A component-based approach to building formal analysis tools

R. E. Kurt Stirewalt, Laura K. Dillon
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
{stire,ldillon}@cse.msu.edu

Abstract

Automatic-verification capability tends to be packaged into stand-alone tools, as opposed to components that are easily integrated into a larger software-development environment. Such packaging complicates integration because it involves translating internal representations into a form compatible with the stand-alone tool. By contrast, lightweight-analysis components package analysis capability in a form that does not involve such a translation. Borrowing ideas from GenVoca and object-oriented design patterns, we developed a domain model and an automatic-generation framework for lightweight-analysis components. The generated components operate directly over the internal form of a specification without requiring a change in representation. Moreover, the domain model identifies several "useful subsets" that can be used to customize analysis capability to a particular application. We validated this domain model by generating lightweight analyzers for temporal logic and the behavioral subset of Lotos.

1. Introduction

Recent advances in automatic verification (e.g., [12, 9, 22]) have resulted in a growing suite of tools (e.g., SMV [30], SPIN [22], and CADP [18]) for verifying the correctness of formal specifications or finding counter-example behaviors that demonstrate flaws. This success has led many in the software engineering community to view such tools as black-box components upon which to base similar analysis capability for more popular, graphical software-engineering notations (e.g., SCR [20], RSML [27], and even OMT [35], and UML [6]). In fact, the last decade has witnessed a flurry of attempts to translate traditional modeling notations into forms that can be analyzed by these automatic tools (e.g., [1, 38, 10]). While conceptually these tools are black boxes, in practice, a stand-alone tool is difficult to efficiently integrate into a larger software-development environment. This paper explores the issues involved in packaging analysis and verification capability into lightweight components that are easily and efficiently integrated into a comprehensive environment.

The core problem we address is that automatic verification capability tends to be packaged into stand-alone tools. To apply the analysis when it is packaged in this manner, a CASE tool must transform its internal representation of a specification (hereafter called an *in-memory representation*) into and back from a different, but semantically equivalent representation that is expected by the stand-alone tool. These changes in representation incur run-time inefficiency due to translation and impose limiting constraints on the architecture of the CASE tool itself. We contend that these undesirable effects can often be avoided.

A better approach is to encapsulate analysis capability into *lightweight components* that compose seamlessly and efficiently within the architecture of a larger environment. Here we use the term *lightweight* to capture two useful properties. First, analysis capabilities should be performed on the in-memory representation of a specification, without translating the representation into another form or invoking an external tool. Second, the analysis software should be designed for *extension* and *contraction*—in the sense suggested by Parnas [33]—so that the software can be easily assembled into "useful subsets" tailored to a particular analysis need. An important class of contractions enable a tool designer to incorporate one or more simplifying assumptions about the specifications that the tool will be used to analyze (such as the specifications are known to be deterministic or to comprise only a finite number of states).

Unfortunately, designing analysis software subject to these criteria is difficult despite the regularity and elegance of formal notations and their analyses. Formal specification languages and their associated semantics are typically

0-7695-1050-7/01 $10.00 © 2001 IEEE

clean mathematical structures. Moreover, these languages and analyses are formalized using many of the same concepts. For example, notations as disparate as LOTOS [5], Promela [22], and formulas in linear temporal logic [29] are formalized in terms of *labeled transition systems*, which are a common abstraction used to reason about behavior. The thesis of our work is that the regularity, elegance, and commonality of different formal languages and their analyses can be exploited to design lightweight analysis components that are more easily integrated into a CASE tool.

Toward this end, we have investigated how to represent these notations and their analysis methods as a *software domain* [32]. By understanding the structure of this domain, we can apply results from domain-based reuse and automatic generation to systematically develop lightweight components. We identified a robust concept, called a *step analyzer*, that can be used to automatically refine high-level analyses—such as the creation of a labeled-transition system and the operation of a testing oracle—down to operations over the in-memory representation of a specification. This property enables verification tools to be packaged as lightweight components, a result that will directly benefit the developers of CASE tools and indirectly benefit the users of these tools.

Our experience in building components in this domain has culminated in an application generator, called Amalia,[1] that supports automated generation of lightweight analysis components for formal notations. The remainder of the paper is structured as follows. We first characterize the problem of representation change in this domain and introduce the concept of a step analyzer (Section 2). We then present our model of the structure of this domain, organized around the step-analyzer concept (Section 3). Next, we show how we used Amalia to validate this domain model for two formal notations: the behavioral subset of LOTOS [5] and a version of linear-time temporal logic customized for the generation of testing oracles [14] (Section 4). We conclude with a discussion of related approaches (Section 5).

2. Background

Verification capability tends to be packaged into stand-alone tools. To integrate a verification tool into a CASE tool therefore requires performing a change of representation, which we term *artifactual* to distinguish it from other kinds of representation change. The core problem can be restated as the use of artifactual representation change as a means to integrate analysis capability into a CASE tool. To understand how to systematically forestall artifactual changes, we have been investigating the combination of two ideas: First is our own research into efficient application of the *visitor pattern* (from object-oriented design) to implement the semantic analysis of the in-memory representation of specifications. Second is the application of GenVoca generators and the associated theory of hierarchical components [4], which is useful in representing the structure of a domain as a library of reusable, plug-compatible refinements.

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

2.1. Effectual vs. artifactual change

We view a change in representation as the result of an explicit design decision by a CASE-tool architect. We also distinguish between two different kinds of representation change according to the rationale used to justify the change. We call the decision to change a representation *effectual* when the decision is justified by the ability to exploit some structure in the target representation that is not explicit in the source representation. Representations take for granted different basic properties of data and structures; i.e., one representation can succinctly and explicitly convey information that might be hidden or difficult to express in another. For example, the decision to convert the control-flow graph of a program into static single-assignment (SSA) form [13] is effectual because the SSA form makes definition-use chains explicit and minimal; whereas this information (while conceptually in the control-flow graph) is not explicit.

By contrast, we call change in representation *artifactual* when the decision is motivated not by some useful explicit property of the target representation, but rather by the desire to reuse an existing software artifact, which could apply to the source representation except that its interface is incompatible with this representation. We contend that artifactual decisions lead to integration inefficiency and impose undesirable constraints on the architecture of a CASE tool. We are concerned with how to forestall artifactual decisions through lightweight analysis components.

Note that while artifactual changes are to be discouraged, we are not trying to forestall effectual changes, which are quite useful and form the basis for much of the work in formalizing popular software-engineering notations (e.g., [1, 38, 10]). On the other hand, effectual decisions are often coupled with artifactual decisions. Using lightweight components, this coupling is not necessary.

2.2. Integration by artifactual change

When integrating a stand-alone tool into a larger environment, we consider the tool to be a *black-box component* with an explicit interface and a hidden implementation. The interface to a stand-alone tool comprises a set of command-line options and an input-file format. Consequently, a CASE tool must translate between its in-memory representation of a specification and this expected file for-

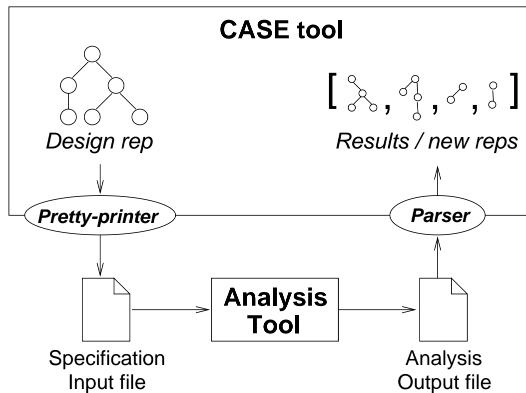

Figure 1. Black-box integration.

mat. This translation potentially involves *pretty-printing* the in-memory representation to a file in the form expected by the tool, running the tool with this new file as input, and then *parsing* the output generated by the tool (Figure 1).

Support for artifactual change as a means to achieve tool integration is an active area of research. In the program-analysis and reengineering communities, *data interchange languages* (e.g., [21, 26, 16]) standardize a means for exchanging information among a set of software tools. In most cases, these languages come with reader/writer infrastructure that simplifies the process of writing parsers and pretty printers. Recently, there has been an effort to standardize the exchange of data among CASE tools [17]. Each of these approaches support artifactual representation change, which we are attempting to avoid[2].

We wish to avoid artifactual representation change as a means for integration because these decisions impose many undesirable constraints on the architecture of the CASE tool. First, the translations exchange a large volume of data through input/output streams, which are significantly less efficient than in-memory translations. Second, the output from a tool is often designed to be read by people rather than machines. Consequently, it can be difficult to devise a reasonable grammar with which to parse the output. Third, the tool must be invoked as a separate process while the CASE tool waits.

2.3. In-memory step analyzers

To avert artifactual representation change requires the ability to analyze the in-memory representation of a specification. In our attempts to build lightweight analysis components for several languages, we identified a key capability that appears to be robust in this domain. This capability is called a *step analyzer*, because it computes the set of "next steps" that can be derived from a formal specification, and it works by traversing the in-memory structure of a specification. The process is described in detail in a companion paper [15]. Here we provide an overview of the approach.

A step analyzer is a software component for analyzing specifications by traversing their in-memory representation. Step analyzers implement the *visitor* pattern [19], which allows an operation over a linked object structure to be encapsulated into an object. Because a visitor object directly traverses in-memory representations, the solution forestalls an artifactual change in representation, at least with regard to computing next steps. Moreover, because visitor components are implemented in a standard programming language with only negligible effect on the design of the representation, the solution does not require an architect to concede flexibility in order to integrate analysis capability into a tool.

Visitors are a convenient mechanism for implementing functions whose domain is an inductively defined data type. Because the syntax of most formal languages are inductive, visitors provide a flexible mechanism for designing and implementing formal analyses. Moreover, because visitor components interact directly with the in-memory representation of a specification, an analyzer that is implemented using visitors is easier to integrate into a CASE tool than an analyzer implemented as a stand-alone tool. Finally, because the analysis is encapsulated into an object, we can apply object-oriented principles to design the visitor object itself for extension and contraction, thereby enabling the production of application-domain specific variants.

2.4. Refinements and parameterized components

One of our core hypotheses is that step analyzers enable the automatic construction of lightweight-analysis software. To justify this claim, we constructed a domain model in which higher-level analyses, such as labeled-transition system generators and testing oracles, use step analyzers to perform their functions. Specifically, we automatically *refine* high-level analyses to operate over in-memory representations by *instantiating* high-level analysis components with a given step analyzer. The connection between automatic refinement and component instantiation is articulated in the GenVoca model of hierarchical components.

In the GenVoca model of software generators, a refinement is represented as a reusable, interchangeable component [4]. Each component is an instance of a type, called a *realm*. A component may be parameterized by the types of components that it uses to implement its services. Component composition, or *layering*, is realized by instantiating one component with other components, such that each actual parameter type checks with the corresponding for-

[2] Note that the program-analysis domain need not share any qualities with the formal verification domain that we are trying to understand in this paper. The former is clearly much more complicated, and it could be that lightweight components in our sense are not feasible in that domain.

Level	Realms and contents
3	LTS = { inferLTS[LWA], minLTS[LTS] }
2	LWA = { lotosLwa[LOTOS], ltlLwa[LTL] }
1	LTL = { ltlTerm, ltlDNF } LOTOS = { lotosTerm }

Figure 2. Domain model.

mal parameter. A component b that must be instantiated with components of types T_1 and T_2 is written: b[T_1, T_2]. Types gather plug-compatible components into libraries. For example,

```
T = { a, b[T, X], c[T] }
```

defines a library of three components, each of whose type is T. Component a does not require parameters; whereas b and c require parameters of type T, and b also requires a component of a different type (X). A particular instantiation of components is called a *type equation*. The type equation

```
d = c[a]
```

represents a component d whose top layer is a c-component and that implements its services using an a-component (the bottom layer). Because the top layer of d is a c-component, the type of d is T, and we can add d to the library.

GenVoca components implement *large-scale refinements* [2], and GenVoca-component layering implements the successive refinement of an abstraction (represented by the realm of the top-layer component) in terms of another (represented by the realm of the bottom-layer component). We use this idea to automatically refine high-level verification algorithms to work over step analyzers, which traverse in-memory representations. The GenVoca model is mature and is supported by a wealth of research into efficient implementations in programming languages.

3. Domain model

In our domain model, lightweight analyzers are defined by type equations that reference components from many different realms (Figure 2). In such an equation, the "lightweight part" is handled by a component from the realm LWA, which stands for *light-weight analyzer*. By design, any verification algorithm that can be parameterized by an LWA component can be automatically refined into a lightweight component for a given notation. This key property makes artifactual representation changes unnecessary.

The domain model is organized around three distinct levels. The lowest level (i.e., *Level-1*) components provide services for constructing, accessing, and comparing in-memory representations for a given formal notation. This collection of services constitutes a basic useful subset of functionality: Different Level-1 components implement different equivalence relations by trading off the complexity of constructing a term in the notation against the complexity of testing two terms for equivalence. One class of tradeoffs involves simplifying the test for equivalence by constructing normal forms. Level-1 realms, called *constructor realms*, hide the implementation of this choice from higher-level components. To date, we have designed components for two notations, linear temporal logic and the behavior language of the formal notation LOTOS.

The next level (i.e., *Level-2*) components provide services upon which to build notation-independent verification tools. Level-2 components contain step analyzers that derive information by traversing the in-memory structure of a specification in a given notation. The top level (i.e., *Level-3*) components implement the labeled transition system abstraction. Operations at this level include returning the initial node of an LTS and computing the image of a node under the transition relation.

We designed two LTS components to explore the assembly of lightweight analyzers. The component inferLTS uses the LWA services to elaborate the nodes and labels of a labeled transition system in a demand-driven fashion. This component is particularly useful in simulation tools that do not construct an entire state graph. Because we never perform a change in representation, a simulator can actually step the user through a sequence of behavior, using a specification in the original notation to describe each state that is being traversed. Using the components in Figure 2, we can assemble three different analyzers according to the following type equations:

```
 lotosLTS  =  inferLTS[lotosLwa[lotosTerm]]
   ltlLTS  =  inferLTS[ltlLwa[ltlTerm]]
ltlDnfLTS  =  inferLTS[ltlLwa[ltlDNF]]
```

The analyzer lotosLTS provides services to explore the nodes and transitions of an LTS on demand given a LOTOS specification of the initial node. The analyzer ltlLTS does the same thing, except the initial node is specified in LTL rather than LOTOS. Both of these components use term equivalence to unify nodes. In contrast, the analyzer ltlDnfLTS uses logical equivalence to unify LTL terms.

The component minLTS derives a minimized LTS, using Hopcroft's finite-automata minimization algorithm [23]. There are two things to note about this component. First, it must construct the entire state graph associated with a specification; consequently, it can only be applied when we know that the specification is finite state. Second, the component is *symmetric*, which is to say that it must be instantiated with

another LTS component [4]. In our examples, we construct the following type equations:

```
lotosMin  =  minLTS[lotosLTS]
  ltlMin  =  minLTS[ltlDnfLTS]
```

Observe that whereas we were able to assemble three analyzers using inferLTS, minLTS can only support two given the current state of the library.

Unlike inferLTS, minLTS must elaborate an entire state graph to compute the minimization. This causes a problem when the type equation references ltlTerm because term equivalence in LTL can make a finite-state specification appear to be infinite, which would cause minLTS to run indefinitely. The problem is that for some formulas, successive invocations of ltlLwa[ltlTerm] produce steps that are made up of progressively larger labels and derived expressions. For example, the terms for the two temporal logic expressions $p \wedge \neg q$ and $p \wedge (\neg q \wedge p)$ are distinct, but they are logically equivalent. When constructing an LTS whose nodes represent expressions, it is important to exploit these equivalences to ensure that the algorithms terminate. Such decisions are easy to specify in our domain model, because constructor components for the same notation are plug compatible, regardless of the equivalence relation that they implement.

In summary, our domain model uses principles of layered design to organize and generate lightweight analyzers. Each analyzer is defined by a type equation, whose innermost components are concerned with details of analyzing in-memory representations. These components are referenced in larger type equations, whose referents provide high-level analysis services, such as elaborating the state space of a specification, model checking, etc. This separation allows an analysis algorithm to be automatically refined to operate on in-memory representations, which greatly simplifies the design of verification components. As new notations are added, existing analysis algorithms can be refined by component instantiation. Moreover, the domain model is open ended: It accommodates new, higher-level realms whose components are parameterized by existing realms, and it guarantees that components in these new realms will be lightweight.

4. Results

Generic LTS components can be automatically refined into lightweight analyzers for LTL and LOTOS using component instantiation. This flexible and efficient assembly relies on the design of the constructor and LWA realms, which we now describe. To support this design, we developed a suite of tools, collectively called Amalia, to generate these components from declarative specifications.

4.1. Level I: Constructor realms

Each different notation (e.g., LOTOS, ESTEREL, or LTL) leads to a different constructor realm. Consequently, whereas higher levels in our domain model are represented by a single realm, Level-1 is represented by a class of different realms. Because the interface to each constructor realm is notation-specific, we describe the basic requirements on a constructor-realm interface. To make the discussion concrete, we use examples from a specific constructor realm called LTL, which contains services for constructing well-formed formulae (wffs) in linear-time temporal logic (shown in Figure 3).

Requirement 1-1: *A constructor-realm interface must export a class model in which each concrete class corresponds to a different operator in the abstract syntax of the notation.* Here we use the term *class model* to define a collection of related classes that are organized by *generalization* (or "kind-of") and *aggregation* (or "part-of") associations. We use UML conventions [6] for denoting aggregation and generalization associations and for distinguishing abstract and concrete classes. Class models are similar to what Lieberherr calls a *class-dictionary graph* [28]. For example, in Figure 3 class Binary generalizes classes And and Or; whereas it aggregates two Wffs, which we call its operands. For brevity, aggregation associations in the diagram stand for explicit operations that construct composite objects from parts and operations that extract the parts from a composite object.

Of course, the interface to a real component will contain these constructor and accessor operations. For example, an aggregation in Figure 3 indicates that class Binary provides a constructor that takes two Wffs as parameters, and that it also provides two accessor operations: getFirstOperand and getSecondOperand, each of which returns a Wff. These constructor and accessor operations are abstract in the realm interface; so different components can provide their own implementations. By separating the process of constructing a specification from details of the data representation, different components are able to implement this process using one of many different forms (i.e., normal forms for some equivalence relation).

Requirement 1-2: *The class model must be visitable.* A class model is *visitable*, if it has been designed using the visitor pattern [19]. Such a design requires an abstract visitor class that provides "call-back" operations for each concrete class in the class model. In addition, the root of the class model must define a polymorphic operation called Accept, which takes as a parameter the visitor class, commensurate with the protocol of the visitor pattern [19]. In Figure 3, WffVisitor is the abstract visitor class.

Requirement 1-3: *The class model must provide an operation called* equiv *that allows a client to test the equiva-*

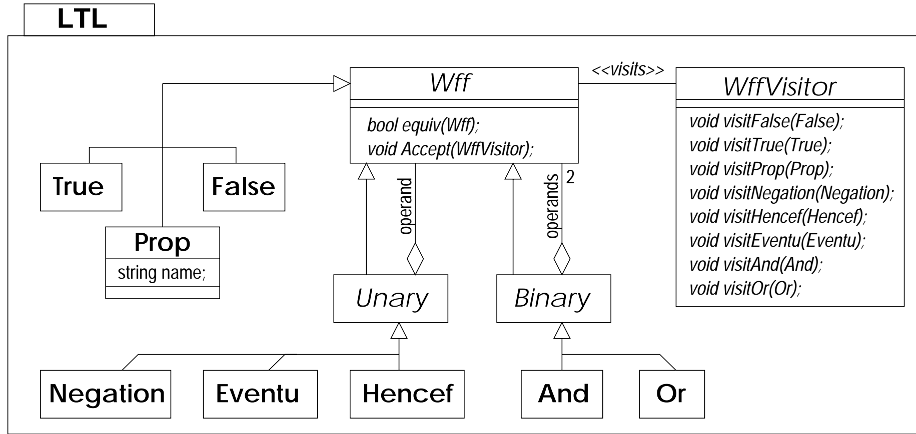

Figure 3. Framework definition of realm `LTL`.

lence of two representations. This operation allows higher-level analysis algorithms to compare specifications without knowing the details of these specifications. There are, of course, many different equivalence relations that can be imposed upon the terms of a notation's syntax (e.g., bisimulation, testing equivalence, etc.). Within a given constructor realm, different components implement different equivalence relations, all under the auspices of the `equiv` operation.

These features of constructor realms provide a powerful facility for making engineering trade-offs. Whereas some equivalence relations are expensive to compute; others are quite simple. Often domain knowledge can be employed to make a reasonable choice. Moreover, some equivalence relations are difficult to compute but easy to maintain by construction. For example, we currently have two components in the `LTL` library. One, called `ltlTerm` maintains the property that syntactically identical specifications share the same storage in memory. Thus, the equivalence relation here is structural equivalence over the terms of a specification. On the other hand, the component `ltlDNF` stores all wffs in a disjunctive-normal form[3] in order to detect a larger percentage of logical equivalences. Both components maintain these normal forms by construction.

4.2. Level II: LWA realm

The key to lightweight analysis is the `LWA` realm, whose interface provides services that are used by high-level verification algorithms, and whose implementations refine these services to operate over in-memory representations. Each `LWA` component is parameterized by a constructor realm. For example, the component `ltlLwa` is parameterized by the constructor realm LTL. Using this component, a designer could define a lightweight analyzer for temporal logic using term equivalence as `ltlLwa[ltlTerm]`, and using logical equivalence as `ltlLwa[ltlDNF]`.

[3] in which temporal operators are not distributed over logical operators

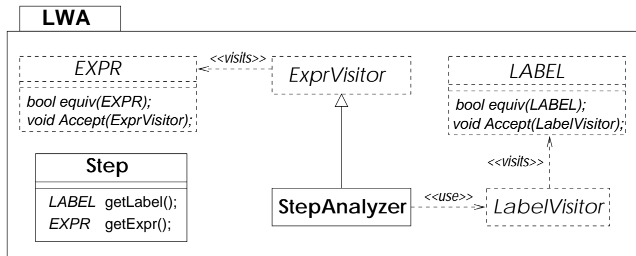

Figure 4. Framework definition of realm `LWA`.

`LWA` components provide step analyzers to derive information that feeds notation-independent analysis algorithms. To explain this derivation requires some terminology. The behavior of an operational specification is determined by computing the set of atomic events that the specification can perform and then constructing new specifications that denote the behavior after performing these events. The specification being analyzed is called the *subject*; the term *label* is used to represent the atomic event; and the new specification is called a *derivative*. Both labels and derivative specifications comprise an in-memory representation just like the subject specification. We use the term *step* to refer to a label paired with a derivative specification. A step analyzer computes the set of steps that can be derived from a subject specification by traversing that subject's in-memory structure.

Figure 4 depicts the interface of the `LWA` realm. `LABEL` and `EXPR` are abstract classes, whose objects are in-memory representations of labels and specifications respectively. Observe that while instances of these classes denote in-memory representations, the realm interface encapsulates all details about the representation, except that it is

```
(1) EXPR spec = /* create specification */
(2) ...
(3) set< Step >            steps
(4) StepAnalyzer           infer(steps);
(5) spec.Accept(infer);
```

Figure 5. Use of LWA-component classes.

visitable and that two instances can be compared for equivalence. Clients of a LWA component need not be concerned with any other details; they might just as well refer to wffs in temporal logic or to action and process specifications in LOTOS. To enforce the intended abstraction, we impose an interface requirement on components in this realm.

Requirement 2-1: *In every LWA component, the classes* LABEL*,* LabelVisitor*,* EXPR*, and* ExprVisitor *must be bound to abstract classes in the constructor realm that parameterizes the component.* As an example, consider again the component ltlLwa. In this component, both LABEL and EXPR would be bound to the class Wff in the realm LTL, and both LabelVisitor and ExprVisitor would be bound to the class WffVisitor in realm LTL. We indicate the fact that interface classes are defined in terms of classes in a subordinate layer by rendering the boxes of these classes in dashed lines. These bindings represent a flow of information from the interface of a subordinate component *up into* the interface of a higher-level component, a phenomenon that has been observed by other researchers in component technologies. In GenVoca, for example, a similar upward flow of information is required to solve the *interface-subjectivity problem* [3]); whereas in the C++ Standard Template Library (STL) [31], an upward propagation of parameter-type information is used to reduce the number of parameters required to instantiate a container whose elements are containers.

The realm interface also defines two new classes, Step and StepAnalyzer. Class Step is merely a representation of the steps that are derived by analyzing a subject specification. Note, however, that its operations have return types that match with EXPR and LABEL. The class StepAnalyzer is defined in more detail in [15]. What is important at this point is to note that StepAnalyzer is a subclass of ExprVisitor, which allows higher-level analysis components to analyze in-memory representations by visiting them with a StepAnalyzer.

Figure 5 illustrates how the various classes in an LWA component are used in concert to compute the steps of a specification. Assume an in-memory representation "spec"exists (line 1). To compute the set of steps that can be inferred from spec, we first create a set, called steps to hold the results of this computation (line 3). We then create a step analyzer (infer) that performs the derivation of steps by visiting the EXPR (line 4). Observe that we pass in the set steps as a parameter to the constructor of the step analyzer. Finally, we invoke the Accept method of spec, passing the step analyzer as a parameter (line 5). When this method invocation returns, steps will contain all of the steps that can be derived from spec.

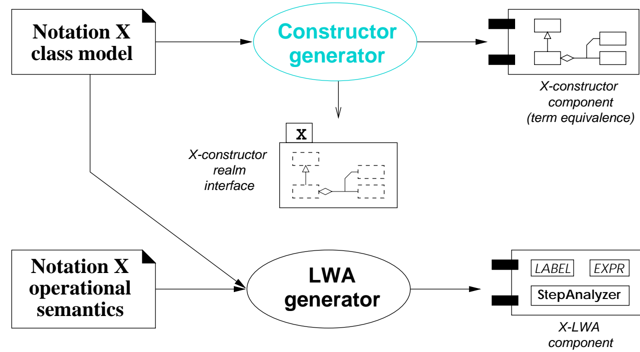

Figure 6. Amalia tool suite.

4.3. Amalia tool suite

Figure 6 depicts the architecture of the Amalia tool suite. In this diagram: documents[4] represent input files that a human designer must construct, ellipses represent Amalia tools, plug-ins represent generated components, and the package icon represents a realm interface. The constructor generator reads a class model, which describes the abstract syntax of a notation, and produces: (1) a constructor-realm interface for the notation and (2) a constructor component that implements term equivalence. The LWA generator reads a description of a notation's operational semantics and constructs an LWA component that is parameterized by the constructor realm for the notation. As of the publication of this paper, we are still completing work on the constructor generator, which is greyed out in the figure.

The LWA generator constructs LWA components from a class model that describes the syntax of a notation and a textual specification of a notation's operational semantics. Specifically, it generates a StepAnalyzer class and the packaging required to export the LABEL, EXPR, and Step classes to higher-level components. The procedure used to derive a step analyzer from an operational-semantic specification is described in [15]. As Figure 2 indicates, we have used the LWA generator to construct two components. The component lotosLwa computes steps by visiting specifications in the behavior-language subset of LOTOS. This component comprises 684 lines of C++ code over 22 files. It is generated from a semantic-description file that comprises 170 lines, a fourfold reduction in size. The compo-

[4] denoted by a crimped-page icon

nent `ltlLwa` computes steps by visiting specifications in linear temporal logic. This component comprises 445 lines over 14 files. It is generated from a semantic-description file that comprises 91 lines, a four-fold reduction in size.

4.4. Claims and disclaimers

Using a library of components that are organized according to the domain model in Section 3, we are able to assemble a respectable collection of lightweight analyzers using component instantiation. Moreover, the generation of `LWA` components is fully automatable, as is the generation of constructor components that implement term equivalence.

Because we are still completing the constructor generator the components `ltlTerm`, `ltlDNF`, and `lotosTerm` were implemented by hand. Implementing term-equivalence constructors is straight-forward: Associate with every operator a list of terms that are already constructed as a function of the sub-terms. Then, when a client invokes a constructor with sub-terms, consult this list before constructing a new term. We are currently building this capability into our constructor generator. This will allow the automatic generation of the components `ltlTerm` and `lotosTerm`

Generating constructor components for more complex equivalence relations is a much harder problem, which we do not yet know how to automate. The component `ltlDNF`, for example, implements a disjunctive-normal form equivalence. Upon construction, a term is encoded into a canonical form. We anticipate the need for more complex equivalence relations for LOTOS (e.g., bisimulation, testing equivalence, etc.), but as yet we have not needed these. While we do not yet know how to generate constructor components that implement these more complex equivalence relations, the mathematics of these relations are well understood.

Finally, we implemented all of the components in Figure 2 in C++, using *mixin layers* to implement parameterization as C++-template instantiation. For details on the use of mixin layers to implement GenVoca-component composition, we refer the interested reader to [36]. This technique allows one to write type equations *directly in C++*. Using C++ saved us from having to design a generator to assemble the type equations into code components. To implement this domain model in another language, say Java, would require building such a generator.

5. Discussion

This research contributes to the body of software-engineering knowledge in two ways. We identify and carefully describe an undesirable integration phenomenon—artifactual representation change—and present an alternative that forestalls this problem. To support this alternative, we present a domain model and accompanying tool suite for building lightweight analyzers that do not incur artifactual representation change. In this section, we discuss some interesting corollaries of this work and our thoughts about future directions.

5.1. Application generators

Our experience in building Amalia tools and components is another data point in the research on application generators. Following the GenVoca model, the guiding principle in the design of the domain model was that software components represent large-scale refinements. Our realms, specifically the constructor realms and the `LWA` realm, represent robust levels of refinement in the domain of lightweight analyzers. Consistent with [4], we view realms as class models, specifically as object-oriented frameworks [25, 24], so that the classes defined in a component instantiate the framework associated with the component's realm. Moreover, consistent with the findings of [3], we discovered the need for realm interfaces to be subjective, and we solved the subjectivity problem by allowing type information to flow, from parameter components, up into the realm interface of the component doing the instantiation. To date, our experience reaffirms the power and generality of the GenVoca model.

We believe we have also discovered a desirable form of composition that is not easily modeled in the GenVoca theory. Recall that Level-1 components in our domain model are gathered into, not one, but a class of different realms; and that this class of realms is specified by a set of constraints (R1-R3). We do not see an easy way to model these constraints any more formally in the existing GenVoca theory. Nor can we specify that each component in the `LWA` realm is parameterized by a realm that satisfies these constraints. On the other hand, with the aid of application generators, we are able to enforce these constraints. Ideally, we could specify them more declaratively and precisely, and use this specification as a basis for designing the application generators. We are currently exploring how to do this.

5.2. Trusted analysis

One promising benefit of Amalia is the support for guaranteeing the correctness of generated analyzers. The last ten years has witnessed a growing attention to the correctness of software analyzers. With the growing emphasis on trusted systems, we can only imagine that the demand for verification of analysis tools will increase. Leveraging the view that components in our libraries are large-scale refinements, we now explore how this structure can be used to build "trust" into Amalia-generated analyzers.

A recent article in *IEEE Computer* outlined the obstacles to building trust into compilers and proposed the use of transformation techniques to address the problem [8]. A transformational approach to reasoning about correctness is appealing because it breaks the verification problem down into small, conceptually manageable steps. The process begins with a specification that can be shown correct by inspection, and then proceeds to apply correctness-preserving transformations to synthesize a concrete program that satisfies the specification. If we cannot represent and transform the specification in its "correct-by-inspection" form, then correctness arguments will be much more complex and probably will not even be attempted.

In our domain, the initial specification is a description of the notation's formal semantics. We should be able to provide a description of these semantics to a trusted tool that refines it into a lightweight analyzer. Other researchers have looked at automatically generating analyzers from formal semantics descriptions. The closest to Amalia is CENTAUR [7], which maps specifications in natural semantics into Horn clauses in Prolog. Another example is the SPARE environment, which synthesizes analysis algorithms from denotational-semantic specifications [37]. In Amalia, the LWA generator refines an operational semantics description of a notation into an LWA component that computes steps based on these semantics. This step involves a proof of correctness, which we will provide in a forthcoming report.

Note that we do not include in this list transformation environments, such as the Software Refinery [34]. While programs written in Refine might well fit the category of trusted analyzers, the environment is a large monolith that cannot easily be integrated into a CASE tool. In fact, a similar criticism may be made of the CENTAUR environment, which requires an online Prolog engine in order to run.

5.3. Further elaboration of Amalia libraries

The most obvious direction for future work concerns how to further elaborate the existing Amalia library, whose structure and contents are shown in Figure 2. Vertical elaboration of Amalia requires building higher-level realms that use the services of `LTS`. Examples are model checkers and front-ends for testing oracles. Conversely, horizontal elaboration is concerned with adding components to the existing realms, specifically those at Level-1.

Regarding vertical integration, we are currently constructing a fourth level in the domain model, which contains model-checking components. Our current project is building an explicit-state CTL model checker [12] that uses the services of our LTS layer. We are also exploring how to build a symbolic model checker. In this work, we plan to exploit the structure of NuSMV [11], a refactored implementation of the classic SMV model checker from Carnegie Mellon University. Support for symbolic model checking will involve an effectual representation change, namely the translation of an LTS into a binary decision diagram (BDD). We are investigating the extent to which this translation can be implemented as a large-scale refinement to an LTS.

Regarding the further elaboration of Level-1, we are pursuing two avenues of research. First, we are looking at how to fit other formal languages into our framework. We are particularly interested in looking at how to exploit the work in formalizing informal notations (e.g., [38, 10]) to design constructor components that use in-memory representations of the informal notations. An open question is whether Amalia can be used to generate analyzers for graphical notations, such as UML dynamic models under a suitable formalization. We are currently exploring this question.

Second, we are looking at how to automate the generation of Level-1 components that implement more complex equivalence relations than term equivalence. Historically, these equivalences are not derivable from the SOS rules of a notation. This is because there are often several different equivalences that can be used to equate terms generated by the same set of inference rules. The standard approach for addressing this problem is to use a theorem prover. However, most theorem provers are heavyweight components and are not easily integrated into scalable libraries, like those defined by Amalia (Figure 2). Rather than use a theorem prover during analysis, we prefer to use theorem-proving techniques to automatically generate a constructor component that implements the relation. Essentially, this means moving the heavyweight functionality out of the CASE tool and into Amalia so that clients of the CASE tool do not have to pay for the inefficiency.

Acknowledgements Partial support for the first author provided by NSF grants CCR-9901017 and EIA-0000433. Partial support for the second author provided by NSF grants CCR-9896190 and EIA-0000433. We would like to thank Betty Cheng, Spencer Rugaber, Hamid Alavi, and Grant Birchmeier for their helpful comments on early drafts of this document.

References

[1] J. Atlee and J. Gannon. State-based model checking of event driven systems requirements. *IEEE Transactions on Software Engineering*, 19(3), January 1993.

[2] D. Batory. Refinements and separation of concerns. In *Proc. of the Second Workshop on Multi-dimensional Separation of Concerns*, 2000. Co-located with ICSE'2000.

[3] D. Batory and B. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Trans. Softw. Eng.*, 23(2):67–82, Feb. 1997.

[4] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Meth.*, 1(4):355–398, October 1992.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comp. Netw. ISDN Sys.*, 14(1), 1987.

[6] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1999.

[7] P. Borras et al. Centaur: The system. In *Proc. of SIGSOFT'88, Third Annual Symposium on Software Development Environments*, Boston, 1988.

[8] J. M. Boyle, R. D. Resler, and V. L. Winter. Do you trust your compiler? *IEEE Computer*, 32(5):65–73, May 1999.

[9] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the* 5th *International Symposium on Logic in Computer Science*, June 1990.

[10] W. Chan et al. Model checking large software specifications. *IEEE Trans. Softw. Eng.*, 24(7):498–520, July 1998.

[11] A. Cimatti et al. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[12] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite–state concurrent systems using temporal logic. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[13] R. Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[14] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proc. of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'96)*, pages 106–117, 1996.

[15] L. K. Dillon and R. E. K. Stirewalt. Lightweight analysis of operational specifications using inference graphs. In *Proc. of the 2001 International Conference on Software Engineering (ICSE'2001)*, 2001.

[16] J. Ebert, B. Kullbach, and A. Winter. GraX: An interchange format for reengineering tools. In *Proc. of the Sixth Working Conference on Reverse Engineering*, pages 89–98, 1999.

[17] J. Ernst. Introduction to CDIF, Sept. 1997. http://www.eigroup.org/cdif/intro.html.

[18] J.-C. Fernandez et al. CADP (caesar/aldebaran development package): A protocol validation and verification toolbox. In *Proc. of the 8th Conference on Computer-Aided Verification (CAV'96)*, pages 437–440, Aug. 1996.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.

[20] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Meth.*, 5(3):231–261, July 1996.

[21] R. Holt. Introduction to TA: The Tuple-Attribute language, 1997. http://www.turing.toronto.edu/∼holt/papers/ta.html.

[22] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[23] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publishing Company, 1979.

[24] R. E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40, Oct. 1997.

[25] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June/July 1988.

[26] R. Koschke, J. F. Girard, and M. Würthner. An intermediate representation for integrating reverse engineering analyses. In *Proc. of the Fifth Working Conference on Reverse Engineering*, pages 241–250, 1998.

[27] N. G. Leveson et al. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.*, 20(9), Sep 1994.

[28] K. J. Lieberherr. Object-oriented software evolution. *IEEE Trans. Softw. Eng.*, 19(4):313–343, 1993.

[29] Z. Manna and A. Pneuli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[30] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.

[31] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Mass, 1996.

[32] J. M. Neighbors. Draco: A method for engineering reusable software systems. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability Volume I: Concepts and Models*, chapter 12, pages 295–319. ACM Press, 1989.

[33] D. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, 5(2), 1979.

[34] Reasoning Systems Incorporated, Palo Alto, CA. *Refine User's Guide*.

[35] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[36] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of the 12th European Conference on Object-oriented Programming*, 1998.

[37] G. A. Venkatesh and C. N. Fischer. Spare: A development environment for program analysis algorithms. *IEEE Trans. Softw. Eng.*, 18(4):304–318, April 1992.

[38] E. Y. Wang, H. A. Richter, and B. H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *Proc. of the IEEE International Conference on Software Engineering*, May 1997.