

RETRIEVAL BY CONSTRUCTION: A TRACEABILITY TECHNIQUE TO SUPPORT VERIFICATION AND VALIDATION OF UML FORMALIZATIONS

MIN DENG*, R. E. K. STIREWALT[†] and BETTY H. C. CHENG[‡]

*Software Engineering and Network Systems Laboratory,
Department of Computer Science and Engineering,
Michigan State University, East Lansing, MI 48824, USA*

*dengmin1@cse.msu.edu

[†]stire@cse.msu.edu

[‡]chengb@cse.msu.edu

Recently, there has been growing interest in formalizing UML, thereby enabling rigorous analysis of its many graphical diagrams. Two obstacles currently limit the adoption and use of UML formalizations in practice. First is the need to verify the consistency of artifacts under formalization. Second is the need to validate formalization approaches against domain-specific requirements. Techniques from the emerging field of requirements traceability hold promise for addressing these obstacles. This paper contributes a technique called retrieval by construction (RBC), which establishes traceability links between a UML model and a target model intended to denote its semantics under formalization. RBC provides an approach for structuring and representing the complex one-to-many links that are common between UML and target models under formalization. RBC also uses the notion of value identity in a novel way that enables the specification of the link-retrieval criteria using generative procedures. These procedures are a natural means for specifying UML formalizations. We have validated the RBC technique in a tool framework called UBanyan, written in C++. We applied the tool to three case studies, one of which was obtained from the industry. We have also assessed ~~our results~~ using the two well-known traceability metrics: precision and recall. Preliminary investigations suggest that RBC can be a useful traceability technique for validating and verifying UML formalizations.

Keywords: Traceability; UML formalization; retrieval by construction; value identity; generative procedures; precision; recall.

1. Introduction

Recently, there has been significant interest in *UML formalization*, which associates formal semantics with UML [27, 31], thereby enabling rigorous analysis of its many graphical diagrams. Central to formalization approaches is the use of *formalization rules* to relate UML models to models in some formal specification language, such as Promela [13, 29] or SMV [23]. Formalization rules describe how a UML feature (or combination of features) manifests in target-model code [4, 15, 18, 25, 32, 37].

These approaches vary in the formality of the language used to write the rules and thus in the precision and completeness of the formalization specification. When the specification is sufficiently precise and complete, the rules can be executed, yielding a tool that translates UML models into target models. Unfortunately, many of the aforementioned approaches cannot be fully automated, which means a human designer must maintain both the source and target models. As would be expected, whenever two related artifacts are maintained independently, there is the potential for inconsistency. We refer to the need to check consistency among extant artifacts as the *verification problem*. In addition, many existing translators lack a precise specification, thereby obscuring the critical design decisions that they embody. Or a translator may have a precise specification but lack the transparency required to enlist the trust of the developers of critical systems, who are the most likely to need the analysis capability afforded by formalization. These scenarios suggest a *validation problem* associated with UML formalization, namely the need for an application developer to validate an existing formalization tool against the requirements of a given application domain. Both problems hinder the more widespread adoption and use of UML formalization. This paper proposes a traceability-based approach to address these problems.

When considering the verification and validation obstacles, we identified four kinds of artifacts that are potentially involved in a formalization: UML models, formalization rules, translators, and target models that are intended to denote the meaning of UML models under formalization. These artifacts are related to one another in complex ways that are difficult to precisely describe. Consequently, these relationships are difficult to establish, demonstrate, and maintain. We address this difficulty using ideas from *requirements traceability* [10], specifically the use of techniques that elucidate complex relationships among artifacts by discovering and explicitly representing so-called *traceability links* among them. In order to facilitate the construction and evaluation of the links, we treat source and target models under formalization as labeled graphs whose nodes correspond to instances of UML features and target-language features respectively. For example, a node in a source model might be an instance of a UML composite state or an event; whereas a node in the corresponding target model might be an instance of a Promela proctype declaration. In this context, a traceability link connects a node in a given source model to its manifestation (under formalization) in the corresponding target model.

This paper explores two issues in the use of traceability techniques to support the verification and validation activities that arise in the adoption and use of UML formalization. The first issue concerns how to represent and manage links of *variable arity*, i.e., links with multiple endpoints. Links of variable arity are common in UML formalization due to a phenomenon called the *code-distribution problem*, where a single instance of a feature in a UML model “touches” many elements in the target model that formalizes it. The code-distribution problem is not specific to UML formalization; in fact, Richardson and Green observed a similar phenomenon in artifacts that are derived using automated program-synthesis [30]. To address this

issue, this paper contributes a technique to structure such links so that they convey information about the purpose of the different link endpoints, including how they coordinate to implement the feature being formalized.

The second issue concerns how to best specify the criteria for retrieving links among artifacts that are related under a UML formalization. A key insight is that link construction (as opposed to retrieval) can be specified through so-called *generative procedures* (GPs), which describe how to construct the target-model manifestation of a given type of UML feature. Given the translational nature of UML formalizations, GPs are a natural means for specifying link construction. In order to address the second issue, this paper introduces the use of GPs to *retrieve* a link among entities in the extant target artifact. When in fact, a GP is a procedure for *constructing* entities in the target artifact. We use the term *retrieval by construction* (RBC) to capture this approach to establishing traceability. The technique exploits an alternative approach to reasoning about identity in object-oriented systems [14]. Briefly, by representing target artifacts in an object system that supports *value identity*, procedures that attempt to construct objects that are identical to existing objects will instead retrieve references to those existing objects. Thus, a GP will perform like a retriever when the target construct being generated already exists, and it will create a new structure when the target construct does not already exist. This capability enables a designer to affirm expected traceability links among extant models and to systematically detect potential inconsistencies.

Two major activities were pursued to validate our techniques. First, we developed a tool framework called UBanyan that provides a forward traceability tool for a previously developed UML-to-Promela formalization [24, 25]. UBanyan provides infrastructure to construct object systems that use value identity and various utilities needed to construct GPs. Using this infrastructure, we created a tool that takes a UML diagram and a Promela model that implements the diagram under formalization to produce a *traceability report*, including a textual link report and an HTML representation of the Promela model. UBanyan supports verification of UML formalizations by reporting any links that have to be constructed (rather than retrieved), thus indicating inconsistencies between the source and target artifacts. It supports validation by allowing users to select individual UML model features and highlighting the relevant target code in the HTML document. In this way, users can validate whether the target code matches their expectation for defining the semantics of the UML feature under question. The second validation tactic assessed how well UBanyan fares in practice, where we applied it to three case studies, one of which was obtained from industry [16]. We evaluated the results using two well-known information retrieval metrics [7]: *precision* and *recall*. Precision is the percentage of correct links among the retrieved links, whereas recall is the percentage of correct links among the actual links. From the results, we consider RBC to be a promising traceability technique for link retrieval.

We believe the UBanyan framework serves as a proof-of-concept of this approach to traceability. Moreover, we believe that RBC with its UBanyan prototype are a

first step to developing tools to support the verification and validation of UML formalizations. The remainder of this paper is organized as follows. Section 2 motivates the need for verification and validation of UML formalizations, and it also overviews related work on software traceability. Section 3 presents the code-distribution problem in the context of our UML-to-Promela formalization. This example formalization serves as the running example for this paper. Section 4 introduces our approach to traceability link representation and management. Section 5 describes our RBC technique for representing and managing traceability links. Section 6 presents our implementation of the RBC technique in the UBanyan framework, describes the results using the precision and recall metrics, compares our results to the results from the literature, and discusses the implication of our RBC technique. Section 7 gives concluding remarks and overviews future directions.

2. UML Formalization and Traceability: Background and Related Work

This section motivates the need for establishing traceability links between artifacts of a UML formalization. Related work on software traceability is also overviewed.

2.1. Related work on formalizing UML

Many projects have investigated how to define formal semantics for UML. These approaches can be broadly classified into two orthogonal categories. Some approaches [11, 19, 20] assign an *operational semantics* [33], which effectively processes procedures for simulating execution. Because these approaches do not involve derived artifacts, our traceability technique does not apply. Other approaches [9, 15, 17, 18, 37], including our UML-to-Promela formalization [24, 25], use a *denotational semantics* [33] approach to assign semantics to UML.

Most denotational approaches specify formalizations using informal rules that include prose descriptions for the conditions under which to apply the rules, and snippets of target code to be generated for a given source feature or a collection of features. This style of specification exhibits two characteristics. First, it does not define the semantics for each UML feature in isolation. This characteristic implies two consequences. It is difficult to justify the correctness of the semantics for each feature. Moreover, it is unclear whether the set of rules have considered all the contexts in which the rules can be applied. Second, the imprecision of natural language potentially hides ambiguities, conflicts, and incompleteness of the conditions under which the rules can be applied. As a result of both characteristics, building translators to automatically translate UML diagrams to target code means that tool developers are burdened with making numerous decisions for what code to generate in different contexts. We observed this problem firsthand when developing the Hydra translator to automatically generate Promela specifications from UML diagrams for a UML-to-Promela formalization [24, 25]. In doing so, we had to resolve several ambiguities to capture contexts for generating Promela for individual UML features.

A few approaches adopt formal notations to define the formalizations. For example, in a UML-to-Object-Z project [15], Kim and Carrington used Object-Z [34] schemas and axiomatic definitions; Goldsby proposed a variant *natural deduction system (NDS)* notation for formally specifying the formalization rules in a UML-to-C++ formalization [9]. We observe that such formal approaches are rare and informal approaches are predominant.

2.2. The verification and validation problems of UML formalizations

We see two fundamental obstacles to the adoption and use of UML formalizations in practice. The first obstacle concerns the verification of a target model that is intended to denote a given UML model under some formalization relation. Verification is necessary whenever there is the potential for inconsistency between the two models, such as the case when both the UML and the formal models are constructed and maintained by hand because the desired formalization lacks an automated translator. A less obvious verification task occurs when a UML translator is implemented using techniques that are not transparent with respect to the rule-based specification. In this case, verification tasks must be undertaken in order to gain confidence that the translator correctly implements its specification. This issue of *tool trust* is such a major concern in some domains, that developers require all tools (including compilers) to be verified against their specifications [2].

The second major obstacle concerns the validation of a particular formalization relation against the specific characteristics of a particular domain of application. Validation checks whether the semantics implemented by a formalization captures the intended or expected semantics. For example, assume a user acquires a formalization tool that automatically translates UML models into Promela specifications. How can the developer know that the design decisions embodied in the formalization are appropriate for a given application domain? Because UML has no formal semantics, it is possible for a given UML feature to be formalized in many, slightly different ways, and in fact, many of the aforementioned formalization approaches are distinguishable in this regard. For example, some formal specification languages assume a *synchronous* model of concurrency, where multiple concurrent processes may perform a computation step at the same moment in time; whereas other languages assume an *asynchronous* or interleaving model. Which model a given formalization assumes is of critical importance to a potential user of the translator, and if the translator is not accompanied by a specification of the formalization, then the user is faced with a difficult validation task before a decision to adopt the translator can be made.

Validation is also a major concern to one who is developing a new UML formalization. Formalization rules tend to be complex; rarely will a formal language provide a feature that offers the intended semantics for each type of UML feature. For example, a UML transition into, out of, or across the boundary of composite

states might manifest as (in a Promela formalization) a complex protocol of interaction among concurrent processes involving signal assignments and communication through queues vs. a synchronous concurrent assignment of new values to state variables (in an SMV formalization) [37]. Formalizations are difficult to validate given the number of possible combinations of target features that can be used to capture the intended semantics of a given UML feature. The key to understanding (and thus to validating) such a complex mapping is to be able to understand, *in isolation*, the target-language meaning of any given UML feature.

This paper proposes to use a traceability approach to aid in addressing these types of verification and validation problems; as such, related work in traceability is discussed next.

2.3. *Related work on establishing and representing traceability links*

Researchers have proposed a number of approaches for establishing and representing traceability links between a variety of software artifacts. We group these approaches into two categories. *Static approaches* [1, 12, 21, 22, 26, 30, 38], including the technique described in the current paper, identify the traceability relationships by analyzing artifacts with no runtime behavior information. Several approaches use *information retrieval* (IR) techniques to relate various types of artifacts, in forward or backward directions (e.g., code and documentation [1, 22], requirements and designs [12], requirements artifacts [38]). The software reflexion model approach [26] checks the conformance of the implementation code with the design model using mapping rules between the design model and the source model extracted from the implementation code. The work that is most closely related to our approach is that by Richardson and Green [30], who use automated synthesis to infer links. Perturbations to the source artifacts are reflected in the regenerated target artifacts and used to identify the links between the source and target artifacts.

In contrast to static approaches, *dynamic* approaches establish the links between artifacts through analysis of runtime behavior. For example, Egyed [6] identifies the traceability links between model elements in a software system by executing a set of test scenarios on the system. In his approach, the hypothesized traceability relationships between the model elements are either verified or refuted by executing the scenarios on the system.

Different from the above approaches, this paper addresses how to establish traceability links between the source UML models and the target formal models in order to support verification and validation of the formalizations. Since the derivation relationships between the source and the target artifacts can be formally specified, we use the semantic information between the source and the target artifacts for link retrieval. Other traceability techniques tend to use syntactic information rather than semantic information.

3. Example UML-to-Promela Formalization

Traceability links elucidate complex relationships between two artifacts. We contribute results that are prerequisite to the construction of tools and environments to support these activities, specifically: (1) an understanding of the nature of links in this domain, and (2) a technique for implementing link-retrieval procedures expressed in a manner that is natural in this domain. We present these contributions with the aid of a running example, which is taken from a previously developed UML-to-Promela formalization [25] (Sec. 3.1). The details of this example illustrate the code-distribution problem, where the formalization of certain UML features produces code that crosscuts the modular structure of the target language (Sec. 3.2). This problem complicates the design of traceability links, since these distributed lines of code are related in complex ways. In order to support verification and validation, the design of the links should reveal the complex relationships among the distributed code.

3.1. Running example

As a running example to illustrate our techniques, we now briefly describe a UML formalization [24, 25] that maps state diagrams into specifications in the Promela specification language [13, 29] so that they can be analyzed using the SPIN model checker [13, 35]. For ease in understanding, we limit our discussion to a small but illustrative example that involves two artifacts, the UML state diagram in Fig. 1 and the (excerpted) Promela model in Fig. 2. For readability purposes, we use italics to refer to source, target, and traceability model elements, courier font for code elements, and the sans serif font to refer to classes for the source, target, and traceability artifacts. Note that the code in Fig. 2 was generated automatically from the state diagram using Hydra, a prototype translator previously developed for the UML-to-Promela formalization [3].

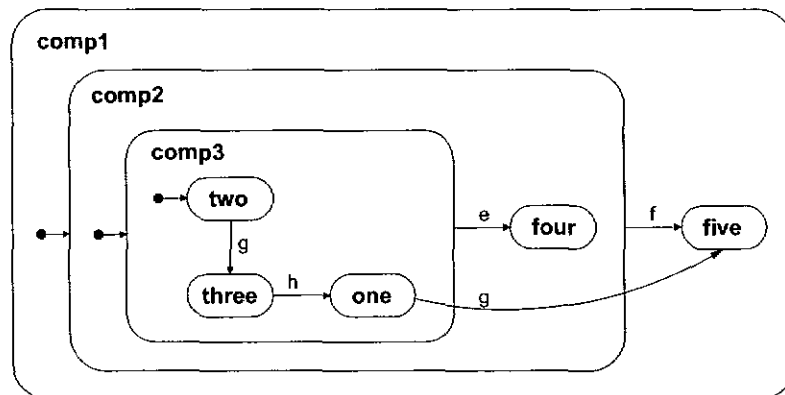


Fig. 1. An example UML state diagram. A transition is labeled by its event. Transitions on event *e*, *f* and *h* are referred to as t_e , t_f and t_h , respectively. Transitions on event *g* from *two* to *three* and from *one* to *five* are referred to as t_{g_1} and t_{g_2} , respectively.

```

1  proctype comp1(mtype state)          1  proctype comp2(mtype state)
2  {                                     2  {
3  int comp2_pid;                       3  int comp2_pid, comp3_pid;
4  mtype msg;                           4  mtype msg;
5  int dummy;                           5  int dummy;
6  /* Init state */                     6  /* Init state */
7  goto to_comp2;                       7  goto to_comp3;
8  /* State five */                     8  /* State four */
9  five:                                9  four:
10 if                                     10 if
11 ::skip -> false                       11 ::atomic{comp1_q?[f] -> comp1_q?f}; wait!_pid, st_five; goto exit
12 fi;                                   12 fi;
13 /* Link to composite state comp2 */  13 /* Link to composite state comp3 */
14 to_comp2: comp2_pid = run comp2(msg); 14 to_comp3: comp3_pid = run comp3(msg);
15 wait??eval(comp2_pid), msg;           15 wait??eval(comp3_pid), msg;
16 if                                     16 if
17 ::msg == st_five -> goto five          17 ::msg == st_five -> wait!_pid, st_five; goto exit
18 fi;                                   18 ::msg == st_four -> goto four
19 exit: skip                            19 fi;
20 }                                     20 exit: skip
                                         21 }

```

(a) Code generated for comp1 composite state. (b) Code generated for comp2 composite state.

```

1  proctype comp3(mtype state)
2  {
3  int comp2_pid, comp3_pid;
4  mtype msg;
5  int dummy;
6  /* Init state */
7  goto two;
8  /* State one */
9  one:
10 if
11 ::atomic{comp1_q?[e] -> comp1_q?e}; wait!_pid, st_four; goto exit
12 ::atomic{comp1_q?[f] -> comp1_q?f}; wait!_pid, st_five; goto exit
13 ::atomic{comp1_q?[g] -> comp1_q?g}; wait!_pid, st_five; goto exit
14 fi;
15 /* State two */
16 two:
17 if
18 ::atomic{comp1_q?[e] -> comp1_q?e}; wait!_pid, st_four; goto exit
19 ::atomic{comp1_q?[f] -> comp1_q?f}; wait!_pid, st_five; goto exit
20 ::atomic{comp1_q?[g] -> comp1_q?g}; goto three
21 fi;
22 /* State three */
23 three:
24 if
25 ::atomic{comp1_q?[h] -> comp1_q?h}; goto one
26 ::atomic{comp1_q?[e] -> comp1_q?e}; wait!_pid, st_four; goto exit
27 ::atomic{comp1_q?[f] -> comp1_q?f}; wait!_pid, st_five; goto exit
28 fi;
29 exit: skip
30 }

```

(c) Code generated for comp3 composite state.

Fig. 2. Promela code generated for composite states.

3.1.1. UML state diagrams

A UML state diagram is a hierarchical state machine, i.e., a state machine whose states may comprise substates. A *simple state* contains no substates; whereas a *composite state* may contain either simple states or (nested) composite states. For example, Fig. 1 contains five simple states, labeled *one* through *five*, and three composite states, labeled *comp1*, *comp2* and *comp3*. We distinguish *direct* from *indirect* substates of a composite state. Unless otherwise noted, a substate means a direct substate. For example, in Fig. 1, state *two* is a direct substate of state *comp3* and an indirect substate of states *comp2* and *comp1*.

A transition connects a *source* state to a *target* state, either of which may be composite, and is labeled by an *event*, which describes the conditions under which the transition may fire. A transition is said to be *enabled* if control resides in the source state (or a direct or indirect substate of the source state if the source is a composite state) and any conditions associated with the event are satisfied. Once enabled, a transition may *fire*, after which control will reside in the target state. For brevity, transitions on events *e*, *f* and *h* in Fig. 1 are referred to as t_e , t_f and t_h , respectively; transitions on event *g* from state *two* to state *three* and from state *one* to state *five* are referred to as t_{g_1} and t_{g_2} , respectively. The transition t_f is enabled when control resides in any of the simple states *one* through *four*, and after it fires, control will reside in the simple state *five*. Conceptually, transition t_f abbreviates a collection of transitions, emanating from all direct or indirect substates of its source state *comp2*. In the context of this paper, we refer to a transition such as t_f and t_{g_2} as a *hierarchical transition* because its firing causes the exit of more than one source state, whereas a transition such as t_{g_1} and t_h is considered a *simple transition*, because its firing only leads to the exit of one source state. The visual abbreviation exhibited by t_f is an important characteristic of visual languages, such as UML, but it contributes to the code-distribution problem.

3.1.2. Promela implementation of states

This formalization incorporates two major design decisions with regard to implementing UML states in Promela. First, every composite state (including the one that represents an entire state diagram) is implemented by a Promela process, whose behavior is declared using a *proctype* declaration. By convention, a *proctype* is named after the composite state whose behavior it declares, e.g., the *proctype* associated with a composite state *comp* will be named *comp*. Second, each direct substate *sub* of a composite state *comp* is implemented by a block of code, hereafter referred to as a *substate block*, in *proctype comp*. A substate block begins with code to perform the entry actions (if any) of the UML state it implements^a and ends with code that explicitly transfers control to another state block. By convention, the block associated with *sub* is labeled with an identifier named *sub* if *sub* is a simple state and *to_sub* if *sub* is a composite state.

^aFor brevity, none of the states in our example contain entry actions.

If *sub* is a composite state, then the corresponding substate block will contain code that spawns^b a new process to implement the behavior of *sub* (i.e., an instance of proctype *sub*) and then blocks until this new process terminates. Henceforth, we refer to the spawning process as the *parent* and the spawned process as the *child*. Because Promela processes run concurrently and asynchronously, process execution must be coordinated to implement this behavior. Thus, after spawning a child process, the parent suspends itself, waiting for the child to terminate before continuing its own execution. This coordination is implemented by having the parent wait for a termination message from the child on a global queue called *wait*. We say a process is *active* if it is not blocked waiting on a message in the *wait* queue. If the source UML model contains no concurrent composite states, then at most one process should be active at any moment in time.

As a concrete example, consider the state diagram in Fig. 1 and the Promela code generated for it under our formalization in Fig. 2. Three proctypes — *comp1*, *comp2*, and *comp3* — are generated for composite states *comp1*, *comp2* and *comp3*, respectively. Because *comp1* contains a simple state (*five*) and a nested composite state (*comp2*), proctype *comp1* contains two labeled locations, *five* (line 9 in Fig. 2(a)) and *to_comp2* (line 14 in Fig. 2(a)), the second of which labels a block of code that spawns a new process of type *comp2*. Similarly, proctype *comp2* declares two labeled blocks, *four* and *to_comp3*, starting from lines 9 and 14 in Fig. 2(b); and proctype *comp3* contains labeled blocks, *one*, *two* and *three*, starting from lines 9, 16 and 23 in Fig. 2(c), respectively.

3.1.3. Promela implementation of transitions

In a UML state diagram without concurrent states, control will always reside in exactly one simple state and its one or more (containing) composite states. Thus, when a transition fires, control may leave one or more of the states in which it resided prior to the transition. For example, when transition t_f fires, control will leave the composite state *comp2*. If, when the transition fired, control resided in the simple state *four*, then control will also leave this state; otherwise it will leave the composite state *comp3* and one of the simple states *one* through *three*. Likewise, when transition t_{g2} fires, control will leave the composite states *comp2* and *comp3* and the simple state *one*. In the UML-to-Promela formalization, leaving a composite state means terminating the corresponding process. In the case of deeply nested composite state hierarchies, firing a UML transition might involve terminating multiple processes.

This formalization implements the firing of a UML transition as a coordinated sequence of *transition steps*, each of which manifests as a Promela guarded command whose guard implements *event observation* and whose action implements *control propagation*. Event observation is the act of detecting that an event has

^bPromela uses the term *run* rather than *spawn*.

occurred within a state from which a transition on that event is possible. For example, in Fig. 2(c), the if-fi statement on lines 24 through 28 contains three transition steps that implement the logic for observing events *h*, *e*, and *f* — all of which trigger a transition out of state *three*.^c In these examples, each guard attempts to match (and then atomically remove) a symbol (i.e., *h*, *e*, or *f*) from the head of an event queue called *comp1_q*. We refer to transition steps that use this form of event observation, i.e., matching a symbol on an external event queue, as *direct observers*. Control propagation is the act of transferring control to a substate block that implements the target state. If this substate block is local to the current process, then control can be propagated using a *goto* statement. For example, the transition step on line 25 in Fig. 2(c) implements a transition whose target state (*one*) corresponds to a substate block in *comp3*. In the sequel, we refer to such a transition step as a *local propagator*.

When a UML transition emanates from or crosses the boundary of a composite state, the event observation and control propagation required to model the firing of this transition become more complex, requiring the coordination of multiple transition steps among multiple processes. As a concrete example, recall the transition *t_f* in Fig. 1, which can fire when control resides in any direct or indirect substate of *comp2*. Because event *f* could be observed whenever control resides in *comp2*, and because state *five* manifests as a substate block in proctype *comp1*, each of three proctypes, *comp1*, *comp2* and *comp3*, must contain transition steps to implement the firing of *t_f*. Moreover, the transition steps in a child process must coordinate with a transition step in its parent.

For example, consider what happens when event *f* occurs when control resides in state *four*. The event will be directly observed in proctype *comp2* via the transition step on line 11 in Fig. 2(b). However, because the target state (*five*) is not local to *comp2*, control cannot be propagated by simply branching to a label. Rather, this process must communicate the observation of *f* to the process that contains the substate block for state *five*. This latter process will always be a parent of the process that directly observes the event; in this case, it is the immediate parent, which is an instance of *comp1*. The child process initiates the communication by sending a message to its parent process and then immediately terminating. The message contains the child's process identifier (PID) and a symbolic identifier that names the intended target of the transition, and the message is added to a global message queue called *wait*. Each process maintains its PID in a local variable called *pid*; thus the transition step on line 11 in Fig. 2(b) adds the message (*pid*, *st_five*) to the *wait* queue and then branches to *exit*, which effectively terminates the process. We refer to such transition steps as *remote propagators*, because control is propagated to another process.

^cNote that in Promela, the options of an if-fi statement are treated as guarded commands, meaning that the system will non-deterministically choose from among the enabled options and block if none are enabled.

The parent process must also participate in the communication, i.e., if a child remotely propagates control by sending a message and terminating, the parent must be receptive to this message and further propagate control as appropriate. This behavior is implemented by a transition step in the parent which observes the event not by checking for a symbol on the event queue (`comp1_q`) but rather by looking for a control-propagation message from the child process that it spawned. For example, the guard in the transition step on line 17 in Fig. 2(a) checks the value of the local variable `msg`, which was bound to the second component of the message read off the `wait` queue on line 15, against the symbolic identifier `st_five`. We refer to such transition steps as *indirect observers*.

Notice that a given transition step is either a direct or an indirect observer and either a local or a remote propagator. For convenience, we give each combination a name. A local propagator and an indirect observer is called a *root transition step*, that is, the transition is the root of a hierarchy of transitions with respect to the outermost containing composite state. A remote propagator and an indirect observer is called a *bridge transition step*; from the perspective of an inner composite state, the transition serves as a bridge between the elements of the inner composite state to the destination state of the transition. A local propagator and a direct observer is called a *simple transition step*. A remote propagator and a direct observer is called a *leaf transition step*; here, the transition emanates from the boundary of a composite state containing simple states or the transition emanates from a simple state crossing its containing composite states.

Figure 2 contains examples of each of the four kinds of transition steps. For example, a bridge transition step appears on line 17 in Fig. 2(b). Moreover, these examples demonstrate that a single UML transition may manifest as a collection of transition steps, which are distributed among various proctype declarations. In fact, transition t_f , engenders six different transition steps on lines 12, 19 and 27 in Fig. 2(c), lines 11 and 17 in Fig. 2(b), and line 17 in Fig. 2(a). This code-distribution problem is a major obstacle to traceability.

3.1.4. Summary of key design decisions

In order to clearly present the key design decisions involved in formalizing UML to Promela, Table 1 gives a concise summary and examples of the design decisions in formalizing the UML state diagram shown in Fig. 1 into the Promela code shown in Fig. 2. We organize the decisions and examples into two categories, one for formalizing states and the other for formalizing transitions, respectively.

3.2. The code-distribution problem

The code-distribution problem occurs when a single instance of a UML feature engenders code (under formalization) that crosscuts the modular structure of the target language. Code distribution is one of the major contributors to the frequency of links with variable arity between artifacts that are related under formalization.

Table 1. A summary of design decisions involved in the UML-to-Promela formalization.

| UML feature | Design decision (UML → Promela) | Example (Figure 1 (UML diagram) → Figure 2 (Promela code)) |
|-------------|--|--|
| state | a composite state → a proctype declaration | <i>comp1</i> → proctype <i>comp1</i> in (a) <i>comp2</i> → proctype <i>comp2</i> in (b) <i>comp3</i> → proctype <i>comp3</i> in (c) |
| | a substate → a labeled block of code called substate block | <i>comp2</i> → lines 14-18 in (a) <i>comp3</i> → lines 14-19 in (b) <i>one</i> → lines 9-14 in (c) <i>two</i> → lines 16-21 in (c) <i>three</i> → lines 23-28 in (c) <i>four</i> → lines 9-12 in (b) <i>five</i> → lines 9-12 in (a) |
| transition | a simple transition → a transition step | <i>t_g</i> → line 20 in (c) <i>t_h</i> → line 25 in (c) |
| | a hierarchical transition → a coordinated sequence of transition steps distributed among multiple proctype declarations. Transition steps in a child process must coordinate with a transition step in its parent. | <i>t_e</i> → lines 18 (b), 11, 18 & 26 in (c) <i>t_f</i> → lines 17 in (a), 11 & 17 in (b), 12, 19 & 27 in (c) <i>t_g</i> → lines 17 in (a), 17 in (b), 13 in (c) |
| | Note: A transition step can be either an indirect or direct observer, and a local or remote propagator. The four combinations are: | |
| | 1) a local propagator & an indirect observer (a root transition step) | lines 17 in (a), 18 in (b) are root transition steps |
| | 2) a remote propagator & an indirect observer (a bridge transition step) | lines 17 in (b) is a bridge transition step |
| | 3) a local propagator & a direct observer (a simple transition step) | lines 20 & 25 in (c) are simple transition steps |
| | 4) a remote propagator & a direct observer (a leaf transition step) | lines 11 & 17 in (b) and 11, 12, 13, 18, 19, 26 & 27 in (c) are leaf transition steps |

In the example study, most of the state-diagram features crosscut the structure of target language (Promela). We now briefly illustrate the problem for UML states and transitions.

Notice in our running example that the composite state *comp3* manifests both as a proctype that declares the behavior of *comp3* (Fig. 2(c)), and as a line of code in the proctype that declares the behavior of *comp2* (line 14 in Fig. 2(b)). Moreover, the behavior of a composite state with deeply nested substates (e.g., *comp1*) will engender multiple proctype declarations. These simple examples illustrate how code generated from a UML state may be distributed among multiple Promela proctype declarations.

The code-distribution problem is even more pronounced in the code generated from transitions. For example, a hierarchical transition, such as *t_f*, engenders six different transition steps that are scattered across three different proctype declarations. Before continuing, we note that the code-distribution problem is not limited to the particular UML formalization that we are using as an example. We have seen it in the UML-to-SMV formalization of [37] and the UML-to-C++ formalization of [9].

4. Traceability Link Representation and Management

In this paper, we propose to support the verification and validation tasks of a UML formalization by establishing traceability links between the source and the target artifacts. Because of the code-distribution problem, the links between the source and the target artifacts may have variable arity, i.e., links with multiple endpoints. The distributed lines of target code manifested from a source feature are related in complex ways. In order to facilitate the verification and validation tasks, we should carefully design such links so that they capture the nontrivial relationships among

the target code. To this end, this section contributes a technique to reify the links as first-class objects. As a result, we can impose structures on links in order for them to convey information about the purpose of the different link endpoints, including how they coordinate to implement the feature being formalized.

Our approach to traceability builds a graph-based model of artifacts and links, an example of which appears in Fig. 3. We treat source and target models under formalization as labeled graphs whose nodes (depicted as roundtangles in the figure) correspond to instances of UML features and target-language features, respectively. Traceability links (depicted as diamonds in the figure) connect a node in a given source model to its manifestation (under formalization) in the corresponding target model. For example, a node in a source model might be an instance of a UML composite state; whereas a node in the corresponding target model might be an instance of a Promela `proctype` declaration. If the `proctype` declaration denotes the composite state under formalization, then this relationship could be represented explicitly by a traceability link connecting the node representing the composite state to the node representing the `proctype` declaration.

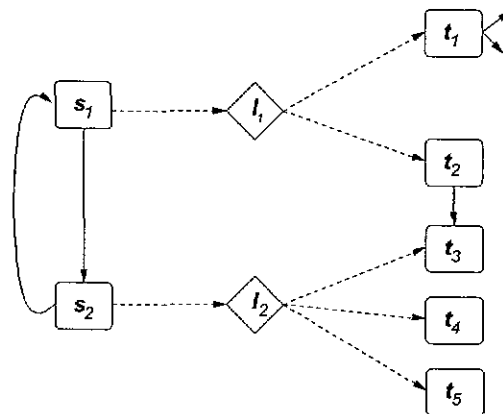


Fig. 3. Our model of source and target artifact graphs and links between them. Roundtangles labeled s and t refer to nodes in the source and target graphs respectively, whereas diamonds refer to links. Solid arrows refer to arrows within a given graph, whereas dashed arrows connect source models to links and links to target models.

Figure 3 depicts two important properties of links [21] in our model. First, links have *navigational directionality*, which means they may be traversed in only one direction (e.g., from source to target). Links that navigate from target back to source artifacts might also be useful in verifying the consistency of derived artifacts and in validating formalization strategies, but these are beyond the scope of this paper (see Sec. 6.4). Second, links have *variable arity*, which means that a link may have multiple endpoints. In fact, for many UML formalizations, the code-distribution problem makes links of variable arity the norm rather than the exception. Notably, Richardson and Green observe similar phenomena when linking source artifacts to targets that are derived using automated program synthesis [30]. In the UML

state-diagram formalization, links have *one-to-many multiplicity*, which is to say that each link connects exactly one source node to one or more target nodes. Given their importance, a key problem concerns how to represent and retrieve one-to-many links.

We take an object-oriented approach to modeling and representing artifact graphs, links, and their interconnections. The nodes and (directed) edges of an artifact graph are objects and object references (i.e., pointers) respectively. If a graph contains an edge (u, v) then the object corresponding to node u must contain a pointer to the object corresponding to node v . Within this framework, links are objects that may reference the target objects that correspond to nodes in the target graph. Consequently, the endpoints of a link are just references to target objects, and a link is connected to its endpoints by assigning these object references to appropriate fields within a link object. We model a one-to-many link as a *rooted network*^d of link objects. Thus, regardless of whether a link is one-to-one or one-to-many, there will always be a unique link object with which to connect a node in the source artifact graph. When the link is one-to-many, this unique object will be the root of the link-object network.

When representing the links, especially one-to-many links, we may have more than one choice. For example, recall (from Sec. 3) the formalization of transition t_f in Fig. 1. As summarized in Table 1, t_f manifests as six lines of distributed code, i.e., a root transition step in *proctype comp1* (line 17 in Fig. 2(a)), a leaf transition step and a bridge transition step in *proctype comp2* (lines 11 and 17 in Fig. 2(b)), and three leaf transition steps in *proctype comp3* (lines 12, 19, and 27 in Fig. 2(c)). For convenience, we call these six transition steps R , $L1$, B , $L2$, $L3$ and $L4$, respectively. These six transition steps have complex coordination relationships among them (Sec. 3.1.3). Intuitively, we may design the links as a simple, flat network comprising one link object that contains a variable-length sequence of references to the six Promela Option objects. However, in order to simplify the validation tasks, we prefer designs that explicitly capture the semantics of the complex relationships among the target objects implementing the source object. Figures 4(a) and 4(b) depict two different models of the one-to-many link that traces t_f to the six Option objects. Compared to the flat network in Fig. 4(a), the structured link model in Fig. 4(b) comprises a hierarchy of six link objects, each of which is associated with one of the endpoints of the link and two of which (i.e., $lnkR$ and $lnkB$) aggregate other link objects. As shown, the structured link model accurately captures the semantics among the endpoints of the link, including their purpose (e.g., the option implements a root, a bridge, a leaf, or a simple transition) and implicit coordination dependencies among them. Thus, we chose the structured link model instead of the trivial network in order to simplify the validation tasks of the formalization.

The decision of making links as first-class objects enables us to impose complex structures on the link objects. The ability of permitting structured designs is crucial,

^dIn our experience to date, these networks have actually been strict hierarchies.

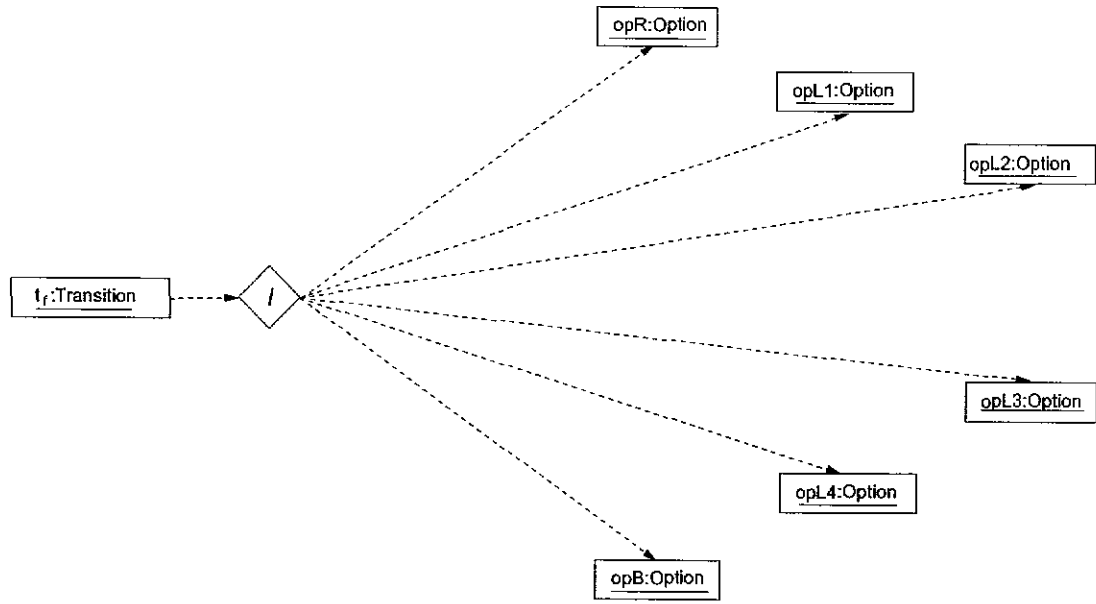
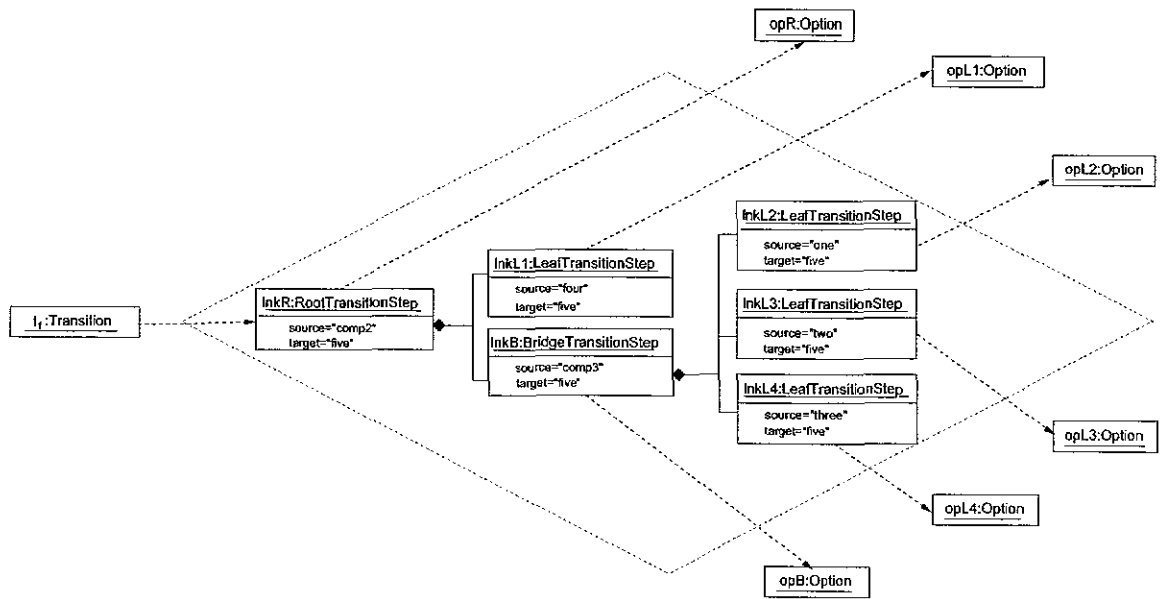
(a) A traceability link for transition t_f .(b) A structured traceability link for transition t_f .

Fig. 4. Two representations of the link between transition t_f in Fig. 1 and its manifestation in Fig. 2. t_f is a transition object; opR , $opL1$, opB , $opL2$, $opL3$ and $opL4$ are objects of Promela Option class that correspond to line 17 in Fig. 2(a), lines 11 and 17 in Fig. 2(b), and lines 12, 19 and 27 in Fig. 2(c), respectively; $lnkR$, $lnkL1$, $lnkB$, $lnkL2$, $lnkL3$ and $lnkL4$ are link objects. Dashed lines represent parts of a logical traceability link. The dashed diamond in (b) is a detailed view of the diamond in (a). It depicts the hierarchy of structures among the link objects.

when the endpoints of the link objects are related in complex ways. In order to simply the verification and validation tasks of the formalizations, we should carefully design the link models in order for them to reveal the nontrivial relationships among the endpoints. In practice, manually identifying the links between the complex extant source and target artifacts is not realistic and is error-prone.

5. Retrieval by Construction

We now describe our technique for retrieving traceability links between UML models and target models that are intended to denote them under formalization. Our technique exploits an alternative approach to reasoning about the *identity* of objects that allows object constructors to behave like object retrievers when the object(s) being constructed already exist. Thus, we use the term *Retrieval By Construction* (RBC) to describe the technique.

5.1. Value identity

Khoshafian and Copeland describe two fundamental approaches to reasoning about identity [14]. Using *object identity*, objects are inherently unique, even if they instantiate the same class and their attributes map to equivalent values. By contrast, using *value identity*, the identity of an object is a function of the values of its attributes. Object systems that use value identity have the useful property that a constructor behaves like a retriever if the object being constructed is value-equivalent to one that was previously constructed. For example, Fig. 5 depicts a group of three classes whose instances can be used to construct arithmetic expressions involving sums of integer literals. Lines (1) and (2) call for the construction of two identical expression trees. In an object system that uses value identity, the assert statement on line (3) will succeed; whereas the assertion will fail in an object system that uses object identity.

```

abstract class Expr;

class Add extends Expr is
  declare Expr left, right;
  constructor Add(Expr l, Expr r)
    is left, right := l, r end
end

class Literal extends Expr is
  declare int val;
  constructor Literal(int v)
    is val := v end
end

(1) Expr term1 := new Add( new Literal(2), new Literal(5) );
(2) Expr term2 := new Add( new Literal(2), new Literal(5) );
(3) assert term1 = term2;

```

Fig. 5. Example of the distinction between object and value identity. The assertion will fail under object identity and succeed under value identity.

Value identity is not directly supported in object-oriented languages, such as C++ and Java; however, there exist design patterns that can be applied to individual classes to make them simulate the use of value identity. Such a design utilizes the *singleton* [8] constructor interface, which precludes clients from directly instantiating a class, instead forcing them to request instances by invoking a static class method called `Instance()`. This idiom decouples the request for instantiation from the actual instantiation that typically fulfills such a request. By recording previously-allocated instances in static data structures, this `Instance()` method can ensure that any two requests to instantiate equivalent objects return the same object, thereby enforcing value identity. Applied systematically, this idiom precludes the generation of two value-identical linked object structures [36]. We use it to interpret a request to construct a target-language structure (e.g., the right-hand side expression on line (2) in Fig. 5) as a recipe for retrieving this structure if it exists, constructing a new one otherwise.

For a concrete example, consider an option in an `if-fi` statement in Promela, such as

```
:: msg == st_five -> goto five
```

on line 17 in Fig. 2(a). Our tools represent such an option internally as an instance of the class `Option`. (For the interested reader, this class and its relationships, as well as naming conventions, are described in the Appendix.) This class is instantiated with an instance of the class `Sequence`, which implements the internal representation of a sequence of steps; in this case, the sequence that contains a guard step followed by an action step:

```
msg == st_five -> goto five.
```

Instances of these classes should use value identity so that we can retrieve a reference to an existing option in some `if-fi` statement by requesting the construction of this object. Figure 6 depicts an elided version of the `Instance()` method. If an `Option` object with a `Sequence` object `s` does not exist in the hash table (memo) associated with the `Option` class, then a new `Option` object is constructed and returned. Otherwise, the reference to the existing object is returned.

| | |
|---|---|
| <pre> class Option is public static method Instance(Sequence) returns Option protected constructor Option(Sequence) static map memo[Sequence → Option] end </pre> | <pre> method Option::Instance (Sequence s) returns Option is if s ∉ keys(Option::memo) then Option::memo[s] := new Option(s) end if return Option::memo[s] end </pre> |
|---|---|

Fig. 6. Example of an `Instance()` method.

We found object systems that support value identity to be useful in implementing link-retrieval specifications. Again, referring to Fig. 5, if the expression tree referred to by `term1` already exists, then the expression on line (2) can be thought of as a retrieval specification, which when executed will return a reference to this tree. By contrast, if this tree did not already exist, then the execution of line (2) would construct it. We use this property to unify the definition of link-retrieval criteria with the definition of procedures for constructing the code that is denoted by a given UML feature under formalization. This duality has two practical benefits. First, these latter definitions (or some approximation of them) may be provided in the specification of a given UML formalization; thus the effort required to transform this knowledge into link-retrieval criteria is minimized. Second, we can build tools that collaborate with the object-management infrastructure to detect and help locate inconsistencies in a target artifact as a user attempts to retrieve links from the features in a given UML model. Because link endpoints are retrieved by construction, new target objects will be instantiated *only* if the correct endpoints cannot be retrieved from the existing target model. By monitoring these object-management structures, our tools detect the missing links and use the generated endpoints to help users locate the source of inconsistency in their target model.

5.2. Generative procedures

A generative procedure is a procedure for establishing a traceability link between an instance of some UML feature and the target-model manifestation(s) of this feature. These procedures are so-called because they retrieve the endpoints of a link, i.e., references to target-model entities, by attempting to explicitly generate these entities. For example, Fig. 7 depicts a GP that attempts to link a UML literal expression to the Promela constant expression that denotes it under formalization. The link itself is an object of class `PromLiteral`, which is constructed with a reference to the appropriate `ConstExpr` object. The request to retrieve this latter object is executed by invoking the `Instance()` method of class `ConstExpr`. Thus, if such an object already exists, then a reference to it will be returned; otherwise a new `ConstExpr` object will be created and returned. In either case, a link object is constructed. The link retrieval attempt is considered successful if no new target objects were constructed, and unsuccessful otherwise.

```

function traceToPromLiteral( Literal l )
  returns PromLiteral
is
  return new PromLiteral( ConstExpr::Instance( l.value ) );
end

```

Fig. 7. Example of a simple generative procedure.

```

function traceToRootTransitionStep( Transition t ) returns RootTransitionStep is
declare
(1)  string          targetName := t.target().name;
(2)  string          msgName   := "st_" + targetName;
(3)  seq [Step]      steps     := [ BinaryExpr::Instance( VarRef::Instance("msg"), "=",
(4)                                     VarRef::Instance(msgName) ),
(5)                                     GotoStmnt::Instance(targetName) ];
(6)  Option          endPoint   := Option::Instance( Sequence::Instance(steps) );

(7)  CompositeState  sourcePeer := peerState(t.source(), t.target());
(8)  seq [RemotePropagator] controlSrcs := mapcar ( traceToRemotePropagator,
(9)                                               distl(t,sourcePeer.substates));

in
(10) return new RootTransitionStep(sourcePeer.name, targetName, controlSrcs, endPoint);
end

```

Fig. 8. Example GP to retrieve a structured link.

The more interesting GPs are those used to retrieve structured links. For example, the GP in Fig. 8^e retrieves the link from a hierarchical UML transition, such as t_f in Fig. 1, to the collection of Promela Option objects that collaborate to simulate its behavior. Briefly, this structured link is constructed in three phases. Recall that a link from a hierarchical transition is a hierarchical link-object structure whose topmost node is a RootTransitionStep object. Lines (1) through (6) in Fig. 8 retrieve (by construction) the Promela Option object that serves as the endpoint of this topmost node of the structured link. Lines (7) through (9) construct the RemotePropagator link nodes that constitute the control sources of this RootTransitionStep. Finally, line (10) actually instantiates and returns the RootTransitionStep object.

Most of the details of this procedure are quite technical and not germane to this paper. Notice, however, that the invocation of this GP causes the invocation of subordinate GPs (traceToRemotePropagator, line (8) in Fig. 8), which collectively retrieve the structured sub-links needed to instantiate the RootTransitionStep link object. While not shown in the figure, each subordinate GP constructs a link object whose endpoint is retrieved by construction, just as is the endpoint of the RootTransitionStep link object. Consequently, as a side effect of the construction of this structured link, any of the endpoints that cannot be retrieved will be explicitly constructed. Thus, during a verification task, our tool may retrieve a structured link that is partially correct in that it links correctly to *some*, but *not all*, of the expected endpoints. As mentioned earlier, because our tools interact tightly with the object system, we can detect such inconsistencies and flag them. Moreover, when a structured link fails to retrieve an endpoint, it will generate what it was expecting to

^eThe `mapcar` utility applies a function to a list of elements. In this case, we have a list of pairs constructed by the `distl` utility applied to two arguments, x and seq . Each pair looks like (x, seq_i) , where seq_i is a member of seq . Both of these utilities are from functional programming languages.

find, and this generated endpoint will be couched in the context of other (correctly retrieved) endpoints. We believe this ability to detect artifact inconsistencies and to express the expected endpoints in context to be a key practical result of our technique to support verification and validation of UML formalizations.

6. Validation

This paper contributes a novel traceability technique (RBC) that is adapted to address the verification and validation problems associated with the formalization of UML models. We evaluated the feasibility of the technique by incorporating it into a proof-of-concept tool, called UBanyan. We then subjected it to case studies in order to assess its efficacy as a traceability technique. Specifically, we applied the tool to three case studies: one is the nested composite state example in Fig. 1, the second one is a version in which we injected inconsistencies, and the final one is a model of a controller for an automotive application obtained from industry [16]. We evaluated our results using the two well-known information retrieval metrics [7]: *precision* and *recall*. The preliminary investigation suggests that RBC can be useful as a link retrieval technique for traceability purposes.

6.1. UBanyan tool framework

We have prototyped the RBC technique in a tool framework called UBanyan^f to validate the use of generative procedures in retrieving links and to explore how traceability can support the various verification and validation activities inherent to UML formalization. The current prototype version of UBanyan traces UML diagrams into Promela models as specified by the UML-to-Promela formalization. Users must provide a source UML model, which they enter using the MINERVA graphical modeling environment [3], and a target Promela model, which is assumed to be in the form of a text file. The tool then retrieves links from instances of features in the source model to instances of features in the target model and generates marked-up Promela code listings and link reports.

The tool retrieves links in one of two modes, batch or interactive. The output generated from both modes make up the contents of a traceability report for the source and target artifacts. In *batch mode*, UBanyan attempts to identify inconsistencies by automatically tracing forward from each feature of the source model. The output is a *link report* comprising multiple sections, each of which contains information about links for a particular source object in isolation. The information includes the type of the source object, the name of the object, the type of the link object produced, the Promela object(s) at the endpoints of the link, and an indication of whether these endpoints were retrieved or constructed. If an endpoint

^fIn the name UBanyan, U stands for UML; Banyan is an East Indian fig tree. It can establish new roots from its branches, but it is always possible to trace the branches back to the original tree. Sometimes the branching can become quite complex, but there is always traceability.

```

1 *****
2 GENERATING A TRACEABILITY LINK for a UML transition object
3 ( comp2--f--> five )
4
5 BEGIN OF RootTransitionStep
6 This is a RootTransitionStep from comp2->five
7 The Promela Option object of this link object is:
8 :: msg == st_five -> goto five
9 retrieved from existing Promela objects
10
11 The subtree of link objects of this RootTransitionStep is shown as follows:
12 (1)-th subtree of the root:
13 This is a BridgeTransitionStep from comp3->five
14 The Promela Option object of this link object is:
15 :: msg == st_five -> wait! _pid,st_five; goto exit
16 retrieved from existing Promela objects
17
18 The subtree of link objects of this BridgeTransitionStep is shown as follows:
19 *1*-th subtree of the bridge:
20 This is a leaf transition from one->five
21 The Promela Option object of this link object is
22 :: atomic { ex_q?[f] -> ex_q?f }; wait! _pid,st_five; goto exit
23 retrieved from existing Promela objects
24
25 *2*-th subtree of the bridge:
26 This is a leaf transition from two->five
27 The Promela Option object of this link object is
28 :: atomic { ex_q?[f] -> ex_q?f }; wait! _pid,st_five; goto exit
29 retrieved from existing Promela objects
30
31 *3*-th subtree of the bridge:
32 This is a leaf transition from three->five
33 The Promela Option object of this link object is
34 :: atomic { ex_q?[f] -> ex_q?f }; wait! _pid,st_five; goto exit
35 retrieved from existing Promela objects
36
37 (2)-th subtree of the root:
...
43 END OF RootTransitionStep

```

Fig. 9. The link report for consistent artifacts: the fragment for t_f .

must be constructed rather than retrieved, then UBanyan reports an inconsistency between the artifacts. Figure 9 depicts a portion of the link report generated when a batch-mode analysis is applied to the UML model in Fig. 1 and the Promela model in Fig. 2. This portion reports information about the link from transition t_f . A RootTransitionStep link object is produced for t_f , and the corresponding Promela Option object was retrieved. This RootTransitionStep link object comprises a subtree of subordinate link objects and their information is also given. From the highlighted lines (i.e., lines 9, 16, 23, 29 and 35), we see that all the Promela objects for the link objects for t_f were retrieved, which means that this link does not indicate an inconsistency between the artifacts. In contrast, Fig. 10 shows the portion of the link report for t_f when the source and target artifacts are inconsistent. Lines 23,

```

1 *****
2 GENERATING A TRACEABILITY LINK for a UML transition object
3 ( comp2--f-> five )
4
5 BEGIN OF RootTransitionStep
6 This is a RootTransitionStep from comp2->five
7 The Promela Option object of this link object is:
8 :: msg == st_five -> goto five
9   retrieved from existing Promela objects
10
11 The subtree of link objects of this RootTransitionStep is shown as follows:
12 (1)-th subtree of the root:
13 This is a BridgeTransitionStep from comp3->five
14 The Promela Option object of this link object is:
15 :: msg == st_five -> wait!_pid,st_five; goto exit
16   retrieved from existing Promela objects
17
18 The subtree of link objects of this BridgeTransitionStep is shown as follows:
19 *1*-th subtree of the bridge:
20   This is a leaf transition from one->five
21   The Promela Option object of this link object is
22     :: atomic { ex_q?[f] -> ex_q?f }; wait!_pid,st_five; goto exit
23     ***CONSTRUCTED DURING TRACEABILITY LINK GENERATION***
24
25 *2*-th subtree of the bridge:
26   This is a leaf transition from two->five
27   The Promela Option object of this link object is
28     :: atomic { ex_q?[f] -> ex_q?f }; wait!_pid,st_five; goto exit
29     ***CONSTRUCTED DURING TRACEABILITY LINK GENERATION***
30
31 *3*-th subtree of the bridge:
32   This is a leaf transition from three->five
33   The Promela Option object of this link object is
34     :: atomic { ex_q?[f] -> ex_q?f }; wait!_pid,st_five; goto exit
35     ***CONSTRUCTED DURING TRACEABILITY LINK GENERATION***
36
37 (2)-th subtree of the root:
38 ...
43 END OF RootTransitionStep

```

Fig. 10. The link report for inconsistent artifacts: the fragment for t_f .

29 and 35 show that the target objects were constructed during the link-retrieval process, an indication of inconsistency.

In *interactive mode*, the user selects instances of source-model features and asks the system to trace forward into the target model. In response, the system constructs a link from the selected feature(s) and highlights the endpoints that were retrieved, flagging any instances of endpoints that were constructed and thus not retrieved. Two forms of output are produced in this mode. As with the batch mode, a link report is generated. In addition, an HTML document representing the target Promela code is produced, where the relevant target elements corresponding to the selected source-model features are highlighted. The screenshots in Figs. 11 and 12 depict this capability in operation when UBanyan is applied to the UML model in Fig. 1 and the Promela model in Fig. 2. In this example, the user has selected the transition t_f . UBanyan responds by attempting to construct a link from this selected feature to endpoints in the Promela model. If the link can be retrieved,

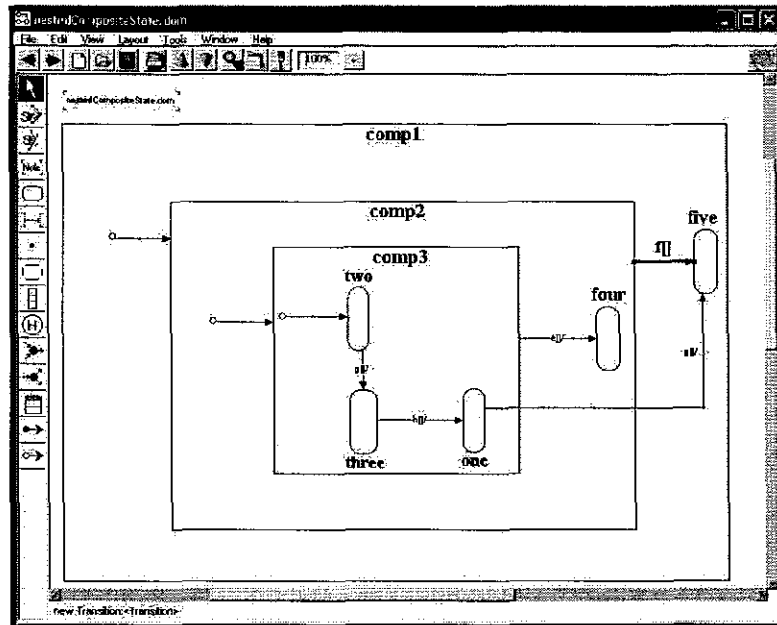


Fig. 11. The UML state diagram with t_f selected.

```

to_comp2: comp2_pid=run_comp2(msg);
wait??eval(comp2_pid), msg;
if
  :: msg==st_five; goto five
fi;
exit: skip;
}
proctype comp2(mtype state)
{
  int comp2_pid;
  int comp3_pid;
  mtype msg;
  int dummy;
  goto to_comp3;
  four:
  if
    :: atomic { ex_q?[f] -> ex_q?f }; wait!_pid,st_five; goto exit
  fi;
  to_comp3: comp3_pid=run_comp3(msg);
  wait??eval(comp3_pid), msg;
  if
    :: msg==st_five; wait!_pid,st_five; goto exit
    :: msg==st_four; goto four
  fi;
  exit: skip;
}
proctype comp3(mtype state)
{

```

Fig. 12. The HTML file produced for t_f with relevant Promela code highlighted.

then its endpoints are marked in a listing of the Promela model that is generated and presented to the user. For readability purposes, retrieved endpoints are highlighted in color and in a distinct font (e.g., red, bold, and italicized). Because the target model in this example correctly implements the source feature according to the given formalization, the appropriate lines in the generated listing have been marked. Had the retrieval of the link failed, the tool would report the error as a potential inconsistency between the two artifacts.

The two modes serve different purposes relative to verification and validation activities. Batch mode is probably best suited to verification, at least initially when a designer wishes to verify that a target model is a correct formalization of a given UML model. Inconsistencies (i.e., cases in which target objects are constructed rather than retrieved) indicate either errors of omission, designer confusion, or clerical errors. A batch-mode analysis is thus a starting point for discovering inconsistencies. Once discovered, the designer can revert to interactive mode to trace forward from the feature(s) that led to the inconsistency. The fact that the expected endpoints are generated when they cannot be retrieved may provide useful context with which a designer can track down the cause of the problem. This is especially true in the case of structured links.

With respect to validation tasks, interactive is the mode of choice, as it allows a potential user to see the effects of formalization decisions on concrete and familiar examples. To this end, inconsistencies highlight potential problems with the formalization itself.

6.2. *UBanyan architecture*

Figure 13 depicts the data flow architecture of UBanyan. Using the conventions of data flow diagrams (DFDs), ellipses represent processes, parallel lines represent data stores, arrows are labeled with data flows, and external entities are represented by rectangles. The tool is built around three in-memory object repositories: the *UML Model Object Store*, the *Promela Model Object Store*, and the *Unaccounted Object Store*. The UML Model Object Store is simply an object-oriented representation of a *Source UML Model*. This store is populated by a *Source Parser*, which parses the MINERVA-generated textual representation of the source model.

The other two internal stores comprise object-oriented representations of target-model (i.e., Promela) code. Specifically, the *Promela Model Object Store* comprises objects that represent some *Target Promela Model*, which is read in by a *Target Parser*. The *Unaccounted Object Store* comprises any Promela objects that are constructed because they do not exist in (and thus cannot be retrieved from) the *Promela Model Object Store*. This store is populated by generative procedures that are invoked by the *Link Retriever* in response to a request to generate a link from a given object in the *UML Model Object Store*. All of the classes that implement Promela-model objects use value identity.

The external data store, *Traceability Report*, contains the traceability information produced by UBanyan. The traceability report comprises two major elements, a *Link report* and a color-coded *HTML document* of the target-model code. The *Link report* is generated for both interactive and batch-mode operations. In contrast, the *HTML document* is only produced as a result from using UBanyan in interactive mode. Details about their generation are described next.

The most important process in this architecture is the *Link Retriever*. For both interactive and batch-mode operations, the *Link Retriever* constructs a link object, attempting to retrieve the endpoints in the *Promela Model Object Store*, if possible. The difference is that in interactive mode, only the link for the Promela object structure corresponding to a user-selected UML object structure is retrieved. In batch mode, the links for all of the UML object structures in a given Source UML Model are retrieved. In both interactive and batch modes, the *Link Retriever* marks all retrieved target-model objects with a *Visit flag*, which is used for the *Link report* and the HTML document. Target objects marked as visited will be displayed as “retrieved from existing Promela objects” in the *Link report* (as shown in Fig. 9). In the case of interactive mode, the *Visit flag* will indicate which target-model code should be pretty printed in an *HTML document* using special colors and fonts to communicate their status as endpoints of some traceability link for a selected UML object structure (as shown in Fig. 12). Endpoints that cannot be retrieved are constructed and stored in the *Unaccounted Object Store* repository; in addition to the newly generated Promela code structure, an appropriate message is added to the *Link report* of the form “*** CONSTRUCTED DURING TRACEABILITY LINK GENERATION***” (as shown in Fig. 10).

The *Link report* produced by UBanyan provides visual feedback to a potential user of a UML formalization regarding the consistency and validity of a UML formalization. Currently, only a textual format is used to convey forward traceability information. The UBanyan architecture is not specific to the source or target languages. We envision that other source and target languages could be “plugged-in”, as long as parsers for the respective languages can be implemented and the appropriate GPs can be constructed to implement the *Link Retriever*.

6.3. Case studies

We evaluated our traceability technique using the standard information retrieval metrics: precision and recall. While the objectives and artifacts of the various traceability analyses and the specific mechanisms may be different when considering one traceability technique to another, the definitions for precision and recall metrics are the same. As such, upon reviewing the precision and recall traceability results for our three case studies and comparing them to related traceability work, we find the RBC technique to be promising.

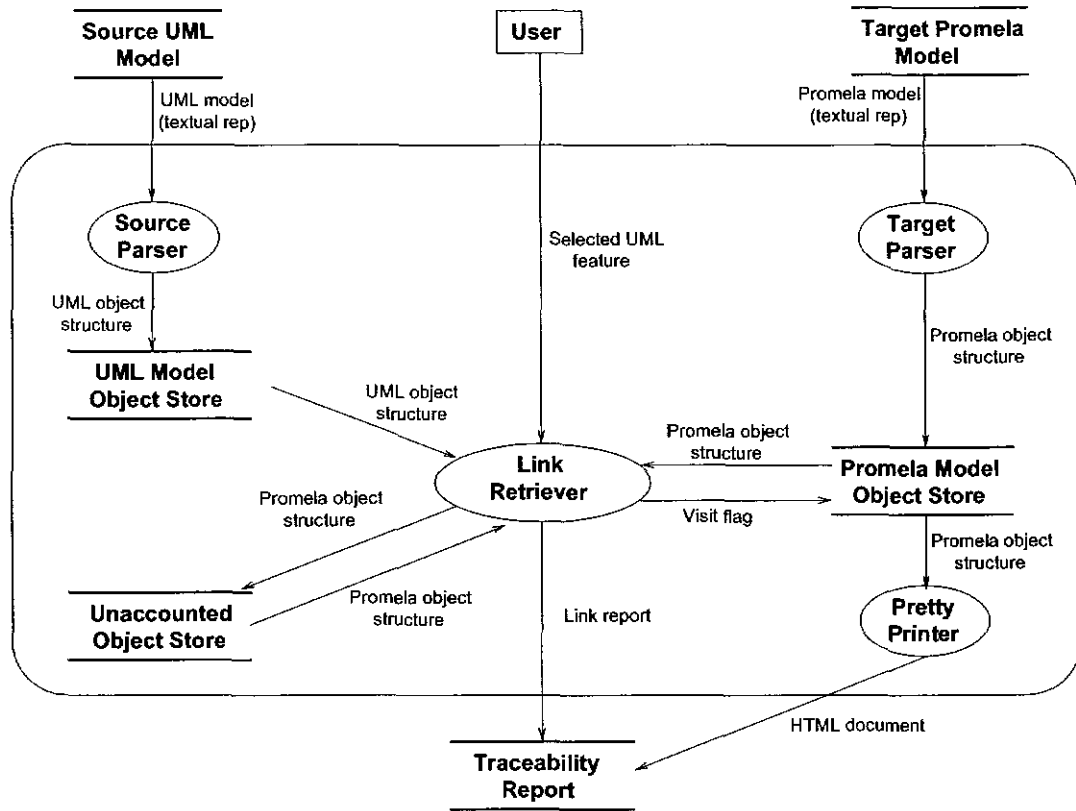


Fig. 13. Data flow diagram (DFD) of UBanyan.

6.3.1. Precision and recall

Precision and *recall* [7] are two commonly used metrics to evaluate the utility of traceability techniques [1, 12, 22, 38, 30]. Precision is the ratio of the number of correct links among the retrieved links to the total number of the retrieved links. In contrast, recall is the ratio of the number of retrieved correct links to the total number of correct links. That is,

$$\text{Precision} = \frac{\text{retrieved correct links}}{\text{total retrieved links}}, \quad \text{Recall} = \frac{\text{retrieved correct links}}{\text{total correct links}}$$

Theoretically, our RBC technique should encode the GPs such that the target endpoints are expected to be correctly retrieved when two artifacts are consistent. The assumption is that for a given UML formalization, each UML feature has a formalization rule that defines its semantics in isolation (which can then be encoded as a GP). In other words, when the source and target artifacts are consistent, the precision value should be 100%. In addition, if we have GPs for every type of source feature, then the recall value should also be 100%. We now present the experimental precision and recall values on the nested composite state example (Fig. 1) and an industrial application [16].

6.3.2. The nested composite state example

As shown in Fig. 1, the nested composite state example contains 3 composite states, 5 simple states, 3 initial pseudostates, 5 transitions and 4 events. The target Promela artifact in use is in Fig. 2, which has approximately 70 lines of code. The two artifacts are consistent. We had GPs for all these source features. Regardless of whether a link is structured or not, more than one target language endpoint may be retrieved. To be more representative of the retrieval process, we report the retrieval of endpoints, rather than links because a structured link may involve a hierarchy of endpoints. For this example, we expect to retrieve 36 endpoints. The 36 endpoints were successfully identified and retrieved by UBanyan and no spurious links were retrieved. As a result, the precision and recall values are both 100% in this case. The results are summarized in the first line of Table 2.

A second case study tests whether UBanyan can identify inconsistent artifacts. Assume there were no automatic translator for the UML-to-Promela formalization, and a user had to construct the Promela specification manually. Further assume that the Promela specification is developed correctly, except the user misunderstands how to formalize a UML event object in terms of Promela code. The resulting erroneous Promela code affects the lines of code in Figs. 2(b) and (c) that relate to the specification of a UML event object.

As indicated by the code in Fig. 2 (e.g., line 11 in (b)), the formalization assumes that before firing a transition, we check that the event is in the designated queue (`comp1_q?[event]`). If so, then we consume the event from the queue (`comp1_q?event`) and then fire the transition. The lines of code that include (`comp1_q?event`) are those that are relevant to the formalization of a UML event object. As such, it is precisely these corresponding lines of code that are erroneously specified manually. Compared to the code in Fig. 2 that is correctly specified with respect to the formalization, each erroneous line was missing the test for an event in the queue (i.e., `comp1_q?[*]`, where ‘*’ represents the events, e, f, g, h, respectively in Fig. 2(b), line 11, and 2(c), lines 11–13, 18–20, 25–27). This subtle type of specification error would have been quite difficult to detect by inspection.

The purpose of this study was to test whether UBanyan can detect the inconsistencies between the UML nested composite state example and the manually produced Promela specification. Another objective was to explore what kind of feedback can be provided for locating the source of the inconsistency. The precision and recall results for retrieving endpoints are shown in the second line of Table 2. The precision value is still 100% as in the previous study, but the recall value drops to 41.67%. We have identified two potential reasons for this lower recall value. First, our generative procedures are so precisely encoded that any mismatch between the expected and the extant target objects will cause the retrieval process to miss those links whose endpoints are the extant target objects. Second, the source object may aggregate a number of parts, where each part may involve a structured link; each structured link may correspond to multiple endpoints. Therefore, if a (structured)

link for one of the aggregate elements cannot be retrieved, then the link for the aggregate object cannot be retrieved either. As such, missing even a single (structured) link for a given UML feature that is part of an aggregate structure (as is the case when a feature is incorrectly formalized) may actually result in an entire hierarchy of endpoints being missed. In this case, the code for implementing transitions comprises the code for events, and the code for implementing simple states comprises the code for transitions. As a result, the links and their respective endpoints are missing not only for events, but also for transitions and simple states.

6.3.3. The diesel filter system example

Our third case study applies to a UML model for an automotive application obtained from industry. The target Promela code was generated using Hydra [3]. The diesel filter system (DFS) [16] is an embedded system that controls a self-cleaning particulate filter that reduces the amount of pollutants emitted from the exhaust of diesel trucks. The class diagram of this model contains 13 classes, each of which is associated with a state diagram. The total number of endpoints is 479, of which 96 are for simple states, 17 are for composite states, 15 are for initial states, 181 are for transitions, and 122 are for events. The test results are summarized in line 3 of Table 2. In this case study, the precision result is again 100%, and the recall value is 88.73%.

Table 2. Test results of the diesel filter system case study.

| Case study | Total endpoints expected | Correct endpoints retrieved | Incorrect endpoints retrieved | Total endpoints retrieved | Missed endpoints | Precision (%) | Recall (%) |
|------------------|--------------------------|-----------------------------|-------------------------------|---------------------------|------------------|---------------|------------|
| consistent_NCS | 36 | 36 | 0 | 36 | 0 | 100 | 100 |
| inconsistent_NCS | 36 | 15 | 0 | 15 | 21 | 100 | 41.67 |
| DFS | 479 | 425 | 0 | 425 | 54 | 100 | 88.73 |

6.4. Discussion

Our precision numbers are consistently 100%, which we attribute to the inherent formality of the relationship between the source and target artifacts. Most other traceability techniques apply to problems for which such a degree of formality is not attainable. Indeed, the precision/recall results of the IR-based approaches [1, 12, 22] suggest an inherent tradeoff between precision and recall in which precision can be gained at the expense of recall and vice versa. Our recall numbers suggest that there is room for improvement, and we are considering a number of approaches to improve them. We speculate that there may be alternative target-model representations — something akin to the *normal forms* used to represent propositional formulae in automated theorem provers — that better reflect semantic equivalences in the syntactic structure of the target-model objects. It is conceivable that one could

improve recall by choosing a “better” representation for the target-model objects, and it is an open question as to whether doing so will affect the precision of our approach.

One factor that can influence the precision number is the degree to which our generative procedures encode so-called *source context* in the target objects. This is best explained by example. Suppose a UML diagram contains two separate composite states, each of which contains simple states named s_1 and s_2 , and each of which contains a transition from s_1 to s_2 on some event e . Under the formalization described in this paper, these simple states and the transitions will map to identical blocks of Promela code in two different proctype declarations. Because target objects are represented using value identity, those contextually distinct but value identical blocks of code will be indistinguishable from one another. Now suppose the user attempts to trace the Promela code generated from one of these s_1 states. When the link is constructed, the aforementioned target object will be retrieved and marked; however marking an occurrence of the block in one proctype declaration effectively marks it in the other. Thus, the color-coded HTML document that lists the link endpoints will have color-coded superfluous blocks of code, i.e., it will have reported a traceability link that does not exist. This problem does not arise in our examples because the rules from which we derived our generative procedures encode sufficient source context into the generated target-model objects. It is an open question as to whether GP-encodings of other UML formalizations will exhibit a similar high degree of precision.

A related issue concerns the impact of using value identity on the ability to traverse links backward from target to source. Backward traceability is not possible given our current design. The major reason for this constraint is that the formalization was defined to identify target features that define the semantics of a given UML feature. The formalization was not, however, designed to be a UML-based graphical front-end for the target language, nor was the formalization intended to be a target-language generator. As such, there are no explicitly defined relationships for moving from the target model back to the source model. Therefore, the nature of our formalization was strictly defined for forward traceability. Future research will explore how the link objects can encode both forward and backward traceability information.

7. Conclusions and Future Work

With the increasing interest in using and formalizing UML, the verification and validation of UML formalizations become increasingly necessary; yet we are unaware of support from the formalization community for this task. We see verification as fundamentally a traceability problem and validation as a problem that can be supported with sophisticated traceability tools and methods. This paper investigates the nature of the traceability problem in UML formalization.

Our technique, called retrieval by construction (RBC), incorporates two ideas. First, whereas one-to-many links seem inherent to artifacts related under UML formalization, we represent these links in a form that captures important semantic relationships among target-model entities and that elucidates key design decisions that underlie a given formalization. Second, whereas generative procedures are the most natural means of specifying UML formalization, we use them to specify criteria for retrieving links. A GP precisely specifies how to construct a link (including the construction of its endpoints) for a given UML object. Our use of GPs as link retrievers assumes that target models are represented in an object system that uses value identity as opposed to object identity. Such an object system is easily implemented using relational databases, and can be implemented idiomatically using the singleton design pattern. We have implemented the RBC technique in the UBanyan tool framework.

In contrast to most traceability approaches in the literature that deal with relating two artifacts where their relationships cannot be precisely specified (e.g., [1, 12, 22, 38]), our approach is applicable when there is a systematic and repeatable approach for deriving target artifacts from source artifacts. In the example formalization used in this paper, we were able to capture these repeatable design decisions using generative procedures and by explicitly modeling the structure of links (see, for example, the metamodel for links from UML transition objects in Fig. 16). We are investigating the extent to which explicit modeling of structured links can capture key design decisions in other UML formalizations. An interesting question concerns how well the approach can retrieve links when the relationship between source and target artifacts is more complex. At the extreme, we do not expect the technique to apply when the relationship between source and target artifacts involves novel and/or complex design decisions, such as data refinements or algorithm design.

This work builds on the assumption that traceability techniques can support the verification and validation of UML formalizations. We envision in particular RBC supporting these tasks as follows. First, with regards to verification, for each given UML object, we can invoke a generative procedure to retrieve a link into the target model. If the endpoints of this link are unexpected, then we may have identified an inconsistency between the source and target models, in which case some corrective action should be performed. Some of these potential inconsistencies can be detected automatically, e.g., if in the process of “retrieving” a link, instead of retrieving references to extant target-model objects, *new* target-model objects are unexpectedly constructed. Second, with RBC, we can provide visual traceability links between the source and target models and thus trace forward from UML to the target language to see the code implementing each UML feature. This process can be used to verify the correctness of the implementation of a translator and validate that the intended semantics have been captured. In addition, the collection of generative procedures may serve to document the design decisions made in a

given formalization, and they may even do so in a manner that is more precise and complete than the formalization rules.

Regarding future work, we plan to pursue the following directions. First, we believe that link-object models, which record the structure of links associated with a given UML feature, may find use as artifacts in validation activities. An interesting question concerns the extent to which link-object modeling can inform and guide the validation task itself. Given that the validation task has much in common with reverse engineering, which is inherently a model-building activity [5], there is much to be investigated here. In addition, we plan to investigate how to address backward traceability from a target-model feature to the UML features that contributed to its existence, which may enable the identification of errors in the UML diagram based on the analysis of the corresponding target code. We are also looking at novel applications of traceability beyond the verification and validation tasks already discussed. One example is the use of traceability to help analysts refine the specification of safety and liveness properties over their UML diagrams into analogous specifications over the (formal) target model. Finally, it would be interesting to explore whether IR-based traceability techniques can play a role in the verification and validation activities of UML formalizations.

Acknowledgments

This work has been funded in part by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CCR-9901017, and CCR-9984727, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and Eaton Corporation, and in cooperation with Siemens Automotive and Detroit Diesel Corporation.

Appendix A

To illustrate our techniques in Sec. 5, we provided pseudocode that references attributes and operations of the objects that represent UML features, Promela features, and links. For readers who are interested in these details, we now describe the structure of these objects and the space of available operations. For brevity, we depict this information graphically, using *metamodels*, which are UML class diagrams that describe the syntax of a modeling notation [27].

Classes in a metamodel are related by *generalizations*, which specify class inheritance, and *navigation associations*, which specify the ability to retrieve references to objects of one class from objects of another. UML provides several different kinds of associations (e.g., aggregation, composition, and general associations); however these distinctions are not salient in this paper. We use these metamodels to infer the names and signatures of the operations that are invoked in the pseudocode of Sec. 5 (Figs. 5–8). For example, we can infer from Fig. 14 that class `Transition` must export at least four navigation operations (`source()`, `target()`, `event()`, and `guard()`). We can also infer the return types. For example, the operation `Transition::source()` must return a reference to a `StateVertex` object. As the

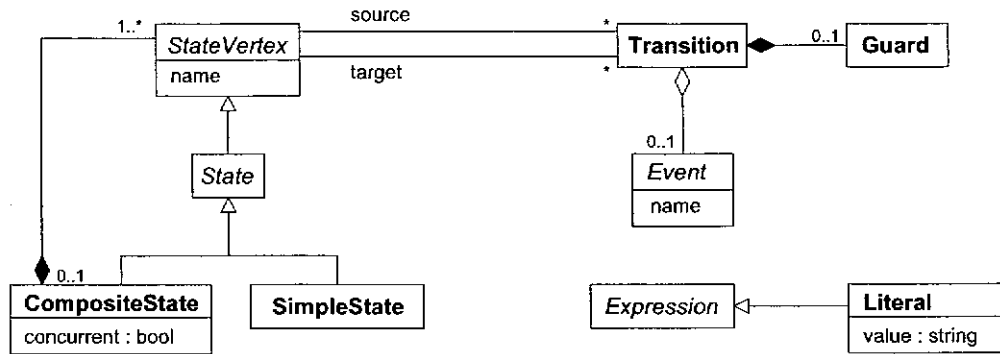


Fig. 14. Excerpted metamodel for UML representation.

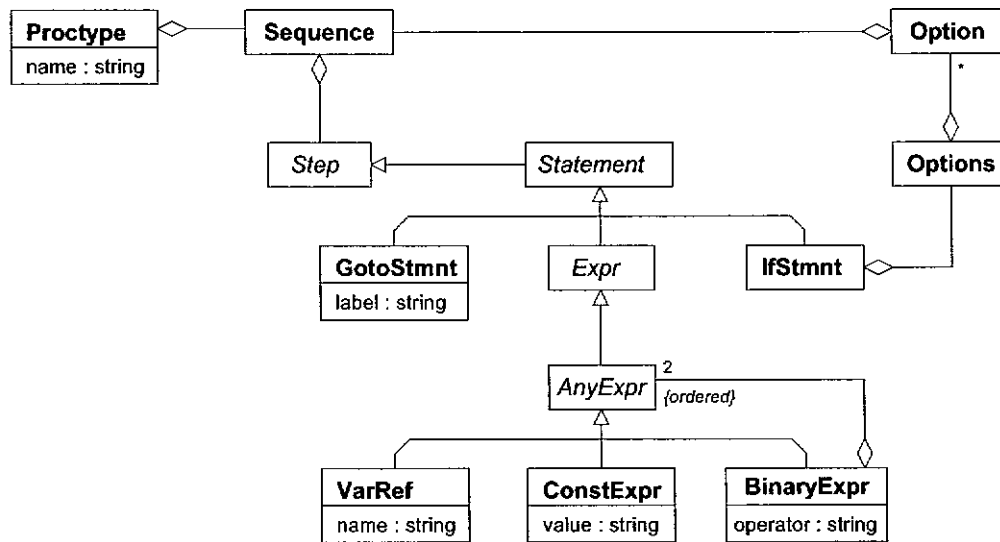


Fig. 15. Excerpted metamodel for Promela representation.

metamodels depicted here are excerpted from the actual metamodels that describe our implementation classes, many of the classes may export additional operations whose names and signatures cannot be inferred from these figures.

Figure 16 depicts the classes used to represent structured links from UML transitions. This model uses abstract classes to encode the link-relevant classification of transition steps into direct vs. indirect observers and local vs. remote propagators. These concepts are joined (using multiple inheritance) to form the concrete classes used to instantiate link objects. Thus, for example, a *RootTransitionStep* is both a local propagator and an indirect observer. The transition steps with which it must coordinate, i.e., the *control sources*, are remote propagators that correspond to *BridgeTransitionSteps* and/or *LeafTransitionSteps*. This relationship is captured by the aggregation relationship between *IndirectObserver* and *RemotePropagator*. A *BridgeTransitionStep* in a parent process is both a remote propagator and an indirect observer, which should coordinate with other *BridgeTransitionSteps* and

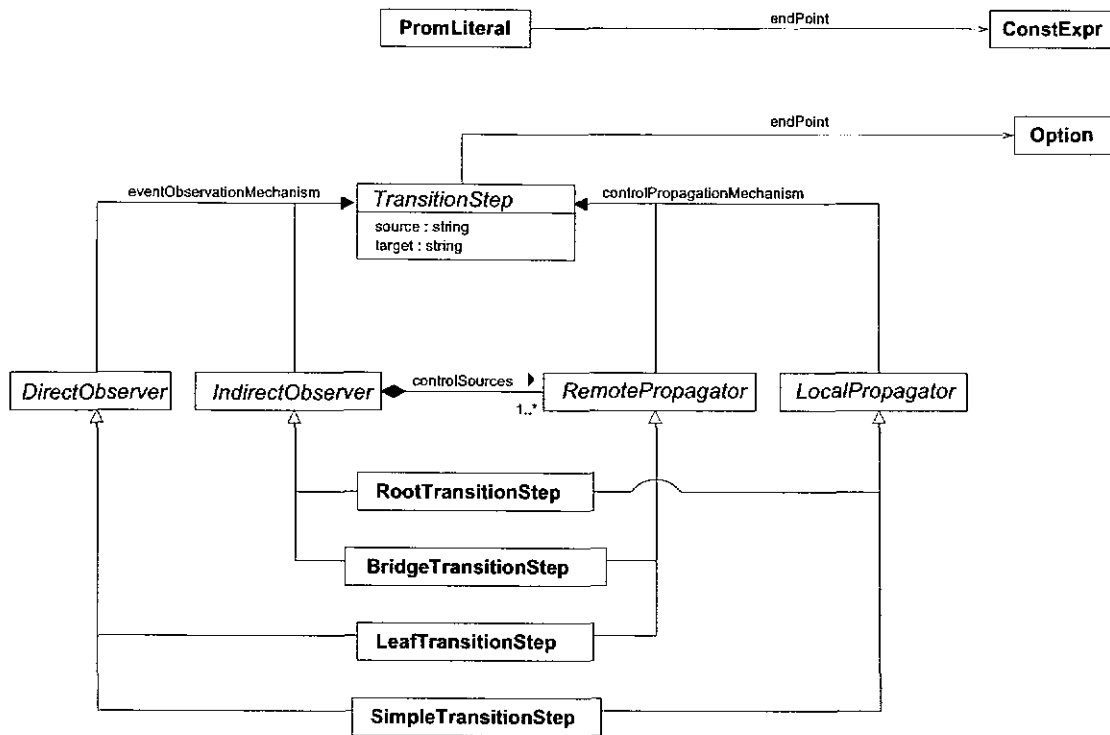


Fig. 16. Metamodel for links from UML literal objects and metamodel for links from UML transition objects.

LeafTransitionSteps in its child processes. LeafTransitionStep is both a direct observer and a remote propagator. SimpleTransitionStep is both a direct observer and a local propagator. For example, the simple transition t_h is formalized by a Promela option we call a simple transition step (line 25 in Fig. 2(c)).

References

1. G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merio, Recovering traceability links between code and documentation, *IEEE Trans. on Software Engineering* **28**(10) (2002).
2. J. Boyle, R. Resler, and V. Winter, Do you trust your compiler? *IEEE Computer* **32**(5) (1999) 65–73.
3. L. A. Campbell, B. H. C. Cheng, W. E. McUmbert, and R. E. K. Stirewalt, Automatically detecting and visualizing errors in UML diagrams, *Requirements Engineering Journal* **37**(10) (2002).
4. B. H. Cheng and E. Y. Wang, Formalizing and integrating the dynamic model for object-oriented modeling, *IEEE Trans. on Software Engineering* **28**(28) (2002) 747–762.
5. E. J. Chikofsky and J. H. Cross, Reverse engineering and design recovery: a taxonomy, *IEEE Software* **7**(1) (1990) 13–17.
6. A. Egyed, A scenario-driven approach to trace dependency analysis, *IEEE Trans. on Software Engineering* **29**(2) (2003).
7. W. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, 1992.

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
9. H. J. Goldsby, Specifying compositional semantic functions for non-hierarchical languages using natural deduction systems, Master's thesis, Michigan State University, May 2004.
10. O. C. Z. Gotel and A. C. W. Finkelstein, An analysis of the requirements traceability problem, in *Proc. 1st Int. Conf. on Requirements Engineering (ICRE'94)*, 1994.
11. D. Harel and O. Kupferman, On object systems and behavioral inheritance, *IEEE Trans. Software Eng.* **28**(9) (2002) 889–903.
12. J. H. Hayes, A. Dekhtyar, and J. Osborne, Improving requirements tracing via information retrieval, in *Proc. 11th IEEE Int. Requirements Engineering Conference*, 2003.
13. G. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.
14. S. N. Khoshafian and G. P. Copeland, Object identity, in *Conference Proc. on Object-Oriented Programming Systems, Languages and Applications*, ACM Press, 1986, pp. 406–416.
15. S.-K. Kim and D. A. Carrington, A formal denotational semantics of UML in Object-Z, *L'OBJET: Software, Databases, Networks* **7**(1) 2001.
16. S. Konrad, L. A. Campbell, B. H. Cheng, and M. Deng, A requirements patterns-driven approach to specify systems and check properties, in T. Ball and S. K. Rajamani, eds., *Model Checking Software*, in Lecture Notes in Computer Science, Vol. 2648, Springer Verlag, 2003, pp. 18–33.
17. D. Latella, I. Majzik, and M. Massink, Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker, *Formal Aspects of Computing* **11**(6) (1999) 637–664.
18. L. Lavazza, G. Quaroni, and M. Venturelli, Combining UML and formal notations for modelling real-time systems, in *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering (FSE)*, 2001.
19. J. Lilius and I. Porres, The semantics of UML state machines, Technical Report 273, Turku Center for Computer Science (TUUS), Abo Akademi University, Turku, Finland, 1999.
20. G. Lüttgen, M. von der Beeck, and R. Cleaveland, A compositional approach to statecharts semantics, in *Proc. of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000)*, 2000.
21. J. I. Maletic, E. V. Munson, A. Marcus, and T. N. Nguyen, Using hypertext model for traceability link conformance analysis, in *Proc. 2nd Int. Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*, October 2003.
22. A. Marcus and J. I. Maletic, Recovering documentation-to-source-code traceability links using latent semantic indexing, in *Proc. IEEE Int. Conf. on Software Engineering*, May 2003.
23. K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
24. W. McUmbler, *A Generic Framework for Formalizing Object-Oriented Modeling Notations for Embedded Systems Development*, PhD thesis, Michigan State Univ., August 2000.
25. W. McUmbler and B. H. Cheng, A general framework for formalizing UML with formal languages, in *Proc. IEEE Int. Conf. on Software Engineering*, May 2001.
26. G. C. Murphy, D. Notkin, and K. J. Sullivan, Software reflexion models: Bridging the gap between design and implementation, *IEEE Trans. on Software Engineering* **27**(4) (2001) 364–380.

27. Object Management Group, UML Specifications, Version 1.5, March 2003.
28. OMG, UML Specification V1.5, <http://www.omg.org/uml>.
29. Promela language reference, <http://spinroot.com/spin/Man/promela.html>.
30. J. Richardson and J. Green, Automating traceability for generated software artifacts, in *Proc. IEEE Int. Conf. on Automated Software Engineering (ASE'04)*, September 2004.
31. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
32. M. Shroff and R. France, Towards a formalization of UML class structures in Z, in *Proc. COMPSAC'97*, 1997.
33. K. Slonneger and B. L. Kurtz, *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley, 1995.
34. G. Smith, *The Object-Z Specification Language: Advances in Formal Methods*, Kluwer Academic Publishers, 2000.
35. On-the-fly, LTL model checking with SPIN, <http://spinroot.com/spin/whatispin.html>.
36. R. E. K. Stirewalt and L. K. Dillon, Generation of visitor components that implement program transformations, in *ACM SIGSOFT Symposium on Software Reusability*, 2001.
37. M. C. Tanuan, Automated analysis of unified modeling language (UML) specifications, Master's thesis, Univ. of Waterloo, August 2001.
38. A. Zisman, G. Spanoudakis, E. Peres-Minana, and P. Krause, Tracing software engineering artifacts, in *Proc. 2003 Int. Conf. on Software Engineering Research and Practice (SERP'03)*, 2003.