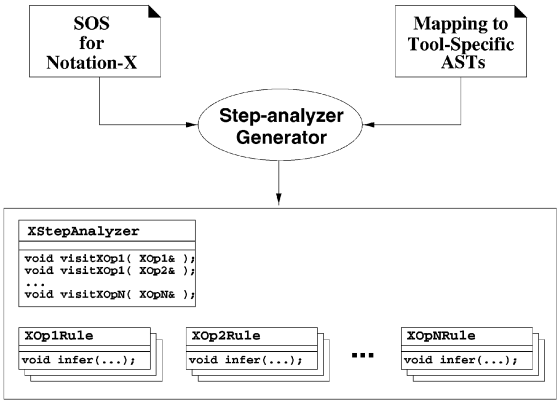


Fig. 3. Amalia step-analyzer generator.

Fig. 3 depicts the Amalia tools that are concerned with generating step analyzers. In this diagram, documents (crimped-page icons) represent input files that a human designer must construct and the boxes written in the UML-like notation denote classes that are generated by the step-analyzer generator. The `XStepAnalyzer` class, which is used by higher-level Amalia components to compute the steps of a given AST, is implemented using the visitor pattern. The other generated classes, called *rule classes*, are described later in the paper. Essentially, one rule class is generated for each axiom or inference rule in the SOS of a notation. In the diagram, we partition the rule classes into groups based on the major operator to which they apply. The rule classes are used by the `XStepAnalyzer` to build and execute an inference graph for an AST as the AST is being traversed. The remainder of this paper is concerned with this process.

3 Denotational Model of the Step Relation

As discussed previously, we require that a step analyzer is implemented as a visitor, which performs an analysis in a single, top-down traversal of an AST. Moreover, for assurance, the implementation of the step analyzer must be transparent—that is, we must be able to rigorously show how it implements the step relation defined by an SOS. The previous formalization of the steps of a process as those steps that are produced by multiple derivations implies a bottom-up traversal of an AST during which steps of each child AST node are buffered for use in computing the steps of its parent. This simple bottom-up strategy has two drawbacks. First, it requires buffering all steps of a child AST node and iterating over these buffers to calculate the steps of the parent AST node. We show in the sequel that buffering of steps is needed at a node with operator *OP* only if there are one or more dual rules for *OP* and if its left operand produces steps that do not satisfy the side-conditions of any left rules for *OP*. Additionally, because a purely bottom-up strategy cannot take context into account, it may result in calculation of steps for nodes that cannot contribute to steps of some ancestors. In LOTOS, for example, the steps of an AST containing a prefix node do not depend on the steps of the prefix operator's operand; thus, there is no reason to compute the steps of the operand AST. A second drawback of the bottom-up strategy is that buffering steps and iterating through these buffers obstructs transparency since the buffers have no direct counterpart in the formalization of the step relation defined by an SOS.

To overcome these problems, we introduce a denotational model of the step relation defined by an SOS. Specifically, we use an SOS to define a *meaning function* $\mathcal{M}$ that maps an AST $OP[\overline{X}]$ to a sequence of steps $\mathcal{M}[\![OP[\overline{X}]]\!]$ containing exactly those steps that can be derived from $OP[\overline{X}]$.[5] Briefly, the meaning function is obtained by treating SOS rules as defining functions over steps, which can be aggregated into larger functions on the basis of the operator in the rule's subject and the operand(s) in the rule's premise assertion(s) (Section 3.1). The aggregated rule functions permit us to define $\mathcal{M}$ by induction on the structure of an AST (Section 3.2). We require an inductive definition in order to justify certain optimizations that are performed when generating inference graphs (Section 3.3) and to show that $\mathcal{M}$ is correct—that is, that $\mathcal{M}[\![OP[\overline{X}]]\!]$ contains precisely those steps derivable from $OP[\overline{X}]$ according to the SOS (Section 3.4). Table 2 summarizes notation needed for the definitions in this section.

3.1 Rule Functions, Initiator Functions, and Transfer Functions

Rule functions are defined using the schemata for representing SOS rules (Fig. 1). We treat an axiom $\mathrm{opA}[\overline{X}]$ as defining a (0-ary) rule function $i_{\mathrm{opA}[\overline{X}]}$, a left (respectively, right) rule $\mathrm{opL}[\overline{X}]$ (respectively, $\mathrm{opR}[\overline{X}]$) as defining a unary rule function $t_{\mathrm{opL}[\overline{X}]}$ (respectively, $t_{\mathrm{opR}[\overline{X}]}$), and a dual rule $\mathrm{opD}[\overline{X}]$ as defining a binary rule function $t_{\mathrm{opD}[\overline{X}]}$:

$$i_{\mathrm{opA}[\overline{X}]}(\ ) \mathrel{\widehat{=}} \begin{cases} F_{\mathrm{opA}[\overline{X}]}(\ ) & \text{if } C_{\mathrm{opA}[\overline{X}]}(\ ) \\ \bot & \text{otherwise,} \end{cases}$$

$$t_{\mathrm{opL}[\overline{X}]}(S_i) \mathrel{\widehat{=}} \begin{cases} F_{\mathrm{opL}[\overline{X}]}(S_i) & \text{if } C_{\mathrm{opL}[\overline{X}]}(S_i) \\ \bot & \text{otherwise,} \end{cases}$$

$$t_{\mathrm{opR}[\overline{X}]}(S_i) \mathrel{\widehat{=}} \begin{cases} F_{\mathrm{opR}[\overline{X}]}(S_i) & \text{if } C_{\mathrm{opR}[\overline{X}]}(S_i) \\ \bot & \text{otherwise,} \end{cases}$$

$$t_{\mathrm{opD}[\overline{X}]}(S_i, S_j) \mathrel{\widehat{=}} \begin{cases} F_{\mathrm{opD}[\overline{X}]}(S_i, S_j) & \text{if } C_{\mathrm{opD}[\overline{X}]}(S_i, S_j) \\ \bot & \text{otherwise.} \end{cases}$$

The rule functions produced in this manner from the rules that apply to the major operator *OP* in an AST $OP[\overline{X}]$ are then summed, based on the operand steps that they work on, to yield four functions.

Definition 1. *Given an AST* $OP[\overline{X}]$*:*

- *The* initiator function $i_{\mathrm{op}[\overline{X}]}$ *aggregates the (0-ary) rule functions produced from the axioms,* $\mathrm{opA}[\overline{X}]$*:*

$$i_{\mathrm{op}[\overline{X}]} \mathrel{\widehat{=}} \sum_{\mathrm{opA}[\overline{X}]} i_{\mathrm{opA}[\overline{X}].}$$

5. An inference graph computes sets rather than sequences, but we model the sets as sequences to simplify the proofs. The ordering of steps in these sets has no bearing on correctness.

TABLE 2
Summary of Notation

Notation	Meaning / definition	Assumptions
S	set of steps	
$\perp$	undefined value or undefined function	
$S^{\perp}$	$S \cup \{\perp\}$	$\perp \notin S$
$\mathrm{dom(f)}$	$\{x \in X \mid f(x) \neq \perp\}$	$f: X \to S^{\perp}$
$S_i + \perp,\ \perp + S_i$	$\perp$	$S_i \in S$
$(f+g)(x)$	$f(x)$, if $x \in \mathrm{dom(f)}$; $g(x)$, otherwise	$f, g: X \to S^{\perp}$; $x \in X$;
S^*	set of sequences over S	
$\langle s(0), \ldots, s(n-1)\rangle$	sequence of length n	$n \geq 0$
$\lvert s\rvert$	length of s	$s \in S^*$
$S_j :: s$	s, if $S_j = \perp$; insert S_j at the front of s, otherwise	$S_j \in S^{\perp}, s \in S^*$
s'	tail of s, or $\langle s(1), \ldots s(n)\rangle$	$s \in S^*,\ s \neq \langle\rangle$, $n = \lvert s\rvert - 1$
$s_1 \frown s_2$	concatenation of s_1 and s_2	$s_1, s_2 \in S^*$
$s \restriction T$	projection of s on T	$T \subseteq S$
$\mathrm{pair}(s, S_j)$	$\langle\rangle$, if $s = \langle\rangle$; $(s(0), S_j) :: \mathrm{pair}(s', S_j)$, otherwise	$S_j \in S, s \in S^*$
$\mathrm{prod}(s_i, s_j)$	$\langle\rangle$, if $s = \langle\rangle$; $\mathrm{pair}(s_i, s_j(0)) \frown \mathrm{prod}(s_i, s_j')$, otherwise	$s_i, s_j \in S^*$
$\mathrm{map}(f, s)$	$\langle\rangle$, if $s = \langle\rangle$; $f(s(0)) :: \mathrm{map}(f, s')$, otherwise	$s \in S^*$

- *The* left transfer function $t^l_{\mathrm{op}[\overline{X}]}$ *aggregates the rule functions produced from the left rules,* $\mathrm{opL}[\overline{X}]$*:*

$$t^l_{\mathrm{op}[\overline{X}]} \ \widehat{=} \ \sum_{\mathrm{opL}[\overline{X}]} t_{\mathrm{opL}[\overline{X}]}.$$

- *The* right transfer function $t^r_{\mathrm{op}[\overline{X}]}$ *aggregates the rule functions produced from the right rules,* $\mathrm{opR}[\overline{X}]$*:*

$$t^r_{\mathrm{op}[\overline{X}]} \ \widehat{=} \ \sum_{\mathrm{opR}[\overline{X}]} t_{\mathrm{opR}[\overline{X}]}.$$

- *The* dual transfer function $t^d_{\mathrm{op}[\overline{X}]}$ *aggregates the rule functions produced from the dual rules,* $\mathrm{opD}[\overline{X}]$*:*

$$t^d_{\mathrm{op}[\overline{X}]} \ \widehat{=} \ \sum_{\mathrm{opD}[\overline{X}]} t_{\mathrm{opD}[\overline{X}]}.$$

The initiator functions and transfer functions needed for defining $\mathcal{M}[\![PROC]\!]$ are shown in Fig. 4. The fourth and sixth transfer functions are examples of summing: t^l_2 is the sum of the rule functions for $\mathrm{disL1}[PING, CTRLC]$ and $\mathrm{disL2}[PING, CTRLC]$, and t^l_3 is similar. For brevity, we use the indicated abbreviations (t^d_1, t^l_1, etc.) in subsequent examples.

3.2 The Meaning Function

To motivate the definition of $\mathcal{M}$, consider an AST $OP[\overline{X}]$ for a binary operator OP. A step analyzer first visits the left operand and then visits the right operand, generating the steps that the respective operands can perform during these visits. This fact suggests that $\mathcal{M}$ can be defined in terms of auxiliary functions, which are given the sequences of left-operand steps and right-operand steps as inputs, and that these auxiliary functions should use first the left-operand steps and then the right-operand steps, from left to right, in a demand driven fashion. We define two such auxiliary functions. The first of these functions (Definition 2) applies the left transfer function to transform left-operand steps that satisfy the side condition of a left rule into steps of $OP[\overline{X}]$. It takes a special parameter (x) which contains left-operand steps that do not satisfy the side condition of any left rule. These steps will used by the second auxiliary function (Definition 3) in building pairs of left- and right-operand steps on which to apply the dual transfer function.

Definition 2. *The* recording function R *is defined for functions* $t_l, t_r : S^{\perp} \to S^{\perp}$ *and* $t_d : S^{\perp} \times S^{\perp} \to S^{\perp}$*, and for sequences* $x, s_l, s_r \in S^*$*:*[6]

$$R_{t_l,t_r,t_d}(x)(s_l, s_r) \ \widehat{=}$$
$$\begin{cases} P_{t_r,t_d}(x)(s_r) & \text{if } s_l = \langle\rangle; \\ t_l(s_l(0)) :: R_{t_l,t_r,t_d}(x)(s_l', s_r) & \text{if } s_l \neq \langle\rangle \text{ and } s_l(0) \in \mathrm{dom}(t_l); \\ R_{t_l,t_r,t_d}(x \frown \langle s_l(0)\rangle)(s_l', s_r) & \text{if } s_l \neq \langle\rangle \text{ and } s_l(0) \notin \mathrm{dom}(t_l). \end{cases}$$

Intuitively, s_l and s_r represent steps of the (respectively, left and right) operands, and t_l, t_r, and t_d represent (respectively, left, right, and dual) transfer functions. The parameter x keeps left-operand steps that do not satisfy the side conditions of any left rules. The second case of the definition applies when the first step of s_l is in the domain of t_l. In this case, a new step $t_l(s_l(0))$ is inserted into the output sequence. The third case applies when the first step of s_l is not in the domain of t_l. In this case, the first step of s_l is inserted into the sequence of left-operand steps that will be passed to the

6. That is, R is a function of six arguments. For better readability, we curry some of the arguments and we show the first three as subscripts. A similar comment applies to the next definition, for the playing function P.

$$t_1^d\colon\ t^d_{\mathrm{par}[PDCC,EDCC]}((a,P),(b,Q)) \ \widehat{=}\ \begin{cases} (a,\ P\ |[ctrlc]|\ Q) & \text{if } a=b=ctrlc \\ & \text{or } a=b=\delta \\ \bot & \text{otherwise} \end{cases}$$

$$t_1^l\colon\ t^l_{\mathrm{par}[PDCC,EDCC]}((a,P)) \ \widehat{=}\ \begin{cases} (a,\ P\ |[ctrlc]|\ EDCC) & \text{if } a\neq ctrlc \text{ and} \\ & \quad a\neq\delta \\ \bot & \text{otherwise} \end{cases}$$

$$t_1^r\colon\ t^r_{\mathrm{par}[PDCC,EDCC]}((a,P)) \ \widehat{=}\ \begin{cases} (a, PDCC\ |[ctrlc]|\ P) & \text{if } a\neq ctrlc \\ & \text{and } a\neq\delta \\ \bot & \text{otherwise} \end{cases}$$

$$t_2^l\colon\ t^l_{\mathrm{dis}[PING,CTRLC]}((a,P)) \ \widehat{=}\ \begin{cases} (a,P\ [>\ CTRLC) & \text{if } a\neq\delta \\ (a,P) & \text{otherwise} \end{cases}$$

$$t_2^r\colon\ t^r_{\mathrm{dis}[PING,CTRLC]}((a,P)) \ \widehat{=}\ (a,P)$$

$$t_3^l\colon\ t^l_{\mathrm{dis}[EXIT,CTRLC]}((a,P)) \ \widehat{=}\ \begin{cases} (a,P\ [>\ CTRLC) & \text{if } a\neq\delta \\ (a,P) & \text{otherwise} \end{cases}$$

$$t_3^r\colon\ t^r_{\mathrm{dis}[EXIT,CTRLC]}((a,P)) \ \widehat{=}\ (a,P)$$

$$i_4\colon\ i_{\mathrm{pre}[ping,EXIT]}(\,) \ \widehat{=}\ (ping,EXIT) \qquad i_5\colon\ i_{\mathrm{pre}[ctrlc,EXIT)]}(\,) \ \widehat{=}\ (ctrlc,EXIT)$$

$$i_6\colon\ i_{\mathrm{exit}[\,]}(\,) \ \widehat{=}\ (\delta,STOP) \qquad i_7\colon\ i_{\mathrm{pre}[ctrlc,EXIT)]}(\,) \ \widehat{=}\ (ctrlc,EXIT)$$

Fig. 4. Transfer functions (t_1^d, t_1^l, etc.) and initiatator functions (i_4, i_5, etc.).

second auxiliary function. The disjointness assumption ensures that only those left-operand steps that are not in the domain of t_l need to be passed to the second auxiliary function. Both of the latter two cases in the definition recurse on the tail of s_l. The first case is the basis case; it applies when the sequence of left-operand steps have been consumed. In this case, the second auxiliary function P is applied with the sequence of left-operand steps that do not satisfy the side conditions of any left rules and with the sequence of right-operand steps.

Definition 3. *The* playing function P *is defined for functions* $t_r : S^\bot \to S^\bot$ *and* $t_d : S^\bot \times S^\bot \to S^\bot$, *and for sequences* $x, s_r \in S^*$:

$$P_{t_r,t_d}(x)(s_r) \ \widehat{=}\ \begin{cases} \langle\,\rangle & \text{if } s_r = \langle\,\rangle; \\ t_r(s_r(0)) :: P_{t_r,t_d}(x)(s_r') & \\ & \text{if } s_r \neq \langle\,\rangle \text{ and } s_r(0) \in \mathrm{dom}(t_r); \\ \mathrm{map}(t_d, \mathrm{pair}(x, s_r(0))) \frown P_{t_r,t_d}(x)(s_r') & \\ & \text{if } s_r \neq \langle\,\rangle \text{ and } s_r(0) \notin \mathrm{dom}(t_r). \end{cases}$$

Informally, the second case of the definition applies when the first step in s_r is in the domain of t_r. In this case, a new step $t_r(s_r(0))$ is inserted into the output sequence. The third case applies when the first step of s_r is not in the domain of t_r. In this case, the first step of s_r is paired with the left-operand steps in x, and t_d is mapped over the resulting sequence of left-and right-operand pairs. Applying t_d to a pair in its domain produces a new step in the output sequence, whereas applying it to a pair outside of its domain has no effect on the output sequence.[7] Both of these latter two cases recurse on the tail of s_r. The basis (first) case applies when the sequence of right-operand steps has been consumed.

These definitions allow us to define the function $\mathcal{M}$.

Definition 4. *Let an AST* $OP[\overline{X}]$ *be given.*

$$\mathcal{M}[\![OP[\overline{X}]]\!] \ \widehat{=}\ i_{\mathrm{op}[\overline{X}]}(\,) :: R_{t^l_{\mathrm{op}[\overline{X}]},t^r_{\mathrm{op}[\overline{X}]},t^d_{\mathrm{op}[\overline{X}]}}(\langle\,\rangle)(s_l,s_r),$$

where $s_l \mathrel{\widehat{=}} \mathcal{M}[\![X_l]\!]$, *if* $\overline{X}$ *contains a left operand* $X_l \in \overline{X}$, *and* $s_l \mathrel{\widehat{=}} \langle\,\rangle$, *otherwise; and* $s_r \mathrel{\widehat{=}} \mathcal{M}[\![X_r]\!]$, *if* $\overline{X}$ *contains a right operand* $X_r \in \overline{X}$, *and* $s_r \mathrel{\widehat{=}} \langle\,\rangle$, *otherwise.*

For example, the meanings of several of the processes in (1) are computed (using the abbreviations from Fig. 4):

$$\begin{aligned} \mathcal{M}[\![CTRLC]\!] &= \langle(ctrlc,\ EXIT)\rangle \\ \mathcal{M}[\![EXIT]\!] &= \langle(\delta,\ STOP)\rangle \\ \mathcal{M}[\![EDCC]\!] &= R_{t_l^3,t_r^3,\bot}(\langle\,\rangle)(\langle(\delta,\ STOP)\rangle, \\ &\qquad \langle(ctrlc,\ EXIT)\rangle) \\ &= (\delta,\ STOP) :: \\ &\qquad P_{t_r^3,\bot}(\langle\,\rangle)(\langle(ctrlc,\ EXIT)\rangle) \\ &= \langle(\delta,\ STOP),\ (ctrlc,\ EXIT)\rangle. \end{aligned}$$

The denotational formulation of the process by which an SOS defines a step relation is well suited to a transparency argument for several reasons. First, it is succinct and mathematically precise. Second, it defines $\mathcal{M}$ in a demand-driven style, which admits efficient reification using a data flow structure. Third, it makes explicit the buffers needed to realize the semantics of binary operators with dual rules (using the parameter x): Only those left-operand steps that do not satisfy the side conditions of the left rules need to be buffered. Finally, it permits us to formally justify optimizations that a step analyzer performs.

7. Because $\bot$ is a left identity for insertion (Table 2).

3.3 Standard Simplifications

Definition 4 is a compromise between simplicity and transparency. For simplicity, we define initiator functions and left, right, and dual transfer functions for all operators, without regard to whether the operators have axioms, or left, right, or dual rules. When the SOS does not provide rules of a given type for an operator, we use the null function for the corresponding initiator or transfer function. For efficiency, however, a step analyzer does not implement degenerate initiator and transfer functions.

The following theorem justifies optimizations that a step analyzer performs when assembling an inference graph in the presence of degenerate functions.

Theorem 1. *Let an AST $OP[\overline{X}]$ be given. Then, we have as special cases:*

- *If $t^l_{\mathrm{op}[\overline{X}]} = \bot$, $t^r_{\mathrm{op}[\overline{X}]} = \bot$, $t^d_{\mathrm{op}[\overline{X}]} = \bot$, and $i_{\mathrm{op}[\overline{X}]} \neq \bot$, then $\mathcal{M}[\![OP[\overline{X}]]\!] = \langle i_{\mathrm{op}[\overline{X}]}(\ )\rangle$*
- *If $i_{\mathrm{op}[\overline{X}]} = \bot$, $t^r_{\mathrm{op}[\overline{X}]} = \bot$, and $t^d_{\mathrm{op}[\overline{X}]} = \bot$, then $\mathcal{M}[\![OP[\overline{X}]]\!] = \mathrm{map}(t^l_{\mathrm{op}[\overline{X}]}, s_l)$.*
- *If $i_{\mathrm{op}[\overline{X}]} = \bot$ and $t^d_{\mathrm{op}[\overline{X}]} = \bot$, then $\mathcal{M}[\![OP[\overline{X}]]\!] = \mathrm{map}(t^l_{\mathrm{op}[\overline{X}]}, s_l)^\frown \mathrm{map}(t^r_{\mathrm{op}[\overline{X}]}, s_r)$.*
- *If $i_{\mathrm{op}[\overline{X}]} = \bot, t^l_{\mathrm{op}[\overline{X}]} = \bot$, and $t^r_{\mathrm{op}[\overline{X}]} = \bot$, then $\mathcal{M}[\![OP[\overline{X}]]\!] = \mathrm{map}(t^d_{\mathrm{op}[\overline{X}]}, \mathrm{prod}(s_l, s_r))$.*

The proof of Theorem 1 requires several intermediate claims, which we do not include because of space limitations. Detailed statements and proofs of these claims can be found in [12], where a proof of Theorem 1 also appears.

3.4 Correctness

We define the meaning function $\mathcal{M}$ to use in arguing that an Amalia-generated step analyzer is correct. For this argument to be valid, we must show that $\mathcal{M}$ correctly models the step relation defined by a set of SOS rules. In the sequel, we prefix a step assertion with "$\vdash$ " to indicate that the assertion is provable by a derivation.

Theorem 2 (Correctness). *Let an AST $OP[\overline{X}]$ be given and let S_j denote a step. Then, $S_j \in \mathcal{M}[\![OP[\overline{X}]]\!]$ if and only if $\vdash OP[\overline{X}] \longrightarrow S_j$.*

To prove soundness, we assume that $S_j \in \mathcal{M}[\![OP[\overline{X}]]\!]$. We then show, by induction on the number of nodes in $OP[\overline{X}]$, that $\vdash OP[\overline{X}] \longrightarrow S_j$. Conversely, to prove completeness, we assume $\vdash OP[\overline{X}] \longrightarrow S_j$. We then show, by induction on the length of the derivation of $OP[\overline{X}] \longrightarrow S_j$, that $S_j \in \mathcal{M}[\![OP[\overline{X}]]\!]$. Details are not included for lack of space. They can be found in [12].

4 Inference Graphs

A step analyzer computes the step-analysis function $\mathcal{M}$ as a side effect of a top-down traversal of an AST. Moreover, it does so in a way that is amenable to integration with different AST representations, is transparent with respect to the formal definition of $\mathcal{M}$ so as to enable a high degree of assurance, and does not require creation and manipulation of unnecessary auxiliary storage structures. Satisfaction of these design goals owes to a computational structure, called an *inference graph*. Whereas an inference graph is a program object and not a mathematical abstraction, we use mathematical terminology, such as "graphs" and "nodes" because it is simpler and more suggestive than programming terminology. An inference graph reifies a data-driven interpretation of $\mathcal{M}$ and admits several optimizations. We now describe this design in detail. For reference throughout this section, Fig. 5 depicts the relationship between the formal abstractions and the computational structures that reify them.

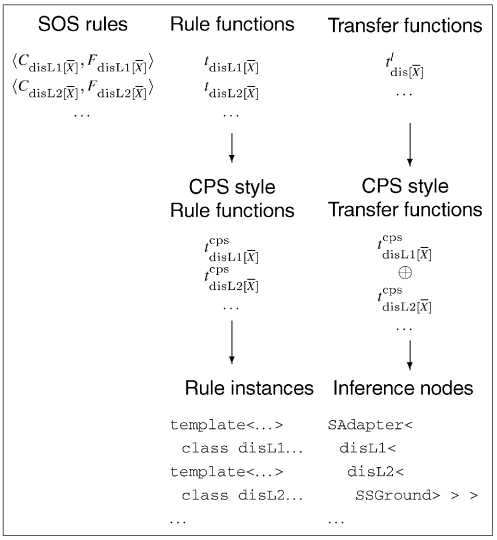


Fig. 5. Reification of mathematical abstractions into computational structures.

An inference graph is a linked assembly of objects that collaborate to compute the steps of an AST. The graph is executed in a data-driven manner, whereby distinguished *initiator nodes* are *fired* to produce basis steps that are then forwarded to *transfer nodes*, which then fire to compute derived steps, which are then forwarded to other nodes, and so on. A distinguished output node collects the steps of the given AST; when fired, it merely adds input steps to a buffer and does not forward them. Eventually, the propagation of steps terminates, at which point the computation is complete. By computing steps in this data-driven style, inference graphs avoid the need to create collections of steps at every node in the graph.

As suggested by their names, initiator nodes reify initiator functions and transfer nodes reify transfer functions. An initiator node provides a firing interface that is invoked without parameters and also an *exit out-flow* along which to propagate the step that it computes. Likewise, a transfer node provides a firing interface and an exit out-flow, on which to propagate the steps that it computes; however, unlike an initiator node, it must be invoked with a (single) step parameter and it needs an additional out-flow. Thus, a transfer node provides an *in-flow*, along which it receives steps. Because it reifies a transfer function, which might return $\bot$ on some inputs, it also provides a *shunt out-flow*, along which to forward unhandled inputs. An inference graph is connected to form a directed acyclic graph whose source nodes are initiator nodes and whose

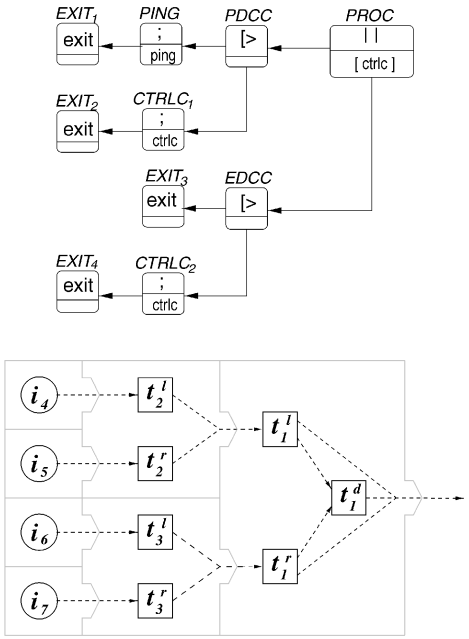


Fig. 6. AST of example specification (top) and its inference graph (bottom).

edges represent connections between the out-flow of one node and the in-flow of another.

Fig. 6 illustrates these concepts. At the top, it depicts the AST of the running example specification. Roundtangles denote AST objects and arrows denote pointers from operator objects to their operand AST(s). Each object is named for reference purposes (above the roundtangle). Some objects (e.g., ";" and "$\|$ ") are attributed with additional syntax (e.g., "ping," "ctrlc," and "[ctrlc]"), which indicates the nonoperand arguments. The bottom of Fig. 6 shows the inference graph for this AST. Initiator nodes are shown as circles, transfer nodes as squares, and communication pathways as dashed arrows. We label each initiator node with the initiator function that is applied when the node is invoked (i_4, i_5, etc.). Similarly, we label each transfer node with the reified transfer function (t_1^l, t_1^r, etc.). These initiator and transfer functions are defined in Fig. 4.

It is useful to view the graph as a hierarchical assembly of *boxes of inference nodes*. Thus viewed, the inference graph constructed for AST $OP[\overline{X}]$ comprises an inference subgraph for each operand $X_i \in \overline{X}$ and a box that uses the steps computed by these inference subgraphs to compute the steps of $OP[\overline{X}]$. A box may have up to two in-flows—a left in-flow, on which to receive steps of the left inference subgraph; a right in-flow, on which to receive steps of the right inference subgraph; and one out-flow, on which it sends steps that it computes. The bottom of Fig. 6 also shows the box structure of the inference graph with shaded lines demarcating boxes, arrow-shaped cut-outs signifying in-flows, and arrow-shaped push-outs signifying out-flows. Communication pathways that go from a left or right transfer node to a dual node are used for shunting premise steps to the dual node. All such shunting pathways are internal to a box.

Transfer nodes are classified as left, right, or dual, according to the type of transfer function that they reify. A left (respectively, right) transfer node fires when its box receives a step on its left (respectively, right) in-flow. If this step is in the domain of the reified transfer function, then the left (respectively, right) node computes and then propagates a new derived step to the exit out-flow. On the other hand, if the step is not in the domain of the reified transfer function, then the node shunts the step to the dual transfer node, if there is one, or else discards the step. The firing logic for a dual node is a bit more complicated, as the node must be fired with a single step, but dual transfer functions require two step parameters. To understand this logic, requires understanding the *incremental* execution of an inference graph, which we now discuss.

4.1 Incremental Assembly/Execution of Inference Graphs

A step analyzer does not sequentially generate and then execute an inference graph. Rather, it dynamically assembles and executes the graph on-the-fly, as it visits an AST. Incremental assembly and execution are possible because an operand's inference graph is no longer needed once the visit of that operand returns. This strategy yields benefits not only in terms of storage efficiency, but also in terms of transparency with the formal definition of $\mathcal{M}$.

A step analyzer is implemented using a visitor class. When instantiated, a step analyzer receives an in-flow, called a *target*, in which to connect the exit out-flow of the inference graph that it creates. For each operator OP, it provides a `visitOP` method that expects to be passed an AST whose root has type OP. A visit method creates a box for the root of the input AST and instantiates new step analyzers to assemble inference subgraphs from the operands of the root operator, if needed. It initializes the step analyzers for the operands with the in-flow of the box just created. This recursive elaboration of the inference graph as a hierarchy of boxes continues until we visit a sub-AST that has no left, right, or dual rules. Each such sub-AST corresponds to a box that contains only a flow initiator.

More precisely, a step analyzer is initialized with a target T to use as an out-flow. When its `visitOP` method is invoked to visit an expression $OP[\overline{X}]$, it assembles a box B for $OP[\overline{X}]$, using T as the out-flow for B. Then, for each operand $X_i \in \overline{X}$, starting with the leftmost operand, if B has an in-flow I that operates on steps of X_i, the step analyzer for $OP[\overline{X}]$ initializes a new step analyzer with target I and applies the new analyzer to X_i to generate an inference graph for X_i that connects to B on I. Incremental execution occurs whenever an initiator node is allocated and linked into the graph. At this point, the initiator node is fired, causing a flow of steps, which may result in steps flowing out of the output of the entire graph.

To support incremental execution requires some special handling in the case of dual transfer nodes. A dual transfer node requires a step from each of two inference subgraphs, but as we mentioned earlier, a dual node must be fired with a single step at a time. Because AST traversals are performed in order, a dual node will receive all the premise steps for the left operand before it receives any premise steps for the right operand. Thus, a dual node operates in one of two modes: It is in *recording mode* during traversal of

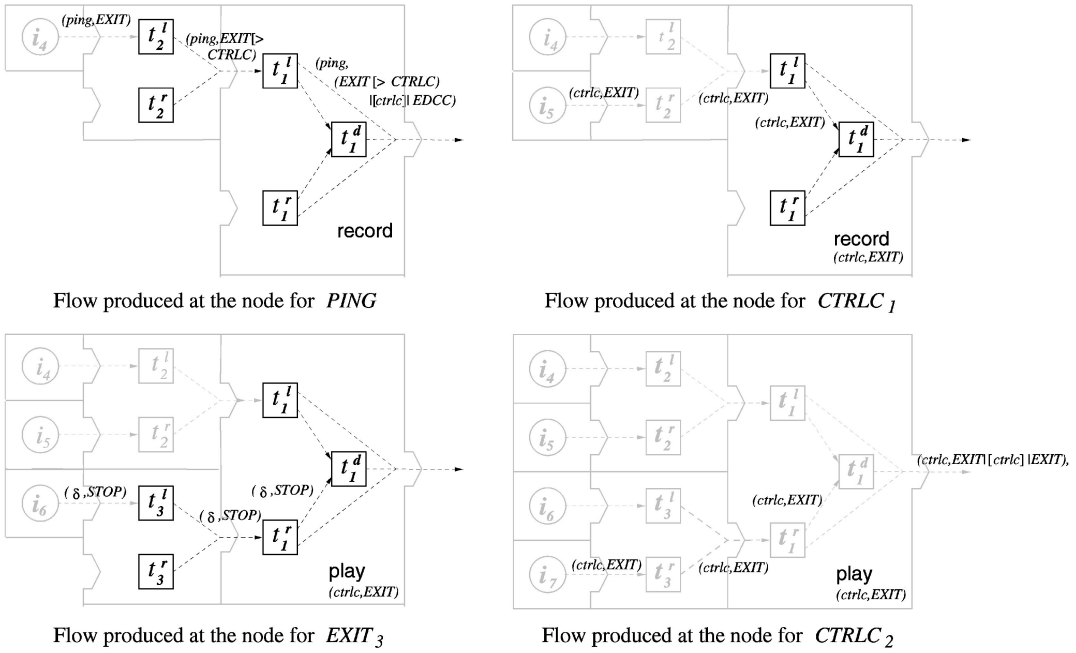


Fig. 7. Assembly and execution of an inference graph.

the left operand AST, after which it transitions into *playing mode* for the traversal of the right operand AST. In recording mode, it buffers all incoming (left operand) steps and, in playing mode, it pairs incoming (right operand) steps with the buffered (left operand) steps and transforms the paired premise steps into conclusion steps. This behavior is designed to be transparent with respect to $\mathcal{M}$, which is defined in terms of the playing and recording functions (Definitions 2 and 3); the logic for firing dual nodes reifies the playing/recording distinction.

Fig. 7 illustrates these concepts by showing four different snapshots of the inference graph that is constructed for the AST of Fig. 6. When a box contains a dual transfer node, we annotate the box with `record` or `play` to indicate the mode of the dual node, and we show any steps that are buffered in the node immediately below the mode annotation. The four configurations of the inference graph illustrate how the graph is assembled and executed as a step analyzer traverses the AST. The top-left configuration illustrates the state of the inference graph once the step analyzer has reached the AST object labeled $PING$, by which point boxes have been created for objects $PROC$, $PDCC$, and $PING$. The box for $PING$ contains only an initiator node, which is fired as soon as it is created, producing a flow of steps. This flow propagates from box to box, eventually outputting a step of $PROC$. In the figure, we illustrate the propagation of steps by labeling the out-flows of the nodes that are fired with the steps that they produce.

The next initiator node is created and fired when the analyzer reaches AST node $CTRLC_1$. This flow shunts a step to the dual node, where it is buffered (top right). On completing the traversal of the AST node $PDCC$, the analyzer signals the dual node to change from recording mode to playing mode. The third flow is then initiated when the analyzer reaches AST node $EXIT_3$. This flow shunts a premise step to the dual node, which pairs the premise step with the step it has buffered. However, as t_1^d is not defined on this pair, the pair is discarded (bottom left). The fourth and final flow is initiated when the analyzer reaches AST node $CTRLC_2$. This flow propagates the second behavior of $PROC$ to the graph's out-flow (bottom right). As a practical matter, the analyzer deallocates the box for an AST node when the traversal of the AST node is complete, which we show by greying out the box and its contents.

This design incorporates several optimizations and it does so without violating transparency. First, the step analyzer visits an operand of an AST node only if some inference node in the box that it creates requires steps from that operand. The box created for an instance of the LOTOS prefix operator is a good example of this behavior. An instance of the prefix operator has an operand, but the semantics of prefix are defined entirely by axioms; thus, there is no need to compute the steps of the operand, nor even to traverse the AST of the operand. Second, at any point during traversal of an AST, the inference graph contains boxes for only those AST operands that are currently being visited. The storage used by functions called for nodes that are being visited will be allocated on the run-time stack, which means that the lifetime of the inference nodes for a given AST node is bounded by the lifetime of the function invocation for that node. Thus, while the inference graph is a dynamic linked structure, the inference nodes can be allocated and reclaimed using automatic storage by declaring them as local variables in the step analyzer(s). Finally, the design does not call for needless buffering of steps at each AST node. Only those left-operand steps that must await pairing with right-operand steps are buffered. Moreover, these buffers are made explicit in the definition of $\mathcal{M}$ enhancing transparency, as discussed in the sequel.

5 TRANSPARENCY

For assurance, we must demonstrate the fidelity of the mathematical model developed in Section 3 against the implementation described in Section 4. Unfortunately, an inference graph is implemented using advanced object-oriented design constructs (e.g., mixin classes [6] and the visitor pattern [4]) for which mature proof techniques have yet to be developed. Thus, it would be very difficult to demonstrate fidelity by means of a formal proof of correctness. We opt instead to demonstrate fidelity by inspection, exploiting the transparency that we carefully engineered into our implementation. Specifically, we argue:

1. An inference node behaves as specified by the corresponding initiator or transfer function.
2. The nodes in a box are interconnected as prescribed by our formalization of inference graphs.
3. The collaboration among these nodes is as prescribed by the playing and recording functions.

We designed inference nodes to be composite assemblies of more primitive objects that reify individual rule functions. These *rule instances* compose so that their (daisy-chained) execution preserves the semantics of the sums of the corresponding rule functions (Section 5.1). An inference node is then an assembly of these rule instances sandwiched between special adapter objects, which manage the flow of steps through the graph (Section 5.2). Inference nodes defined in this way are transparent implementations of the initiator and transfer functions defined in Section 3; however, the solution incurs some indirection, a result of the shunting logic and the introduction of adapters. We demonstrate how to remove this indirection—essentially collapsing a composite assembly of adapters, terminators, and rule instances into a single object—using ideas from advanced OO design (Section 5.3). Finally, we demonstrate how these (optimized) inference nodes are linked together into a graph by a step analyzer and how they collaborate as prescribed by the playing and recording functions (Section 5.4). As an example, we argue correctness for part of the analyzer generated for LOTOS.

5.1 Reifying Rule Functions and Sums

A rule instance is an object that reifies a rule function by providing an `infer` operation, which can be invoked with zero, one, or two step parameters, as appropriate to the arity of the rule function. Rule instances play roles in the inference graph. We say that a rule instance *receives* a step (or pair of steps) when its `infer` operation is invoked. A rule instance R also contains two references to other objects. The first reference, called `target`, points to a transfer node, which is fired with the step that is computed when R receives a premise step (or pair of premise steps) that satisfies its side condition. The second reference, called `shunt`, points to another rule instance with a plug-compatible `infer` operation. When R receives a step that does not satisfy its side condition, it forwards this step to the rule instance referenced by `shunt`.

Whereas a rule instance is an object in a data-flow graph, a rule function is a mapping over $S^{\perp}$. To demonstrate that a rule instance reifies a rule function, we first rewrite this function in a form called continuation-passing style (CPS) [13]. Without loss of generality, consider a unary function $f : S^{\perp} \to S^{\perp}$. We define the CPS reformulation f^{cps} of f as follows:

$$f^{\mathrm{cps}}(S_i, exit, shunt) \;\widehat{=}\; \begin{cases} exit(f(S_i)) & \text{if } S_i \in \mathrm{dom}(f) \\ shunt(S_i) & \text{otherwise,} \end{cases}$$

where $exit$ and $shunt$ are *continuations*, i.e., functions that express "what to do next" upon completion of f. Generally speaking, continuations are invoked with one parameter, although that parameter might aggregate multiple objects (e.g., a pair of steps as opposed to a single step). Here, $exit$ is the "normal" continuation, which f^{cps} invokes with its result, when the result is a value other than $\perp$; whereas $shunt$ is the continuation that f^{cps} invokes with its original input parameter, when this parameter is not in the domain of f.

Rule functions, of the form used in Section 3, are easily and transparently reformulated in CPS. Consider, without loss of generality, a left rule function $t_{\mathrm{opL}[\overline{X}]} : S^{\perp} \to S^{\perp}$. It is reformulated in CPS as follows:

$$t^{\mathrm{cps}}_{\mathrm{opL}[\overline{X}]}(S_i, exit, shunt) \;\widehat{=}\; \begin{cases} exit(F_{\mathrm{opL}[\overline{X}]}(S_i)) & \text{if } C_{\mathrm{opL}[\overline{X}]}(S_i) \\ shunt(S_i) & \text{otherwise.} \end{cases}$$

A similar reformulation yields a CPS version for axioms and for right- and dual-rule functions.

A rule instance is a transparent implementation of the CPS reformulation of a rule function. To demonstrate the transparency, observe that the first parameter to the CPS reformulation corresponds to the parameter of the rule instance's `infer` operation; whereas the second and third parameters of the reformulation correspond, respectively, to the transfer node pointed to by `target` and the rule instance pointed to by `shunt`. When connected in this manner, invoking the exit continuation with a new step corresponds to firing the target reference with this new step. Likewise, invoking the shunt continuation with an unhandled premise step corresponds to invoking the infer operation of the shunt reference with this step.

Whereas a rule instance is a transparent implementation of a rule function, a daisy-chained sequence of rule instances is a transparent implementation of the sum of the corresponding rule functions. Such a sequence is one in which: All of the target references point to the same transfer node and the shunt reference of the rule instance in position $i < n$ in the sequence points to the rule instance in position $i + 1$. Fig. 8 illustrates a daisy-chained sequence. To show that such a sequence reifies the sum of the corresponding rule functions, it suffices to show that sums of functions are preserved under CPS reformulation, and then to argue that a daisy-chained sequence transparently implements the CPS form of sums.

Commensurate with the definition for the sum of functions f and g, their CPS versions are summed as follows:

$$(f^{\mathrm{cps}} \oplus g^{\mathrm{cps}})(x, exit, shunt) \;\widehat{=}\; f^{\mathrm{cps}}\ (x,\ exit,\ \lambda z\ .\ g^{\mathrm{cps}}(\ z,\ exit,\ shunt\ )).$$

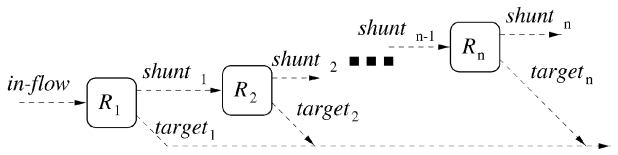


Fig. 8. A daisy-chained sequence of rule instances.

Thus, if a given value x is in the domain of f, then the exit continuation is invoked with $f(x)$. Otherwise, the shunt continuation of f^{cps}—in this case, the function formed by binding the exit and shunt parameters of g^{cps} with $exit$ and $shunt$, respectively—is invoked with x. Consequently, if $x \notin \text{dom}(x)$, then the behavior of $f^{\text{cps}} \oplus g^{\text{cps}}$ is the behavior of g^{cps}.

The method for daisy chaining rule instances is a transparent implementation of operator $\oplus$. Suppose, for example, that $\texttt{R}_f$ is the rule instance that corresponds to f^{cps}, and that $\texttt{R}_g$ is the rule instance that corresponds to g^{cps}. Making the `target` reference of both $\texttt{R}_f$ and $\texttt{R}_g$ point to the same transfer node corresponds to binding the exit continuation of both f^{cps} and g^{cps} to the same parameter ($exit$). Likewise, connecting the `shunt` reference of $\texttt{R}_f$ to $\texttt{R}_g$ corresponds to binding the shunt continuation of f^{cps} to the λ-abstraction that binds the exit and shunt continuations of g^{cps}.

5.2 Inference Nodes

Intuitively, an inference node is just a daisy-chained sequence of rule instances. However, interface incompatibilities prevent us from connecting assemblies of rule instances into the configurations described in Section 4. There are two sources of interface incompatibility. First, the `infer` operation of a singular rule instance has a different signature from that of a dual rule instance; the former expects to receive one step, whereas the latter expects to receive two. Second, the `infer` operation, which is invoked to send steps to a rule instance, differs from the firing operation, which is invoked to send steps to a transfer node. We overcome these incompatibilities by wrapping a daisy-chained sequence of rule instances with special *adapter* objects.

The adapter design pattern is used to "convert the interface of an existing class into another interface that clients expect" [4]. We use two kinds of adapters: input adapters and terminators. An input adapter converts the `infer` interface of a rule instance into the firing interface of a transfer node. Two varieties are needed: *s-adapters* for use with singular rule instances and *d-adapters* for use with dual rule instances. Both s-adapters and d-adapters provide a firing operation that is invoked with a single step parameter. In response to being fired with a step, an input adapter may *play* one or more steps by invoking the `infer` operation of the rule instance to which it is connected. An s-adapter plays a step immediately upon receiving it since singular instances require only a single premise step. However, a d-adapter cannot play steps as it receives them since it is fired with a single step at a time and a dual instance requires a pair of left- and right-operand steps.

In fact, because a dual transfer node receives all steps of the left operand before receiving any steps of the right operand, a d-adapter cannot play a pair of steps until after it has received all left-operand steps. A d-adapter has two modes of operation: *recording* and *playing* modes. While in recording mode, the d-adapter 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-adapter stores incoming steps in a local buffer. In playing mode, it pairs an incoming step with each step in this buffer and then plays each pair to a dual rule instance, one after another. The analyzer switches the d-adapter from recording mode to playing mode on completing the visit of the left operand and before starting the visit of the right operand.

In contrast with the input adapters, a terminator ties down the shunt out-flow of the last rule instance in a daisy-chained sequence. Recall that the `shunt` reference of a rule instance is another rule instance and not a transfer node; whereas, the shunt out-flow of a transfer node is connected to the in-flow of another transfer node. There are two kinds of terminators: *shunt grounds* and *shunt forwards*. A shunt ground provides an `infer` operation that receives a step (or pair of steps) and then returns without processing or further propagating the step (pair of steps), effectively discarding it. By contrast, a shunt forward provides an `infer` operation that receives a step and then propagates the step to a transfer node. We have three different kinds of terminators: a singular shunt forward, a singular shunt ground, and a dual shunt ground. There is no dual shunt forward, as a transfer node is fired with only one step.

A transfer node is a daisy-chained sequence of rule instances sandwiched between an input adapter and a terminator. The out-flow of the input adapter is connected to the in-flow of the first rule instance in the sequence and the shunt out-flow of the last rule instance in the sequence is connected to the in-flow of the terminator. Fig. 9 shows four different inference-node assemblies. At the top left is a singular node for $PDCC$, which a step analyzer assembles by connecting an s-adapter (SA) to a sequence of two rule-instances (S_1^L and S_2^L) and terminating this sequence with a singular shunt ground (SSG). The other three assemblies collaborate to form the box associated with $PROC$. These assemblies demonstrate the use of singular shunt forwards (SSF), a d-adapter (DA), and a dual shunt ground (DSG).

5.3 Static Assembly of Inference Nodes

At first glance, the function-call overhead incurred by assembling an inference node as a linked sequence of smaller encapsulated objects might appear to be unacceptable. However, Amalia implements the daisy chaining of rule instances and the attachment of adapters statically, not dynamically. This static composition affords all the transparency benefits of modeling inference nodes as assemblies of rule instances and adapters without incurring the efficiency costs.

Every rule instance is an instance of a rule class, which we generate automatically. A rule class uses a technique called *parameterized inheritance*: It is a C++ template class whose parent class is not specified until the template is instantiated. Such class templates are called *mixin classes* [6]. Mixin classes have the useful property that invocations (within the mixin class) of parent-class operations are

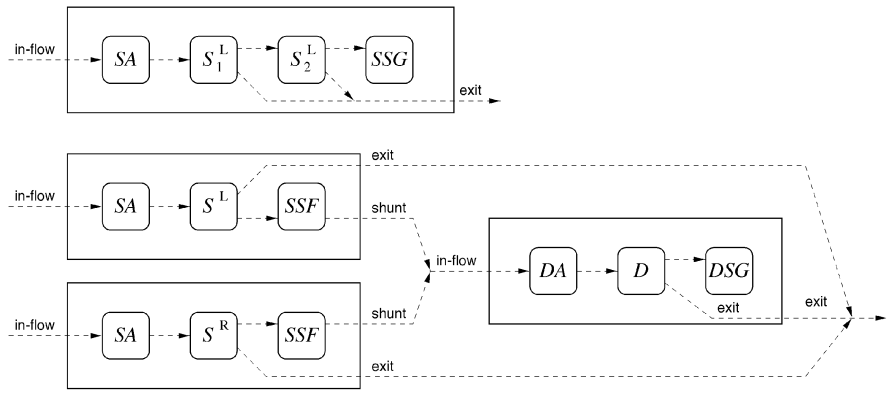


Fig. 9. Example inference nodes.

implemented using static binding and can therefore be inlined. We exploit this property to implement daisy chaining and adapter composition statically, thereby removing unnecessary indirection.

Fig. 10 depicts an example rule class that corresponds to the rule $\mathrm{disL1}[P,Q]$ from Table 1. The class is a mixin whose template parameter is named `SHUNT` to suggest that the template will be instantiated with another rule class (or a terminator) to which objects of class `disL1` will shunt unhandled premise steps.

The `infer` operation takes two parameters, an action `a` and a process `P_Prime`, which together constitute a premise step. The side-condition test (line 3) uses a function `delta` to check if the premise action `a` is equal to the termination action (δ). If `delta(a)` is false (the side condition of rule $\mathrm{disL1}[P,Q]$ is satisfied), then a conclusion step is computed and the `target` node is fired to propagate the conclusion step. (We show in the next section that invoking `target` fires the appropriate transfer node in the inference graph.) On the other hand, if the input step does not satisfy the side condition, then it is shunted (line 10). Shunting is performed by invoking the `infer` operation of the superclass (`SHUNT`). When the template is instantiated, the call to `SHUNT::infer(a, P_prime)` is implemented using static binding, which a good compiler can optimize away for efficiency.

```
    template<class SHUNT>
    class disL1 :  public SHUNT {
       ...
(1)    void infer( const Action* a,
(2)                const Process* P_prime )
(3)    { if( !  delta(a) ) {
(4)      // Conclude new step,
(5)      // and send to the outflow
(6)      target( a, new Dis(P_prime,Q) );
(7)    }
(8)    else {
(9)      // Shunt away the input step.
(10)     SHUNT::infer(a, P_prime);
(11)   }
(12) }
     ...
    };
```

Fig. 10. Example rule class.

Using mixin composition, we can daisy-chain an arbitrary sequence of rule classes so that the steps shunted by one rule instance are efficiently piped into another. Consider, for example, the inference node that is used to process steps from the left operand of an expression whose major operator is disable (`Dis`). Fig. 9 depicts such a node as a linked assembly of four objects—an s-adapter (SA), two daisy-chained rule instances (S_1^L and S_2^L) that correspond to rules $\mathrm{disL1}[P,Q]$ and $\mathrm{disL2}[P,Q]$, and a singular shunt ground (SSG). The C++ class for this inference node is declared as the mixin composition:

```
SAdapter< disL1< disL2< SSGround > > >
```

Thus, the node is a single, monolithic object, which incurs no indirection when choosing the appropriate rule function to apply to its input step. At the same time, it transparently implements the design for the node illustrated at the top of Fig. 9.

5.4 Linking Objects into Boxes

Next, we argue that an Amalia analyzer connects the nodes in an inference graph as prescribed by our model. Specifically, we show that the left and right nodes shunt unprocessed steps to a dual node, if one exists, and that all conclusion steps produced by these nodes are piped to the outflow of the box.

Amalia generates a visitor class, which provides a `visitOp` operation for each distinct AST operator `Op`, to implement a step analyzer. Fig. 11 illustrates a portion of the step analyzer class generated from the LOTOS rules. This portion includes code to assemble a box for the disable operator `Dis` (lines 1-7) and for the parallel operator `Par` (lines 8–17). The method `visitDis` takes a parameter `P`, which is a pointer to an AST whose root operator is a `Dis`; whereas, the method `visitPar` takes a parameter `P`, which is a pointer to an AST whose root operator is a `Par`. The step analyzer object itself carries, as a private data member, an object called `target`, which is a reference to an in-flow of the box into which any conclusion steps are forwarded (line 0). Recall from the previous section that a rule class's `infer` operation invokes this object to send conclusion steps to its exit out-flow.

As Table 1 indicates, the semantics of the disable operator are governed by two left rules, $\mathrm{disL1}[P,Q]$ and $\mathrm{disL2}[P,Q]$; a right rule, $\mathrm{disR}[P,Q]$; and no axioms. Thus,

```
    LotosStepAnalyzer::LotosStepAnalyzer( Outflow& o )
(0) :  target(o){}


    void LotosStepAnalyzer::visitDis( const Dis* P )
    {
(1)   SAdapter<
(2)     disL1< disL2< SSGround > > >   t_l(target);
      ...
(3)   SAdapter< disR< SSGround > >     t_r(target);
      ...
(4)   LotosStepAnalyzer                I_l(t_l);
(5)   P->getFirst()->Accept(I_l);

(6)   LotosStepAnalyzer                I_r(t_R);
(7)   P->getSecond()->Accept(I_r);
    }


     void LotosStepAnalyzer::visitPar( const Par* P )
     {
(8)    SAdapter< parL<SSForward> >       t_l(target);
       ...
(9)    DAdapter< parD<DSGround> >        t_d(target);
       ...
(10)   SAdapter< parR<SSForward> >       t_r(target);
       ...
(11)   t_l.tee(t_d);
(12)   t_r.tee(t_d);

(13)   LotosStepAnalyzer                 I_l(t_l);
(14)   P->getFirst()->Accept(I_l);

(15)   t_d.play();

(16)   LotosStepAnalyzer                 I_r(t_r);
(17)   P->getSecond()->Accept(I_r);
     }
```

Fig. 11. Example code.

the `visitDis` method creates two inference nodes, `t_l` and `t_r`. Node `t_l` is assembled from an `SAdapter`, the rule classes for the left rules, and an `SSGround` (lines 1 and 2). The node `t_r` is assembled in a similar manner, but using the rule class for $\mathrm{disR}[P, Q]$ (line 3).

In contrast, the semantics of the parallel operator are governed by a left rule $\mathrm{parL}[P, A, Q]$, a right rule $\mathrm{parR}[P, A, Q]$, a dual rule $\mathrm{parD}[P, A, Q]$, and no axioms. Thus, the `visitPar` method creates three different inference nodes, `t_l`, `t_r`, and `t_d`. Both `t_l` and `t_r` (lines 8 and 10) are terminated by an `SSForward`, instead of an `SSGround`, which means that unused steps will be forwarded to a dual node. This forwarding connection is established by invoking the function `tee` on `t_l` and `t_r`, passing a pointer to `t_d` as a parameter (lines 11 and 12). Continuing to elaborate the generated code in this manner, we argue that the structural model of a box is transparent with respect to how the nodes in a box are interconnected.

It remains to demonstrate how the nodes in a box are connected to the outflows of inference subgraphs for the operands. These connections are established by creating new visitor objects, which traverse the left and right operands of an AST, and initializing these visitor objects with references to either `t_l` or `t_r` as appropriate. For example, line 13 creates the visitor `I_l` that assembles and executes the inference graph for the left operand of `P`. Observe that the `target` object of `I_l` is initialized with `t_l`, which plays the role of the in-flow for steps produced by the inference subgraph associated with `P`'s left operand. Thus, any conclusion step generated by `I_l` flows into `t_l`. Line 14 actually performs the visit, using the protocol in Section 2.4, which assembles and executes the inference subgraph for the left operand of `P`. A similar idiom is used for the right operand in lines 16 and 17.

5.5 Execution Logic

Finally, we argue that the recording and playing logic defined in Section 3.2 accurately represents the operational behavior of the generated inference graphs. Consider, for example, the operational behavior of the `visitPar` method in Fig. 11. This behavior is modeled in Section 3.2 in terms of sequences of premise steps, which are assumed to be produced as a result of analyzing the operands of a given AST. We have already argued that any step generated by analyzing the left operand (in line 14) flows into `t_l` and any step generated by analyzing the right operand (in line 17) flows into `t_r`. We have also argued that, for each step S that flows into `t_l` (respectively, `t_r`), if S satisfies the side condition of the left rule $\mathrm{parL}[P, Q]$) (respectively, right rule $\mathrm{parR}[P, Q]$), the `infer` operation of the corresponding rule class sends a conclusion step to the out-flow, commensurate with Definition 2, case 2 (respectively, Definition 3, case 2). On the other hand, if S does not satisfy the side condition of the left (respectively, right) rule, then it is shunted to the in-flow of `t_d`. (As noted previously, Amalia selects this shunting behavior by instantiating the mixins that comprise `t_l` and `t_r` with the `SSForward` class.)

By design, the dual node `t_d` is initialized in recording mode, and it remains in recording mode until instructed to transition into playing mode. This transition occurs after the traversal of the left operand, but before the traversal of the right operand (line 15). Thus, the shunting of a step to `t_d` by `t_l` occurs when `t_d` is in recording mode. By design, `t_d` buffers incoming steps when it is in recording mode. The shunting of a step to `t_d` by `t_l` is therefore commensurate with Definition 2, case 3.

On the other hand, the shunting of a step to `t_d` by `t_r` occurs when `t_d` is in playing mode. By design, when a dual node in playing mode is invoked with a step, it loops through each of the buffered steps to create a pair of steps which it then passes as a parameter to the `infer` operation(s) of the rule classes for the dual rule(s). Thus, the shunting of a step to `t_d` by `t_r` is commensurate with Definition 3, case 3.

Finally, observe that the invocation of `t_d.play()` after the traversal of the left operand is accurately reflected in Definition 2, case 1: As there are no longer any steps left to process from the left operand, the collaboration transitions from recording to playing mode.

6 Conclusions and Related Work

Amalia is concerned with designing analysis software subject to three design goals: extension/contraction, tight integration into constraining external contexts, and assurance. To satisfy these disparate design concerns, we have integrated ideas from several subdisciplines. Specifically, we provide for extension/contraction by designing analysis algorithms in the GenVoca style. We address integration

with other program development tools using OO design patterns and generation technology. For assurance, we combine concepts from denotational semantics and data-flow programming, and discharge proof obligations through rigorous inspection of the code. The GenVoca domain model produced by Amalia is described in [1]. In this paper, we have focused on how to provide assurance in the face of the complex mechanisms that we adopted for efficiency and in order to achieve integration.

6.1 Integration with Other Tools

To integrate formal analysis capability into an existing target tool requires overcoming two obstacles. First, the analysis capability must interact with an existing representation via the APIs provided by that representation. Second, the capability to be integrated cannot arbitrarily add functionality or constraints to these existing APIs. We address the first obstacle by automatically customizing our step analyzers to use the APIs provided by an existing representation. We address the second obstacle by imposing only one additional constraint, namely, that the existing representation must implement traversal methods in accordance with the visitor pattern.

The visitor pattern allows us to separate step-analysis capability from the design of the AST classes, which, in turn, allows us to automate the generation of a step analyzer that operates on a given AST notation. However, other technologies for separating concerns could be used. For example, using adaptive programming [14], the processing currently performed by a visitor could be localized within a *propagation pattern* and woven into the AST classes. Alternatively, using aspect-oriented programming [15], we could associate *pointcuts* with AST classes and define an *aspect* that performs the processing currently performed by a visitor; the aspect could then be woven into the AST classes using the pointcuts. Each of these approaches—visitor-based design, adaptive programming, and aspect-oriented programming—allows for the design of step-analysis capability with only negligible effect on the design of an internal representation. We chose to use visitors because, unlike the other techniques, visitor-based designs can be implemented in standard programming languages, without the need for additional tool support, such as weaving tools.

While we have yet to undertake a major case study with a third-party tool, we designed Amalia around current best practices for maintenance and integration. For example, we use generators to customize generic designs to existing APIs and we use the visitor and other design patterns. This paper is concerned with achieving assurance in analysis capability that is designed for integration according to these current best practices.

6.2 Assurance

An inference graph is a network of objects that collaborate to compute an analysis. Here, the term *collaboration* refers to a protocol of behavior among multiple objects, which play *roles* in the collaboration. These collaboration-based designs typically involve a complex exchange of messages among a group of stateful objects that are working together to solve a problem. Research on formalizing and analyzing such collaboration-based designs is less mature than that for functional designs, which is why we opted to provide assurance through transparency; but some techniques for collaboration-based designs are being developed.

Fisler and Krishnamurthi [16] suggest an approach in which collaborations are modeled as sets of abstract-state-machine (ASM) fragments that are annotated with properties that the collaboration requires and properties that the collaboration provides. Assume-guarantee reasoning is used to prove properties of compositions that join ASM fragments at designated states. To use this approach with an Amalia-generated step analyzer, the components that make up an inference graph would need to be partitioned to form collaborations and the collaborations specified as ASM fragments and required/provided properties.

Parallels exist between our approach and that of Liu et al. [17] for showing correctness of code generated from ML specifications of *microprotocol* compositions. Microprotocols are compiled into functions that are parameterized by states and events. Optimizations result from specializing these functions for special states and/or events. In principal, this approach does not require a transparency argument because the microprotocols are implemented in a formal notation. However, in practice, a verification of a nontrivial protocol that involves more than one participant has yet to be fully automated.

Given the increasing popularity of design methods that encourage collaboration-based design (e.g., design patterns [4], aspect-oriented programming [15], and mixin layers [18]), we expect that there will be a growing need for reasoning techniques that are transparent with regard to the role structure of these collaborations. Such proofs typically rely on mathematical abstractions of source code. Because the connection between the actual code and the model is ultimately informal, they should be based on assumptions and abstractions that can be discharged by inspection. Thus, the results of this paper provide an example that could be consulted in developing transparent models and correctness proofs of other collaboration-based designs.

6.3 Automated Analyzer Generators

Many other researchers have worked on automatic generation of analyzers from formal-semantics descriptions. For example, CENTAUR [19] maps specifications in natural semantics into Horn clauses in Prolog and SPARE [20] synthesizes analysis algorithms from denotational-semantic specifications. Of particular interest, the Concurrency Workbench of North Carolina (CWB-NC) provides a full toolkit for analyzing operational specifications [21]. It contains a Process Algebra Compiler (PAC) that enables use of the CWB-NC with different specification notations. Like Amalia, the PAC takes as input definitions of a notation's abstract syntax and SOS rules defining the notation's semantics. From these definitions, it 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 abstract syntax, the PAC generates a front end that interfaces with the CWB-NC.

The OPEN/CAESAR effort [22] also shares many of the same goals as Amalia. OPEN/CAESAR is an architecture

for rapidly prototyping formal analysis tools from reusable algorithms, which can be mixed and matched and applied to multiple formal notations. Both OPEN/CAESAR and Amalia achieve flexibility and reuse by exploiting some of the same abstract data types that enable the decoupling of exploration algorithms from the specific formal notation(s) being analyzed. That similar abstractions and decoupling decisions play such central roles in two independent efforts suggests that these abstractions and decouplings are fundamental in efforts at integrating formal analysis capability into larger environments.

A related problem concerns the customization of analysis capability to accommodate multiformalism specifications (i.e., when a system specification comprises multiple component specifications, each written in a different formalism). Pezze and Young describe how to accomplish this customization without requiring a translation of each constituent formalism into a common semantic notation [23]. Rather, these specifications are mapped into a framework in which the semantics of composition can be tailored to the particular set of notations being composed. The semantics of composition are expressed in a rules language, which seems amenable to tool support so that custom analyzers could be automatically generated.

Some general transformation environments, such as the Software Refinery [24], can automate the generation of trusted step analyzers from declarative specifications of SOS rules. Similarly, general-purpose theorem provers like Isabelle [25] or HOL [26] can infer steps from declarative specifications of semantic rules. Day and Joyce [27] describe a framework that, given an embedding in HOL of a notation's semantics and of a specification, formalizes the specification's next-step relation as a HOL type. This framework uses symbolic functional evaluation to allow certain analyses (e.g., consistency and completeness checking and model checking) without using a theorem prover [28]. However, these efforts do not address issues of integration with external tools.

6.4 Validation Studies

We have validated that our design using inference graphs allows generation of efficient analyzers using a prototype implementation, written in C++. This prototype was used to build analyzers for two notations, pure LOTOS and a subset of linear-time temporal logic (LTL); the latter case study is described in [1].

Step analyzers for the notations were automatically generated from semantic rules. The LTL case study demonstrated that the generated step analyzer worked with two different implementations of the LTL class model. The step analyzers were also used to refine two high-level analysis components to generate labeled transition systems from specifications. The first analysis component `inferLTS` uses a step analyzer to elaborate the nodes and labels of a labeled transition system in a demand-driven fashion. This component is useful in simulation tools that do not construct an entire state graph. Because this analyzer does not perform a change in representation, a simulator can step the user through a sequence of behavior using a specification in the original notation to describe each state that is being traversed. The second analysis component `minLTS` derives a minimized LTS using Hopcroft's finite-automata minimization algorithm [29]. This component constructs the entire state graph associated with a specification; consequently, it can be applied only if the specification is finite state. These case studies demonstrated that the packaging of components in Amalia supports extension and contraction by virtue of layering higher- and lower-level components, having different characteristics, with a step analyzer.

Currently, Amalia does not handle rules with the expressive power that is provided by other automated analyzer generators, such as CENTAUR and CWB-NC. As we mentioned in Section 2, our step-analyzer generator currently only handles binary, unary, and nullary operators. This is not a fundamental limitation of the approach; in fact, an n-ary operator can be treated as a unary operator whose operand is a sequence of length n. Dealing with such operators requires extensions to our rules language for constructing and traversing sequences, and we are currently implementing these extensions.

We continue to extend Amalia with new languages and high-level analysis algorithms. We are currently specifying analyzers for StateCharts (using the semantics of [30]) and for Graphical Interval Logic [31]. We also plan to add model-checking and trace-checking layers to Amalia. Finally, we propose to conduct a case study integrating the analyzer generated for StateCharts into ArgoUML [32] to validate that the design decisions implemented by an Amalia-generated analyzer facilitate integration with tools in an existing environment.

Acknowledgments

This work is supported in part by the US National Science Foundation Grants EIA-0000433, CCR-9984726, and EIA-0130724; and by the US Office of Naval Research Grant N00014-01-1-0744. Any opinions, findings, and conclusions or recommendations expressed in this article are the authors' and do not reflect views of the sponsoring agencies.

References

[1] R.E.K. Stirewalt and L.K. Dillon, "A Component-Based Approach to Building Formal Analysis Tools," *Proc. 2001 Int'l Conf. Software Eng. (ICSE '01)*, pp. 167-176, 2001.

[2] D. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Eng.*, vol. 5, no. 2, pp. 128-138, 1979.

[3] L.K. Dillon and R.E.K. Stirewalt, "Lightweight Analysis of Operational Specifications Using Inference Graphs," *Proc. 2001 Int'l Conf. Software Eng. (ICSE '01)*, pp. 57–67, 2001.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Trans. Software Eng. Methods*, vol. 1, no. 4, pp. 355-398, Oct. 1992.

[6] B. Stroustrup, *The Design and Evolution of C++.* Addison Wesley, 1994.

[7] J.M. Boyle, R.D. Resler, and V.L. Winter, "Do You Trust Your Compiler?," *Computer*, vol. 32, no. 5, pp. 65-73, May 1999.

[8] G.D. Plotkin, "A Structural Approach to Operational Semantics," Technical Report DAIMI FN-19, Computer Science Dept., Aarhus Univ., 1981, http://www.dcs.ed.ac.uk/home/gdp/publications/SOS.ps.gz.
[9] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *The Formal Description Technique LOTOS,* P.H.J. van Eijk, C.A. Vissers, and M. Diaz, eds., pp. 23-73, 1989.
[10] Z. Manna and A. Pneuli, *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.
[11] I. Attali and D. Parigot, "Integrating Natural Semantics and Attribute Grammars: The Minotaur System," Technical Report 2339, INRIA Sophia Antipolis, 1994.
[12] L.K. Dillon and R.E.K. Stirewalt, "An Example Proof of Correctness for a Collaboration-Based Design," Technical Report MSU-CSE-01-6, Computer Science and Eng. Dept., Michigan State Univ., East Lansing, Mar. 2001, www.cse.msu.edu/~ldillon/Sel_pubs/ig.ps.
[13] A.W. Appel, *Compiling with Continuations.* Cambridge Univ. Press, 1992.
[14] K.J. Lieberherr, "Object-Oriented Software Evolution," *IEEE Trans. Software Eng.,* vol. 19, no. 4, pp. 313-343, 1993.
[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, , C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *European Conf. Object-Oriented Programming (ECOOP '97),* M. Aksit and S. Matsuoka, eds., pp. 220-242, 1997.
[16] K. Fisler and S. Krishnamurthi, "Modular Verification of Collaboration-Based Software Designs," *Proc. ESEC/ACM SIGSOFT Conf. Foundations of Software Eng.,* V. Gruhn, ed., pp. 152-163, Sept. 2001.
[17] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable, "Bulding Reliable, High-Performance Communication Systems from Components," *Operating Systems Rev.,* vol. 34, no. 5, pp. 80-92, Dec. 1999.
[18] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers," *Proc. 12th European Conf. Object-Oriented Programming,* 1998.
[19] P. Borras, D. Clement, T. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, "CENTAUR: The System," *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Practical Software Development Environments (PSDE),* P. Henderson, ed., pp. 14-24, 1988.
[20] G.A. Venkatesh and C.N. Fischer, "Spare: A Development Environment for Program Analysis Algorithms," *IEEE Trans. Software Eng.,* vol. 18, no. 4, pp. 304-318, Apr. 1992.
[21] R. Cleaveland and S. Sims, "Generic Tools for Verifying Concurrent Systems," *Science of Computer Programming,* vol. 42, no. 1, pp. 39-47, 2002.
[22] H. Garavel, "OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing," *Proc. First Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98),* 1998.
[23] M. Pezzè and M. Young, "Constructing Multiformalism State-Space Analysis Tools," *Proc. 19th IEEE Int'l Conf. Software Eng.,* W.R. Adrion, ed., pp. 239-250, May 1997.
[24] Reasoning Systems Incorporated, Palo Alto, Calif., *Refine User's Guide,* 2002.
[25] B. Gramlich and F. Pfenning, "Strategic Principles in the Design of Isabelle," *Proc. Workshop Strategies in Automated Deduction,* pp. 11-16, July 1998.
[26] A.J. Camilleri, "Mechanizing CSP Trace Theory in Higher Order Logic," *IEEE Trans. Software Eng.,* vol. 16, no. 9, pp. 993-1004, Sept. 1990.
[27] N.A. Day and J.J. Joyce, "A Framework for Multinotation Specification and Analysis," *Proc. Fourth IEEE Int'l Conf. Requirements Eng. (ICRE 2000),* pp. 39-49, June 2000.
[28] N.A. Day and J.J. Joyce, "Symbolic Functional Evaluation," *Proc. 12th Int'l Conf., Theorem Proving in Higher Order Logics (TPHOLs),* pp. 341-358, 1999.
[29] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
[30] G. Luttgen, M. vonderBeeck, and R. Cleaveland, "A Compositional Approach to Statecharts Semantics," *Proc. ACM SIGSOFT Symp. Foundations of Software Eng. (FSE 2000),* D. Rosenblum, ed., pp. 120-129, 2000.
[31] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna, "A Graphical Interval Logic for Specifying Concurrent Systems," *ACM Trans. Software Eng. Methods,* vol. 3, no. 2, pp. 131-165, Apr. 1994.
[32] J.S. Robbins and D.F. Redmiles, "Cognitive Support, UML Adherence, and XMI Interchange in ArgoUML," *Information and Software Technology,* vol. 42, no. 2, pp. 79-89, 2000.

Laura K. Dillon received the PhD degree in computer science from the University of Massachusetts, Amherst, in 1984. She was subsequently on the faculties of the Computer and Information Science and Electrical and Computer Engineering Departments at the University of Massachusetts, Amherst (1984-1985) and on the faculty of the Computer Science Department at the University of California, Santa Barbara (1985-1998). In 1997, she joined the Department of Computer Science and Engineering at Michigan State University, East Lansing. She has been active in the IEEE and ACM since 1980, serving over the years in various editorial capacities and on numerous program committees. Currently, she is a program cochair of the 2003 International Conference on Software Engineering and a member of the executive committee of the ACM Special Interest Group in Software Engineering. Her research interests center on formal methods for specification and analysis of concurrent software systems, programming languages, and software engineering.

R.E. Kurt Stirewalt received the PhD degree in computer science from the Georgia Institute of Technology in 1997 and has since served as an assistant professor in the Department of Computer Science and Engineering at Michigan State University, East Lansing. Prior to joining MSU, he worked as a research scientist in the Graphics, Visualization, and Usability (GVU) Center at Georgia Tech and as a member of the technical staff at the MITRE corporation in McLean, Virginia. He will serve as program cochair of the IEEE International Conference on Automated Software Engineering in 2004. His research concerns the practical use of formal and semiformal graphical models in the design, verification, and maintenance of large software systems. He is a member of the IEEE Computer Society.