TRANSIT: Specifying Protocols with Concolic Snippets *

Abhishek Udupa Arun Raghavan Jyotirmoy V. Deshmukh†
Sela Mador-Haim Milo M. K. Martin Rajeev Alur
University of Pennsylvania
{audupa, arraghav, djy, selama, milom, alur}@cis.upenn.edu

Abstract

With the maturing of technology for model checking and constraint solving, there is an emerging opportunity to develop programming tools that can transform the way systems are specified. In this paper, we propose a new way to program distributed protocols using *concolic* snippets. Concolic snippets are sample execution fragments that contain both concrete and symbolic values. The proposed approach allows the programmer to describe the desired system partially using the traditional model of communicating extended finite-state-machines (EFSM), along with high-level invariants and concrete execution fragments. Our synthesis engine completes an EFSM skeleton by inferring guards and updates from the given fragments which is then automatically analyzed using a model checker with respect to the desired invariants. The counterexamples produced by the model checker can then be used by the programmer to add new concrete execution fragments that describe the correct behavior in the specific scenario corresponding to the counterexample.

We describe TRANSIT, a language and prototype implementation of the proposed specification methodology for distributed protocols. Experimental evaluations of TRANSIT to specify cache coherence protocols show that (1) the algorithm for expression inference from concolic snippets can synthesize expressions of size 15 involving typical operators over commonly occurring types, (2) for a classical directory-based protocol, TRANSIT automatically generates, in a few seconds, a complete implementation from a specification consisting of the EFSM structure and a few concrete examples for every transition, and (3) a published partial description of the SGI Origin cache coherence protocol maps directly to symbolic examples and leads to a complete implementation in a few iterations, with the programmer correcting counterexamples resulting from underspecified transitions by adding concrete examples in each iteration.

Categories and Subject Descriptors D.1 [*Programming Techniques*]: Automatic Programming; D.2.2 [*Design Tools and Techniques*]: Computer-aided Software Engineering; D.2.4 [*Software Verification*]: Model Checking

Keywords Program Synthesis; Distributed Protocol Synthesis; Cache Coherence Protocols; Programming by Example.

* This research was partially supported by NSF award CCF 0905464 and NSF Expeditions in Computing grant CCF 1138996.

† Jyotirmoy V. Deshmukh is currently a researcher at Toyota Technical Center, Gardena, CA.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

1. Introduction

Over the last few decades, technology for program analysis, model checking, and software verification has matured, and has witnessed growing adoption in industry. However, verification tools have largely been confined to discovering bugs in systems that have already been designed. This raises the question: how can we leverage the advances in analysis tools to assist programmers, during the program development phase, in an interactive manner? In this paper, we propose a methodology for programming distributed protocols, in which a programmer can express the intended design using multiple formats, which a synthesis tool then integrates into an executable implementation. Our approach is inspired by recent work on *sketching* and on *programming by examples*. In *sketching*, a programmer writes a partial program with incomplete details, and a synthesizer then fills in the missing details based on user-specified assertions [23, 24] (see also [15, 26]). With this approach, the programmer can continue to use the familiar imperative style of programming, but can use the synthesis tool to find intricate details necessary for fine-tuning code by supplying a few high-level invariants. In *programming by examples*, the programmer expresses the desired behavior using a set of example input/output traces, and the synthesis tool constructs an executable implementation consistent with these examples. This has been shown to be an intuitive and effective style for programming finite-state reactive controllers [9–11], Excel macros performing string manipulations [7], and pointer-manipulations for updating data structures [22].

This paper focuses on programming reactive systems such as distributed protocols. For such protocols, although the core algorithms are published with accompanying correctness proofs and performance analyses, these descriptions typically do not specify every detail, and translating them into a correct working implementation is challenging due to corner cases arising from asynchrony and concurrency [2]. A distributed protocol is typically described by listing the set of processes participating in the protocol, the set of channels connecting the processes, the types of messages each channel can carry, and the descriptions of individual processes given as *extended finite-state-machines* (EFSMs). An EFSM description consists of a list of internal state variables, a set of control states, and a set of transitions connecting control states. Each transition is specified by a guard condition and update code that refer to the internal state variables and fields of incoming and outgoing messages. This style of specifying distributed protocols is common in standardized descriptions of network protocols as RFCs, rigorous textbook descriptions of distributed algorithms [17], modeling languages supported by formal verification tools such as SPIN [12] and Murφ [5], and domain-specific languages such as SLICC [18]. The traditional programming style requires the programmer to fill in every detail of every transition, and our goal is to offer programmer assistance with this tedious and error-prone task.

Using the proposed language, TRANSIT, a programmer describes the communication architecture of a distributed protocol

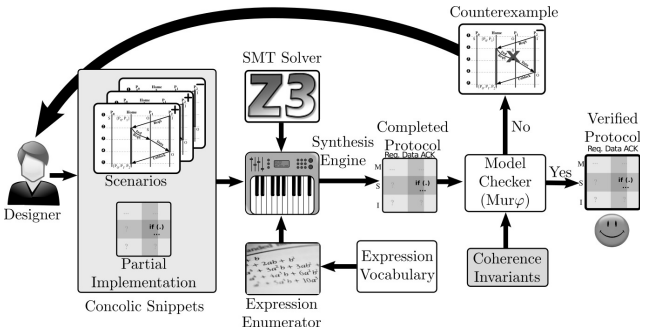

Figure 1. Design methodology using concolic snippets

by listing the set of processes, the set of channels connecting them, the types of messages each channel can carry, and the EFSM skeleton for each process by listing control states and internal state variables. TRANSIT also allows behavioral specification of each process using *concolic* snippets, which are fragments of desired execution that contain conditional updates to variables using both concrete and symbolic values (the term "concolic" was coined in the context of testing programs using both concrete and symbolic inputs [20]). This approach offers the programmer the flexibility to either describe each EFSM transition in the traditional way using symbolic expressions for guards and updates, or to specify only representative examples of such transitions. The TRANSIT synthesis tool then generates a complete protocol by inferring guards and update actions that are *consistent* with the specified snippets. The programmer also specifies high-level temporal invariants which the protocol needs to satisfy. A finite instance of the synthesized protocol is then checked against these invariants using a model checker. The programmer debugs any reported counterexamples with a visualizer and adds more snippets to rule out erroneous behaviors. Figure 1 illustrates the proposed methodology.

The main computational problem in our approach is to infer an expression to be used in a guard or an update appearing in an EFSM transition that is (1) built in a type-consistent manner using the specified vocabulary of function symbols and variables, and (2) is consistent with the given set of concrete and symbolic examples. Our solution is a variant of the *counterexample-guided inductive synthesis* (CEGIS) approach [8, 13, 23], and involves an interplay between pruning based on concrete examples and concretization of symbolic examples using an SMT solver. Specifically, our implementation enumerates expressions of increasing sizes, where the space of expressions is pruned significantly by considering two expressions equivalent if they evaluate to the same values for the given concrete examples. This results in an expression that is consistent with all the concrete examples. This expression is then checked for consistency with respect to the symbolic examples using an SMT solver. If the consistency check succeeds, then the expression is the desired answer. If it fails, the evidence for inconsistency returned by the solver contributes to a concrete example, which is considered in subsequent iterations.

The TRANSIT prototype implementation invokes the SMT solver Z3 [19] and the model checker Murφ [5]. To evaluate the algorithm for inferring an expression from concolic examples, we consider a vocabulary of the common operations on Booleans, integers, and bounded sets. The algorithm can infer expressions with a size of 15 symbols within a few minutes.

To evaluate the feasibility of the proposed methodology, we focus on cache coherence protocols, a class representative of distributed protocols. We performed two case studies where students with no experience with coherence protocols used the TRANSIT prototype to program two canonical cache coherence protocols from textbook descriptions using concolic examples [25]. Both case studies resulted in generating protocols that were successfully checked by the model checker. We found that: (1) the final, correctly generated protocol typically required one or two concrete snippets per transition, (2) synthesizing the protocol from snippets took only a few seconds, and (3) the entire iterative protocol development required a few hours of manual effort for either case study.

In our final case study, we used TRANSIT to specify the industrial-strength SGI Origin protocol. The published description [16] was directly mapped to symbolic snippets corresponding to typical transitions. However, as the published prose is not a complete specification, it results in an implementation that violates coherence invariants during the model-checking phase. Because the counterexamples produced by the model checker correspond to concrete executions, our methodology allowed the programmer to augment the original description with concrete fixes to these violating traces, culminating in a correct implementation of the protocol.

2. Illustrative Examples of Our Approach

We illustrate the use of TRANSIT using cache coherence protocols as an example application. Such protocols maintain a coherent view of shared memory among threads running in multi-core and multi-socket systems. Although programmers view memory as a single data store, most systems incorporate private per-core caches to reduce access latency. Coherence protocols prevent threads from reading stale data from caches by ensuring that reads to memory locations (addresses) receive the last value written to that location by *any* thread. Coherence is conventionally enforced using mutually exclusive read/write permissions: either multiple processors may simultaneously cache an address with read permissions (*i.e.*, *shared* readers) or a single processor may cache that address with read/write permissions (*i.e.*, *exclusive* writer). Although several different hardware and software approaches exist, this paper focuses on *invalidation-based directory* hardware implementations. In such a system, processors issue requests to a central directory which tracks the current permissions of each cache. On receiving a request for write permission, the directory issues *invalidation* messages to all caches that hold a valid copy (read or write permissions) and gathers acknowledgments from these caches before granting write permissions to the requester. Similarly, the directory redirects read requests to the last writer to that address (tracked as the "owner" in the directory) to ensure that valid data is transferred to the requesting processor.

As with existing approaches for modeling protocols using state machines, TRANSIT views the protocol in terms of a well defined structure (referred to as the *protocol skeleton*) and its behavioral description.

Protocol skeleton. A protocol description usually includes the high-level structural description of the desired protocol. For example, from Chapter 8 of the coherence primer [25], the specification would include: (1) Processes: a distinguished EFSM called the *Directory* and one EFSM per cache controller. (2) Communication architecture: three networks that the EFSMs use to send/receive messages, including any ordering guarantees. (3) Message types: such as `Get` and `Data` that may be carried on specific networks (for instance, the `Response` network can only carry `Data` and `Inv-Ack` message types). (4) Control states of each EFSM: such as M, S and I for the directory EFSM. (5) Internal variables for each EFSM: such as `Sharers` in directory, which is used to track the list of cache processes with a valid copy of data. The above *protocol skeleton* is usually expressed as type declarations and variable definitions in existing approaches. The programmer specifies the protocol skeleton similarly in the proposed approach as well.

Traditional behavioral description. Having set up the above structure, a protocol implementation next requires the behavioral specification of the EFSMs. Protocol designers typically describe behavior in terms of example flows [27] or interaction of processes when they exchange messages, triggering events and consequent actions. These descriptions are commonly expressed and communicated informally (using visual aids like example state diagrams and message sequence charts, or listing case-by-case actions textually) both in industrial protocols as well as in academic literature [16, 25]. Such informal descriptions can completely specify relatively straightforward protocols like the textbook MSI example [25]. In these cases, the programming task is largely to infer and symbolically codify the semantics implied by the examples. For more complex protocols, the informal descriptions convey salient features, like common-case behavior and perhaps some of the more interesting scenarios that might arise (usually in terms of rules). Implementing such protocols involves the additional step of filling in any unspecified details. However, it has been shown that this approach to implementing complete protocols symbolically as a program can be error-prone even for experts, with bugs manifesting even in released processors [1, 2].

Specifying behavior with snippets. We observe that the complexity in protocols does not usually stem from the code for individual transitions — these tend to be straight line code. The complexity is due to corner cases and race conditions arising from concurrency and asynchrony when protocols are considered in their entirety [2]. Using a concolic approach allows the programmer to mix symbolic snippets (transition code) describing well-defined behavior (like rules or state machine transitions) with concrete snippets describing specific example scenarios.

The TRANSIT tool composes the given concrete and symbolic snippets to generate an implementation which allows *at least* all the behaviors described in the input snippets. This implementation is then model checked for coherence invariants. Any underspecification or error in the specification of the EFSM behavior will result in a counterexample during model checking, which the programmer can then fix using concrete snippets to prevent subsequent occurrences of at least that specific violation.

Example using concolic snippets. We illustrate this mixed (or concolic) use of TRANSIT with an anecdote from our case study in implementing the SGI Origin protocol from published informal textual rules [16] (Section 6 contains more details on this case study, and using TRANSIT with concolic snippets to implement protocols which are completely specified). The protocol skeleton states that the directory EFSM has variables `Sharers` and `Owner` to maintain its internal state. One of the rules in the paper describes the behavior of a directory upon a read request from a cache as: *"If directory state is Exclusive with another owner, transitions to Busy-shared with requester as owner and send out an intervention shared request to the previous owner and a speculative reply to the requester. Go to 5b"*. Note that this description does not specify how the `Sharers` process variable in the directory EFSM needs to be updated. The programmer indicated that if the sender of the message was not the previous `Owner`, the value of the `Sharers` variable needs to contain *at least* the sender of the received message in addition to the old value of the `Sharers` variable. Based on this specification, TRANSIT generated the code:

```
Sharers := Sharers ∪ {Msg.Sender}
```

However, an attempt to verify the generated implementation using Murφ resulted in a coherence violation. TRANSIT provided a visual trace of the counterexample to the programmer, a simplified version of which is shown in Figure 2. Observe that the transition on the directory shown in Figure 2 is a concrete instance of

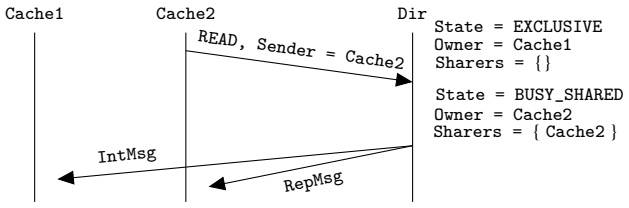

Figure 2. The counterexample trace

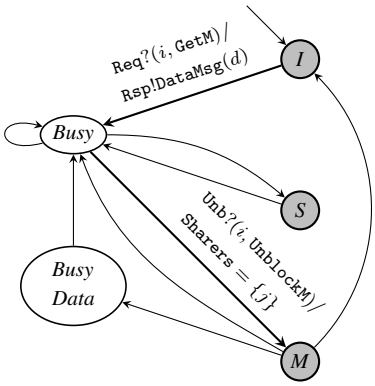

Figure 3. EFSM representing a directory process. The M (modified) state indicates read/write permissions. The S (shared) state indicates read-only permissions and the I (invalid) state indicates no permissions. The rest of the states are transient states.

the scenario described earlier, where `Msg.Sender` is `Cache2` and the `Owner` variable is initially set to `Cache1`. Upon inspecting the trace, the programmer recognized that in this particular scenario, with `Sharers` needed to include the previous value of the `Owner` as well. After a concrete snippet specifying this specific conditional update was added, TRANSIT then generated a new implementation including the correct update to `Sharers` as:

```
Sharers := Sharers ∪ {Msg.Sender, Owner}
```

This example illustrates how a programmer can benefit from using the concolic nature of TRANSIT. TRANSIT allows the programmer to combine the familiar symbolic programming style for specifying the protocol skeleton and well-understood behavior. TRANSIT also makes it convenient to fix bugs with concrete execution fragments that describe the desired outcome in the particular scenario where the bug manifests.

3. Protocol Specification using Concolic Snippets

A protocol implementation results in transition code for communicating EFSMs. Figure 3 shows the EFSM for a directory process in a cache coherence protocol. TRANSIT borrows from traditional software [18] and hardware [4] protocol description languages to specify the protocol skeleton (*i.e.*, types and variable definitions) and transitions (using a guarded command language). However, TRANSIT offers the additional ability to mix concrete and symbolic transitions as snippets, and to under-specify elements of transitions for the tool to infer. We now describe how to specify a protocol using TRANSIT.

3.1 High-level Building Blocks of TRANSIT

Coherence protocols typically assume an *asynchronous, message-passing* based model of communication. Each process description includes a list of input channels to receive messages, and output channels on which messages are sent. A channel can be modeled as a multiset or a queue depending on the desired ordering guarantees.

```
Transition(CurrentState, InputEvent)
  [optional guard] => (NextState, OutMsg1, ...)
      Pre_1 ==>
          Post_11;
          Post_12;
          ...
      Pre_2 ==>
          ...
```

Figure 4. A concolic snippet. *CurrentState* and *NextState* are the start and end control states. The snippet specifies zero or more outbound messages. It also specifies a *guard-action* block for each *guard* containing a set of conditional updates. The expression Pre_i specifies the condition (on process variables and the fields of the received message) under which the boolean constraints Post_{ij} hold. Each Post_{ij} constrains the updated value of exactly one process variable or output message field in terms of the old values of the process variables and the fields of the received message.

The syntactic elements of TRANSIT, shown in Figure 4, are as follows:

Control states define the logical state of the directory and cache machines. These can be thought of as the *program counter* for these machines.

Process variables internal to each machine, such as `Sharers` in the directory. These may be Booleans, integers or bounded sets.

Input events correspond to receiving messages or external triggers (to issue requests). An input event is specified as a single message variable and the network on which it is received.

Guards are Boolean expressions over process variables and fields of received messages.

Output events correspond to sending messages to other processes. They are represented as a list of message variables and the name of network on which each message is sent.

Action statements update process variables to new values and specify values for the fields of outbound messages.

We defer a discussion on *Pre* and *Post* to Section 3.2. A complete specification of the behavior of an EFSM is thus a set of actions A that are to be executed, a set of output events O that are to be generated, the next control state s' to transition to, all predicated on a guard g, for each combination of input event and control state $\langle e, s \rangle$. The semantics of such a specification are that the EFSM executes actions A, generates events O, and finally transitions to control state s' in response to an input event e when the EFSM is in control state s, provided g evaluates to `true`. In the asynchronous model of computation, the order in which processes execute is nondeterministic. However, once a process has been chosen for execution, its actions are assumed to be deterministic.

3.2 Programming with TRANSIT

Figure 4 shows the composition of a single transition snippet in TRANSIT. Using existing approaches such as SLICC [18] or hardware description languages, the programmer would completely specify the guards and actions in the transition. However, TRANSIT allows for guards to be left empty and for actions to be specified as constraints governed by pre- and post-conditions as shown in Figure 4. In such cases, the tool computes expressions for the empty guards and update statements satisfying the given constraints.

The programmer has a choice of specifying completely symbolic, completely concrete, or mixed concolic snippets. In each case, TRANSIT generates a symbolic implementation. The pre- and post-conditions are Boolean predicates over process variables and message fields. Primed variables denote updated values of process variables. We assume a parallel assignment model, which precludes primed variables from appearing in pre-conditions and allows at most one primed variable per post-condition. Note that although guards and pre-conditions are both Boolean predicates ranging over the same variables, they have markedly different uses: guards appear in the *complete* protocol implementation of the EFSM in a guarded-command language, whereas pre-conditions are inputs to the inference engine which allow users to capture particular scenarios and are not part of the generated protocol implementation.

Symbolic snippets. By specifying the guard and actions symbolically, we obtain a symbolic snippet which completely specifies a transition. A non-empty guard is assumed to be symbolic (*i.e.*, completely specified and not generated by TRANSIT). Symbolic actions are specified by making the post-condition an equality constraint.

Concolic snippets. If the guard is left empty, or the actions are specified using pre- and post-condition constraints (instead of equality), the snippet is concolic in nature. The TRANSIT tool generates symbolic code for update statements and/or the guard to be consistent with the given constraints. The following concolic snippet specifies the transition for the directory process mentioned in the SGI-Origin anecdote in Section 2.

```
Transition(EXCLUSIVE, ReqNet Msg) {
  [] => (BUSY_SHARED, RepNet RMsg, IntNet IMsg) {
      (Msg.MType = READ & Msg.Sender != Owner) ==> {
          SubsetOf(SetUnion(Sharers, {Msg.Sender}),
                   Sharers');
          ...
}}}
```

This snippet is applicable only when the current state is `EXCLUSIVE` and a message is received on the network `ReqNet`. To be consistent with the snippet, TRANSIT must generate code to update the new value of `Sharers` to be a superset of the old value and the sender of the incoming message.

Concrete snippets. Concolic snippets may contain symbolic pre- and post-conditions over variables, but concrete snippets specify concrete values for any variables in these constraints. They may leave the guard empty as well. For example, the concrete snippet below shows the bug-fix described in Section 2:

```
Transition(EXCLUSIVE, ReqNet Msg) {
  [] => (BUSY_SHARED, RepNet RMsg, IntNet IMsg) {
      (Msg.MType = READ & Owner = C1 &
       Msg.Sender = C2) ==> {
          Sharers' = {C1, C2};
          ...
}}}
```

In summary, symbolic and concrete snippets can be thought of as instances of concolic snippets in which guards and actions are either completely specified (symbolic), or specified with concrete values for all variables (concrete). The task of the TRANSIT tool is to generate expressions for guards and update expressions for variables such that they are consistent with the given snippets. We formalize this problem in Section 5 after first describing our expression inference algorithm in the next section.

4. Inferring Expressions

We first define the problem of inferring an expression from concrete or concolic examples in Section 4.1, which our synthesis procedure uses as a subroutine. Section 4.2 presents a solution for a restricted version of this problem in which the examples involve only concrete values, whereas Section 4.3 presents a solution for the general case where the examples have concolic values.

The expression inference problem corresponds to the following computational problem: given a quantifier-free formula $\mathcal{C}$ over (typed) variables $\mathcal{V}$ and a distinguished (output) variable o, find an expression e such that the formula $\mathcal{C}[o := e]$ is valid, where $\mathcal{C}[o := e]$ is the usual notation for the formula obtained by syntactically substituting e for every occurrence of o in $\mathcal{C}$. Conceptu-

Function	Description
add (Int, Int) → Int	Integer Addition
sub (Int, Int) → Int	Integer Subtraction
inc (Int) → Int	Add one to an Integer
dec (Int) → Int	Subtract one from an Integer
setadd (Set, PID) → Set	Add an entry into a Set
setsize (Set) → Int	Cardinality of a Set
setunion (Set, Set) → Set	Set Union
setinter (Set, Set) → Set	Set Intersection
setminus (Set, Set) → Set	Set Difference
setof (PID) → Set	Create a singleton Set
or (Bool, Bool) → Bool	Boolean Disjunction
and (Bool, Bool) → Bool	Boolean Conjunction
not (Bool) → Bool	Boolean Negation
setcontains (Set, PID) → Bool	Membership test on a Set
iszero (Int) → Bool	Test if an integer is Zero
$\forall t \in \mathcal{T}$ equals $(t, t) \rightarrow$ Bool	Equality Test
ge (Int, Int) → Bool	Greater than or equal to
gt (Int, Int) → Bool	Greater than
$\forall t \in \mathcal{T}$, ite (Bool, t, t) → t	Conditional Expression
numcaches () → Int	# of Caches (constant)

Table 1. Expression Vocabulary used in Coherence Protocols

ally, this problem is similar to solving $\exists\forall$-formulas, and we use the *counterexample-guided inductive synthesis* (CEGIS) strategy that has previously been used to solve such problems [8, 23].

The algorithm enumerates expressions, with suitable pruning strategies, to find a candidate expression e that is consistent with the concrete examples that contribute to $\mathcal{C}$. It then checks if $\neg\mathcal{C}[o := e]$ is satisfiable. If $\neg\mathcal{C}[o := e]$ is unsatisfiable, then e is the desired expression. Otherwise, the satisfying model contributes a concrete example that is added to the set of concrete examples and the entire process is repeated. Thus, the algorithm handles "exists an expression" using enumerative techniques and "is valid for all variable values" using symbolic constraint solving.

4.1 Problem Definition

We define an *expression vocabulary* G as a tuple $(\mathcal{T}, \mathcal{F})$, where $\mathcal{T}$ is a set of *types*, and $\mathcal{F}$ is a set of typed function symbols, where for each $f \in \mathcal{F}$, $\mathsf{arity}(f) \in \mathbb{N}$ denotes the arity of f, $\mathsf{type}(f) \in \mathcal{T}$ denotes the type of the range of f, and $\mathsf{argtype}(f, i) \in \mathcal{T}, i \in [1, \mathsf{arity}(f)]$ denotes the type of the domain of the i^{th} argument to f. Function symbols of arity zero are constants. $\mathcal{F}$ is finite, so we have a fixed number of constant symbols. However, constant expressions such as 2 are allowed as an abbreviation for $\mathtt{add}(1, 1)$.

Let $\mathcal{V}$ denote a set of typed variable symbols and let $\mathsf{type}(v) \in \mathcal{T}$ denote the type of a variable v in $\mathcal{V}$. Given an expression vocabulary $G = (\mathcal{T}, \mathcal{F})$ and a set of typed variables $\mathcal{V}$, we denote the set of expressions of type t as $\mathsf{Exp}(\mathcal{F}, \mathcal{V}, t)$. This set is defined inductively as follows: (1) If $v \in \mathcal{V}$ then $v \in \mathsf{Exp}(\mathcal{F}, \mathcal{V}, \mathsf{type}(v))$, and (2) If $f \in \mathcal{F}$, and $\forall i \in [1, k] : e_i \in \mathsf{Exp}(\mathcal{F}, \mathcal{V}, \mathsf{argtype}(f, i))$, then $f(e_1, e_2, \ldots, e_k) \in \mathsf{Exp}(\mathcal{F}, \mathcal{V}, \mathsf{type}(f))$, where $k = \mathsf{arity}(f)$.

If $e \in \mathsf{Exp}(\mathcal{F}, \mathcal{V}, t)$, then we write $\mathsf{type}(e) = t$. We assume that the Boolean type (denoted Bool) is always present in any instantiation of $\mathcal{T}$ and that the basic Boolean operators (conjunction, disjunction and negation) are part of $\mathcal{F}$.

For specifying cache coherence protocols, we use an expression vocabulary where the set of types is $\mathcal{T} = \{\mathtt{Bool}, \mathtt{Int}, \mathtt{PID}, \mathtt{Set}\} \cup \mathtt{Enums}$. Here, Int denotes the integer type, PID is a special type for process identifiers, Set is a type denoting a set of values of type PID, and Enums is a set of user defined enumerated types. The set of functions $\mathcal{F}$ in the vocabulary is shown in Table 1.

The *size* of an expression e, $\mathsf{size}(e)$, is defined as the number of function symbols and variable symbols appearing in e. Further, we denote by $\mathsf{Exp}_k(\mathcal{F}, \mathcal{V}, t)$ the set $\{e : e \in \mathsf{Exp}(\mathcal{F}, \mathcal{V}, t) \wedge \mathsf{size}(e) = k\}$, *i.e.*, the subset of expressions of type t with size k.

Given a set of typed variables $\mathcal{V}$ and a distinguished typed variable $o \notin \mathcal{V}$ (henceforth known as the output variable), an *example* C is a Boolean formula of the form $\mathsf{pre} \Rightarrow \mathsf{post}$ where $\mathsf{pre} \in \mathsf{Exp}(\mathcal{F}, \mathcal{V}, \mathtt{Bool})$ and $\mathsf{post} \in \mathsf{Exp}(\mathcal{F}, \mathcal{V} \cup \{o\}, \mathtt{Bool})$. In other words, if the variables in $\mathcal{V}$ satisfy the pre-condition pre, then the output variable o should be assigned an expression over these variables such that the post-condition post holds. An *example set* over the output variable o, denoted $\mathcal{C}(o)$, is defined as the set of examples $\{C_1, \ldots, C_n\}$ such that the conjunction $\bigwedge_n(\mathsf{pre}_i \Rightarrow \mathsf{post}_i)$ is satisfiable. An expression e is *consistent* with the example set $\mathcal{C}(o)$ iff for all $C \in \mathcal{C}(o)$, $C[o := e]$ is a formula that is valid.

Example. Let $\mathcal{V} = \{x, y\}$, where $\mathsf{type}(x)$ and $\mathsf{type}(y)$ are both Int. Consider the example set $\{C_1\}$, where C_1 is $((x = 0) \wedge (y = 1)) \Rightarrow (o = 1)$. The expressions y and $x + y$ are all consistent with $\mathcal{C}(o)$, while the expression x is inconsistent, as $C_1[o := x]$ is the invalid formula $(x = 0) \wedge (y = 1) \Rightarrow (x = 1)$.

Now, consider the example set $\{C_1, C_2\}$, where C_2 is the example $(x = 1) \wedge (y = 1) \Rightarrow (o = 2)$. The expression y is not consistent with the new example set, as $C_2[o := y]$ is the invalid formula $(x = 1) \wedge (y = 1) \Rightarrow (y = 2)$; however, the expression $x + y$ is still consistent.

The expression inference problem can now be stated as: given an expression vocabulary $G = (\mathcal{T}, \mathcal{F})$ with a set of variables $\mathcal{V}$ and an example set $\mathcal{C}(o)$ over the typed output variable o, find an expression e in $\mathsf{Exp}(\mathcal{F}, \mathcal{V}, \mathsf{type}(o))$ which is consistent with $\mathcal{C}(o)$.

4.2 Inferring Expressions from Concrete Examples

We say that an example is *concrete* if it has the form: $(\bigwedge_{v \in \mathcal{V}}(v = k_v)) \Rightarrow (o = k_o)$, where for all $v \in \mathcal{V}$, k_v and k_o are concrete expressions[1] of types $\mathsf{type}(v)$ and $\mathsf{type}(o)$, respectively. An example set containing only concrete examples is called a concrete example set. An alternative representation for a concrete example is the tuple (S, k_o), where S is a valuation function mapping variables in $\mathcal{V}$ to values and k_o is the concrete value for the output variable o. We use the notation $S(e)$ to denote the evaluation of an expression e for the valuation S for its constituent variables. Checking if an expression e is consistent with a concrete example C is straightforward: we check if $S(e) = k_o$. Given a concrete example set $\mathsf{ConcreteExs} = \{C_1, \ldots, C_n\}$, e is consistent with ConcreteExs iff e is consistent with each C_i.

The algorithm enumerates expressions inductively until it finds an expression e that is consistent with ConcreteExs. In the first step, only expressions of size one are considered, *i.e.*, variable symbols or constants. In the n^{th} step, expressions of size n are obtained by composing expressions obtained in previous steps using functions in $\mathcal{F}$. As each expression e is enumerated, it is checked for consistency with ConcreteExs.

To enhance scalability, the algorithm uses the examples to guide the search as well, rather than just to check for consistency. Intuitively, if two expressions e and e' evaluate to the same values for each example $C \in \mathsf{ConcreteExs}$, then only one of e or e' needs to be carried forward to the next iteration, as the information encoded in ConcreteExs is insufficient to distinguish e and e'. The algorithm exploits this property to prune the search space of expressions.

Let $\mathsf{ConcreteExs} = \{C_1, \ldots, C_n\}$ be a concrete example set. Suppose each C_i is represented as (S_i, k_i). Then the *signature* $\mathsf{Sig}(e, \mathsf{ConcreteExs})$ of an expression e with respect to ConcreteExs is a vector of values $\langle k_1, k_2, \ldots k_n \rangle$, where each $k_i = S_i(e)$. In other words, the signature of an expression e is the

[1] These are constant-valued expressions that do not contain variables. For example, if the only integer-valued constants in $\mathcal{F}$ are 0 and 1, then k_v is allowed to be the term 2 as an abbreviation for $\mathtt{add}(1, 1)$.

Algorithm 1: SolveConcrete

Input : A concrete example set ConcreteExs on a typed variable o, an expression vocabulary $(\mathcal{T}, \mathcal{F})$ and a set of typed variables $\mathcal{V}$.
Output: An expression e, which is consistent with ConcreteExs.
Data : The sets $\mathsf{ExpSet}(t, i), t \in \mathcal{T}, i \in \mathbb{N}$, denoting sets of expressions of type t and size i, initialized to empty sets. A set SigSet which stores the signatures of all enumerated expressions e over ConcreteExs, initially empty.

```
1  Goal ← ⟨k_1, k_2, ..., k_n⟩, for each C_i = (S_i, k_i) ∈ ConcreteExs
2  SizeOneExps ← ⊥
3  foreach v ∈ V do
4      SizeOneExps ← SizeOneExps ∪ {v}
5  foreach (c ∈ F) ∧ (arity(c) = 0) do
6      SizeOneExps ← SizeOneExps ∪ {c}
7  foreach e ∈ SizeOneExps do
8      if Sig(e, ConcreteExs) ∉ SigSet then
9          SigSet ← SigSet ∪ {Sig(e, ConcreteExs)}
10         if Sig(e, ConcreteExs) = Goal then
11             return e
12         ExpSet(type(e), 1) ← ExpSet(type(e), 1) ∪ {e}
13 i ← 2
14 while true do
15     foreach f ∈ F do
16         m ← arity(f)
17         foreach partition p of i − 1 into m-partitions do
18             foreach (e_1, e_2, ..., e_m) ∈
               ∏_{j=1}^{m} ExpSet(argtype(f, j), p_j) do
19                 e ← f(e_1, e_2, ..., e_m)
20                 if Sig(e, ConcreteExs) ∉ SigSet then
21                     SigSet ← SigSet∪{Sig(e, ConcreteExs)}
22                     if Sig(e, ConcreteExs) = Goal then
23                         return e
24                     Add e to ExpSet(type(e), size(e))
25     i ← i + 1
```

vector of values obtained by evaluating e over the concrete examples. Two expressions e and e' are said to be *indistinguishable* w.r.t a concrete example set ConcreteExs iff $\mathsf{Sig}(e, \mathsf{ConcreteExs}) = \mathsf{Sig}(e', \mathsf{ConcreteExs})$.

Example. Consider the expression vocabulary shown in Table 1. Suppose that we wish to describe the expression to compute the maximum of two integers a and b. Consider concrete example $C_1 = (S_1, k_1)$, where S_1 is the map $\langle a : 5, b : 3, c : 4\rangle$. Also consider the concrete example $C_2 = (\langle a : 3, b : 1, c : 2\rangle, k_2)$. Let $\mathsf{ConcreteExs} = \{C_1, C_2\}$. For expressions $e_1 = a+b, e_2 = b+a$, $e_3 = c + c$, the signatures $\mathsf{Sig}(e_1, \mathsf{ConcreteExs}) = \langle 8, 4\rangle$, $\mathsf{Sig}(e_2, \mathsf{ConcreteExs}) = \langle 8, 4\rangle$, and $\mathsf{Sig}(e_3, \mathsf{ConcreteExs}) = \langle 8, 4\rangle$ are equal. Thus e_1, e_2, and e_3 are indistinguishable with respect to ConcreteExs. Note that this implies that larger expressions such as $a + (a + b)$ and $a + (c + c)$ are also indistinguishable w.r.t. ConcreteExs. □

Algorithm 1, referred to as SolveConcrete, implements the enumeration based strategy, with the pruning of the search space based on the notion of indistinguishability, to find an expression that is consistent with respect to a given set of concrete examples.

4.3 Inferring Expressions from Concolic Examples

We now describe an algorithm for the general problem defined in Section 4.1, *i.e.*, we relax the restriction that the examples be concrete. Each example $C \in$ ConcolicExs — where ConcolicExs is a set of *concolic* examples — has the form $\mathsf{pre} \Rightarrow \mathsf{post}$, where the structure of pre and post are unrestricted.

Algorithm 2: SolveConcolic

Input : A set of concolic examples ConcolicExs over a typed output variable o, an expression vocabulary $\mathcal{F}$ and a set of typed variables $\mathcal{V}$.
Output: An expression e, which is consistent with ConcolicExs.
Data : ConcreteExs, a set of concrete examples inferred from ConcolicExs.

```
1  ConcreteExs ← {}
2  while true do
3      e ←SolveConcrete (ConcreteExs, F, V)
4      if e is consistent with ConcolicExs then
5          return e
6      else
7          foreach C ∈ ConcolicExs such that ¬C[o := e] is
           satisfiable do
8              S ← a valuation of variables v ∈ V which satisfies
               the formula ¬C[o := e]
9              k_o ← a value that satisfies the formula
               (⋀_{v∈V} v = S(v)) ∧ post(C[o := k_o])
10             ConcreteExs ← ConcreteExs ∪ (S, k_o)
```

Evaluating a candidate expression e is sufficient to check if e is consistent with respect to a given set of concrete examples. However, checking consistency of an expression w.r.t. a concolic example set requires a validity query involving terms from the expression vocabulary. For our expression vocabulary we can use a satisfiability modulo theories (SMT) solver to check consistency.

We now discuss how we can adapt Algorithm SolveConcrete to infer expressions consistent with respect to a set of concolic examples. The notion of indistinguishability has been defined only with respect to a set of concrete examples. It has no simple analogue with respect to concolic examples. But using indistinguishability to prune the search space contributes significantly to the scalability of our enumerative approach, as our experiments show in Section 4.4. Thus, it would be desirable to retain the benefits of using indistinguishability to prune the search space in the solution for the general problem of inferring an expression consistent w.r.t. a set of concolic examples. To this end, the algorithm which we now propose uses a set of *concretizations* of the concolic examples to serve as concrete examples w.r.t. which the search space is pruned.

Consider a concolic example set ConcolicExs over an output variable o and an expression e found to be inconsistent with some concolic example C of the form $\mathsf{pre} \Rightarrow \mathsf{post}$. Then there must exist a valuation S over the variables $v \in \mathcal{V}$ which satisfies the formula $\neg C[o := e]$. Now, pre must necessarily be satisfiable under the valuation S. Let k_o be a value which satisfies the formula $\left(\bigwedge_{v \in V} v = S(v)\right) \wedge \mathsf{post}(C[o := k_o])$. The tuple (S, k_o) is now a *concretization* of the concolic example C. The proposed algorithm thus maintains a concrete example set ConcreteExs containing concretizations of concolic examples $C \in$ ConcolicExs. By ensuring that any expression e' enumerated henceforth is consistent with ConcreteExs before checking for consistency over ConcolicExs, the number of expensive SMT queries needed can be reduced. Whenever an expression e' that is consistent with ConcreteExs, but is inconsistent with ConcolicExs is found, additional concretizations of the concolic examples are added to ConcreteExs. This technique helps in two ways: (1) the number of queries to an SMT solver is reduced, and (2) the search space can be pruned on the basis of indistinguishability.

Algorithm 2, referred to as SolveConcolic, makes use of these pruning strategies. Note that it uses Algorithm SolveConcrete as a subroutine. It maintains a set of concrete examples, ConcreteExs

Expression	Counterexample	Inferred Example
—	—	$\langle a:0, b:-1, o:0\rangle$
a	$\langle a:0, b:1, o:0\rangle$	$\langle a:0, b:1, o:1\rangle$
$\mathsf{ite}(\mathsf{iszero}(\mathsf{dec}(b)), b, a)$	$\langle a:0, b:2, o:0\rangle$	$\langle a:0, b:2, o:2\rangle$
$\mathsf{ite}(\mathsf{gt}(b, a), b, a)$	—	—

Table 2. Illustration of the working of SolveConcrete

— which are concretizations of examples in ConcolicExs — and invokes the SolveConcrete algorithm on it. If SolveConcrete finds an expression e which is consistent w.r.t. ConcreteExs, the algorithm checks if e is consistent w.r.t. ConcolicExs with a query to an SMT solver. If e is inconsistent with ConcolicExs, then the algorithm augments ConcreteExs with new concrete examples as described earlier and repeats the entire process.

Example. To illustrate the working of the algorithm, consider finding an expression for $\mathsf{max}(a, b)$, with the expression vocabulary in Table 1. We can specify $\mathsf{max}(a, b)$ with the concolic example:

$$\texttt{true} \Rightarrow (o \geq a) \wedge (o \geq b) \wedge ((o = a) \vee (o = b))$$

Table 2 shows the expressions that were checked for consistency, the witness for the inconsistency of the expression obtained, and the concrete example inferred from the witness. The first row of the table seeds the set of concrete examples. The subsequent rows indicate the expression queried for symbolic validity and the concrete example inferred. We observe that the expression corresponding to $\mathsf{max}(a, b)$ was discovered after making only four calls to the SMT solver, although the algorithm enumerated approximately five hundred expressions. □

4.4 Evaluation of the Expression Inference Algorithm

To evaluate the utility of the expression inference algorithm, we focus on the *size* of the expressions which the algorithm is able to compute successfully as a key metric. To benchmark the performance of the algorithm `SolveConcrete`, a large number of random expressions of varying sizes were generated. For each expression, a set of ten concrete examples that were consistent with the expression was generated. For each such concrete example set, the Algorithm SolveConcrete was used to compute an expression that is consistent. Figure 5 shows that the "Pruned" variant — which prunes the search space using the notion of indistinguishability — often explores two to three orders of magnitude fewer expressions than the "Exhaustive" variant — which does not perform any pruning — for expression sizes larger than ten (note the logarithmic scale on the Y-axis in Figure 5).

To evaluate the algorithm SolveConcolic, we used the benchmarks shown in Table 3. The algorithm computes expressions of up to size 15 within a reasonable amount of time as shown in Table 3. The algorithm exceeds our 30 minute time-out on only one benchmark, whose solution has an expression size greater than 20. The right-most column in Table 3 shows that the algorithm reaches the desired solution within a few iterations of the CEGIS outer loop.

5. The TRANSIT Synthesis Tool

We now describe the implementation of TRANSIT, which uses the algorithms described in Section 4 to generate an implementation of the protocol from concolic snippets. To this end, TRANSIT needs to: (1) compute process-variable update expressions, and (2) compute guards for transitions.

5.1 Computing Update Expressions

TRANSIT employs a parallel assignment model, allowing the update for each primed variable to be computed independently. For each primed variable v', the TRANSIT snippets specify a set of concolic examples, each of the form $\mathsf{Pre} \Rightarrow \mathsf{Post}$. This is precisely the form in which the Algorithm SolveConcolic expects a concolic example and can thus be solved directly by the algorithm. We instantiate o in Algorithm SolveConcolic with v', $\mathcal{V}$ with the set of all process variables and incoming message fields *i.e.*, all the variables which can be *read* in the current scope and ConcolicExs with the set of concolic examples provided in the TRANSIT snippets.

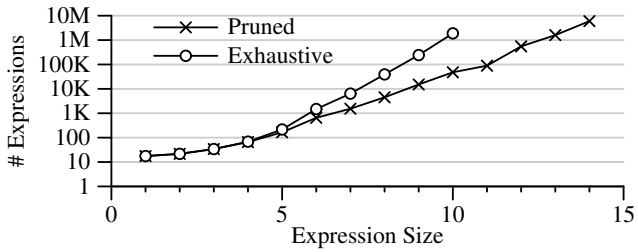

Figure 5. Average number of expressions explored for various expression sizes by the Pruned and Exhaustive variants of Algorithm SolveConcrete. We omit data for the Exhaustive variant for sizes greater than 10 where it exceeds the memory limit of 3.5 GB.

5.2 Computing Guard Expressions

A guard can be viewed as a Boolean-valued expression. The key difference between computing guards and computing update expressions is that for a given control state and input event, guards cannot be computed independently of each other. To ensure that the behavior of the EFSM implementations generated by TRANSIT are deterministic, the computed guards for each control state and input event pair are required to be pairwise mutually exclusive. To compute guards on transitions from a given state, TRANSIT groups the concolic snippets with the same starting state, input event and next state into one guard-action as shown in Figure 4. Therefore, given a starting state and input event, each possible next state has a corresponding guard-action associated with it.

Given a set of guard-actions $B_1, \ldots, B_n$, the j^{th} guard-action block is a set of examples conditioned by Pre_{j1}, ..., Pre_{jk_j}. The algorithm for computing guards sequentially computes the guards for each of the blocks, starting with B_1. Thus, before synthesizing the j^{th} guard, it has the guards $\varphi_1, \ldots, \varphi_{j-1}$ corresponding to the guard-action blocks $B_1, \ldots, B_{j-1}$ available to it. To compute a guard φ_j for the guard-action block B_j, we observe that φ_j must evaluate to false whenever the guard φ_i evaluates to true, for any $i < j$. This property is expressed with the concolic examples:

$$\mathsf{ConcolicExs}_1 = \{\varphi_i \Rightarrow \neg\varphi_j \mid i < j\}$$

Next, φ_j must evaluate to true whenever any of the preconditions $\mathsf{Pre}_{jl}, l \in [1, k_j]$ evaluate to true. This property can be expressed with the following concolic example:

$$\mathsf{ConcolicExs}_2 = \left\{ \left(\bigvee_{l=1}^{k_j} \mathsf{Pre}_{jl} \right) \Rightarrow \varphi_j \right\}$$

Also, corresponding to each block B_i for which a guard has not yet been synthesized (*i.e.*, $i > j$), φ_j must evaluate to false whenever any of preconditions in B_i evaluate to true. This property is expressed with the following symbolic examples

$$\mathsf{ConcolicExs}_3 = \left\{ \left(\bigvee_{l=1}^{k_i} \mathsf{Pre}_{il} \right) \implies \neg\varphi_j \;\middle|\; i > j \right\}$$

Finally the concolic example set required for inferring φ_j is the union of $\mathsf{ConcolicExs}_1$, $\mathsf{ConcolicExs}_2$ and $\mathsf{ConcolicExs}_3$. Again, $\mathcal{V}$ is instantiated to be the set of all process variables and the incoming

#	Description	Expected Expression	Exp. Size	Constraints	Time (s)	# Iters
1	Max. of a, b	$\mathsf{ite}(\mathsf{gt}(a,b),a,b)$	6	**(a)** $(a>b)\Rightarrow(o=a);(b>a)\Rightarrow(o=b)$	<1	1
				(b) $\mathtt{true}\Rightarrow(o\geq a\wedge o\geq b\wedge(o=a\vee o=b))$	<1	2
2	Max. of a, b, c	Similar to 1	15	Similar to 1**(a)**	536	7
				Similar to 1**(b)**	762	16
3	Sym. Diff. of s_1, s_2	$\mathsf{setunion}(\mathsf{setminus}(s_1,s_2),\mathsf{setminus}(s_2,s_1))$	7	$\mathtt{true}\Rightarrow(o\subseteq(s_1\cup s_2))$; $\mathtt{true}\Rightarrow(o\cap(s_1\cap s_2))=\{\}$; $\mathtt{true}\Rightarrow(o\cup(s_1\cup s_2))=(s_1\cup s_2)$	<1	2
4	Sym. Diff. of 3 sets	Similar to 3	11	Similar to 3	<1	6
5	Sym. Diff. of 4 sets	Similar to 3	15	Similar to 3	132	14
6	Conditional Update	$\mathsf{ite}(\mathsf{equals}(e,c_1),a,b)$	6	$(e=c_1)\Rightarrow(o=a)$; $(e\neq c_1)\Rightarrow(o=b)$	<1	4
7	Largest of 2 sets	$\mathsf{ite}(\mathsf{gt}(\mathsf{setsize}(s_1),\mathsf{setsize}(s_2)),s_1,s_2)$	8	**(a)** $(\lvert s_1\rvert>\lvert s_2\rvert)\Rightarrow(o=s_1)$; $(\lvert s_2\rvert>\lvert s_2\rvert)\Rightarrow(o=s_2)$	<1	1
				(b) $\mathtt{true}\Rightarrow(\lvert o\rvert\geq\lvert s_1\rvert\wedge\lvert o\rvert\geq\lvert s_2\rvert\wedge(o=s_1\vee o=s_2))$	<1	2
8	Largest of 3 sets	Similar to 7	>20	Similar to 7**(b)**	TO	–

Table 3. Description of the benchmarks used to evaluate the algorithms and experimental results

		Synthesis						
		Updates			Guards			
Protocol	# Scenarios	Num. synth.	Exps tried	Time (secs)	Num. synth.	Exps tried	Time (secs)	**State-Space**
VI	19	49	449	<1	17	525	<1	140K
MSI	77	157	3330	<1	45	3710	<1	854K

Table 4. Performance of snippet-based design

message fields. o is instantiated to be φ_j. Algorithm SolveConcolic can now be used to infer the required guard expression φ_j.

6. Evaluating TRANSIT

To evaluate the proposed approach for specifying protocols with concolic snippets, we describe our experiences in using TRANSIT to specify some representative cache coherence protocols. As the proposed aims to be easy to use, a direct scientific comparison with existing approaches is challenging for two reasons: (1) a large user study, besides logistical difficulty, presents the problem of defining and extracting meaningful comparative metrics, and (2) a programmer attempting the same problem with two different approaches is subject to a familiarity bias. We instead document the experiences of three different programmers — each a co-author of this paper — in specifying three coherence protocols of varying complexity (two textbook examples and one industrial strength protocol) with TRANSIT. We summarize the perceived advantages as well as limitations of our approach based on these experiences. All experiments were performed using the expression vocabulary shown in Table 1.

We first validated the feasibility of using our expression enumeration approach by transcribing fully specified protocols from the GEMS simulation toolkit [18] into fully symbolic TRANSIT snippets. With four cache processes and one directory, the entire synthesis process took less than a second for each protocol. The key results are summarized in Table 4.

6.1 Case Study A: Non-blocking MSI Protocol.

We specified the non-blocking "MSI protocol" described in the synthesis lectures [25] using concolic snippets in TRANSIT. A non-blocking directory allows a greater number of concurrent requests to be in flight, requiring the programmer to consider a larger number of corner cases due to increased concurrency.

The scenarios described in the text resulted in a sparse initial set of snippets, as most of the tricky corner cases were either indirectly specified in the textual description or were left unspecified. Hence, the programmer added 67 more snippets over 13 debugging iterations before converging to a correct protocol. In each such iteration, the programmer either added symbolic snippets, when the behavior of the protocol in some corner case was completely unspecified, or concrete snippets, when a specification existed but was incomplete. Table 5 summarizes the effort and complexity in this experiment.

6.2 Case Study B: From MSI to MESI

The goal of our second case study was to augment the blocking MSI protocol with an "*E*" state to arrive at the MESI protocol. The *E* state (shorthand for *exclusive*) is an optimization that grants read-write permissions to the first reader of an unshared address (*i.e.*, not present in any cache) — as opposed to just read permission in MSI — thereby eliminating coherence traffic on a subsequent write to the same address. The synthesis lectures [25] describe this protocol in terms of new scenarios and modifications to scenarios in the MSI protocol. Our approach was to add the corresponding snippets to the existing set of snippets used to specify the MSI protocol. Because the examples describe a MESI protocol with a non-blocking directory, we modified our baseline MSI protocol correspondingly.

The extended protocol contained five new states (four for the cache, one for the directory), and seven new message types. In the first iteration, we added 19 snippets to specify transitions involving the *E* state and the non-blocking behavior of the directory. These snippets described the behavior of the protocol in under-specified corner cases and scenarios involving transient states and were added in response to the errors reported by the model checker. The programmer was able to obtain a fully verified protocol by adding twelve additional snippets over eight iterations. Additional metrics gathered during this case study are presented in Table 5.

6.3 Case Study C: The SGI Origin Protocol

For our final case study, we chose the coherence protocol used in the SGI-Origin 2000 servers [16], which is highly cited in the cache coherence literature. The Origin protocol is a directory-based, MESI protocol, and it supports multiple concurrent requests to the same address. Processes communicate through messages that may be arbitrarily re-ordered in the network. The consequent race conditions made it an interesting candidate for this case study.

Laudon and Lenoski [16] describe the common case protocol behavior using request flows. In this experiment, ignoring the "poisoned" directory state (used for page-migration), we transcribed each of the read, read exclusive, upgrade, and write-back flows using symbolic snippets in TRANSIT. Except for obvious cases, we left most of the guards empty and specified all conditional attributes on message fields and process variables with pre-conditions.

The protocol skeleton comprised of the cache process and directory processes, four request types, twelve response types, the request and response networks, and an intervention network used to buffer intervention requests. We initially specified 56 transitions in the cache machine and 18 transitions in the directory machine. We also specified the guards in instances where the incoming message type was found to be inconsequential; doing so prevented the tool from exploring artificially large expressions involving the disjunction of these enumerated types. The resulting protocol resulted in an error discovered by the model checker due to the cache process receiving an unexpected message. We fixed this case by adding a concrete snippet describing the desired behavior of the cache. Once again we left the guards unspecified, but the pre-conditions and update constraints were predicated by identical values for the input message fields and internal process variables, as seen in the violating trace.

Continuing similarly, we added concrete snippets to fix error traces. In some cases, the tool identified inconsistencies between the added trace and a pre-existing constraint. We found it straightforward to reconcile these differences before converging to a protocol that model checked. The final synthesis step took a little over 30 minutes, exploring over four million states during model checking. The generated TRANSIT specification had a total of 50 transitions.

6.4 Discussion and Limitations

We found the primary convenience of using TRANSIT to be the manner in which the initial specification phase and the iterative debugging phases could be expressed differently. Although it was natural to transcribe the bulk of the protocol symbolically from the algorithmic description of flows, corner cases invariably resulted. Most errors occurred due to unintended interactions between flows. The unexpected message condition cited above resulted from a cache process that was participating in a read-write-back race scenario. TRANSIT generalized the concrete fixes provided by the programmer in a manner that was guaranteed not to contradict the constituent flows. Fixing this bug symbolically would have required reasoning about the impact on both these flows. Similarly, another coherence violation was the result of the sharer set in the directory being updated incorrectly when a previous owner was downgraded. Again, the fix involved adding a snippet that concretely specified the next contents of the sharer set with the pre-condition specifying only the erroneous case.

Similar to other synthesis approaches, one limitation of TRANSIT is the readability of the produced code. Although expression size might sometimes approximate desirable metrics (like gate count in hardware descriptions), it can result in less intuitive code. For instance, consider the following code generated by TRANSIT in one of our case studies:

```
SetSize(SetAdd(Sharers, InMsg.Sender)) - 1
```

Although this code is more is compact than the equivalent, but more verbose:

```
if-then-else(SetContains(Sharers, (InMsg.Sender)),
             SetSize(Sharers) - 1,
             SetSize(Sharers))
```

the latter is more intuitive. Similarly, aggressive combining of guard-action blocks sometimes resulted in guard expressions that contained conjunctions of several different message types, making the generated code more difficult to read (although not less efficient than what a programmer would write).

	Case Study A	Case Study B
Snippets in the first/last version	19/86	96/108
Writing first set of snippets	2 hrs	6 hrs
Total manual effort	6 hrs	13 hrs
Number of iterations	13	8
Number of traces inspected	5	6
Number of updates/guards inferred	175/80	260/74
States in verified protocol	1.48M	1.5M

Table 5. Effectiveness Metrics for Protocol Design

7. Related Work

Broadly speaking, the topic of this paper can be classified as *program synthesis*, an active area of research with numerous approaches. We limit the comparison of our work to those that we feel are most closely related.

Methodologically, our approach is inspired by *program sketching* [23] and template-guided program synthesis [26]. In such approaches, a programmer specifies the scaffold of a program by listing the high-level structure, the variables, the functional specification as a pre and post condition, and the form of desired assignments inside the specified structure. The desired expressions are then formalized as terms with unknown parameters, whose values are computed using constraint solving techniques analogous to the ones used for automatic derivation of program invariants.

The problem of inferring an expression that matches a given set of concrete examples is addressed by the work by Gulwani on a variety of domains including end-user spreadsheet programming [7]. Although they do not use symbolic examples, their strategy for concrete examples, based on *version space algebra*, is different from the approach we adopt: the algorithm computes a representation of the set of all expressions that are consistent with each example, achieves compactness of the representation by a judicious use of data structures and pruning based on domain knowledge, and constructs the product of all such representations to choose an expression that is consistent with all the examples.

The problem of inferring expressions consistent with concrete and symbolic snippets can be viewed as an instance of the *counterexample-guided inductive synthesis* (CEGIS) strategy [8, 13, 23]. As mentioned earlier, the problem is analogous to solving an $\exists\forall$ formula. The solutions described in prior work [8, 13] explicitly encoded the problem into an SMT constraint, which was then solved using an SMT solver. The approach described in this work uses an enumerative technique to handle the existential quantifier and uses an SMT solver to check for universality.

Distributed protocols, such as cache coherence protocols, have been the canonical application for model checking [3]. In *reactive synthesis*, a finite-state controller is derived from correctness requirements in temporal logic, but this problem becomes undecidable when synthesizing a distributed protocol (see [28] for a survey and [6, 21] for recent approaches aimed at coping with the high computational complexity of the synthesis problem). An interesting recent approach to distributed protocol design relies on genetic programming [14]: given an initial protocol and correctness requirements, if the model checker finds that the protocol does not satisfy the requirements, the tool tries multiple mutations of the guards and updates used in the protocol, ranks the resulting versions by estimating how close they are to satisfying the requirements using state-space analysis, and iterates by probabilistically

selecting a variant with weights proportional to ranks. Note that we require the programmer to specify the skeleton of the protocol (such as control states and state variables), and thus we focus not on deriving protocol logic from high-level requirements, but on providing assistance to complete the intended design correctly.

8. Conclusions

In this paper, we have proposed an approach for specifying distributed protocols by adopting verification tools that interact with the programmer. The approach relies on the observation that parts of the protocol behavior, as well as fixes during the debugging phase, can be naturally expressed in terms of example snippets. We described the concept of *concolic* snippets to allow a programmer to specify the protocol behavior as a mix of concrete examples and symbolic partial transitions. For the computational problem of inferring an expression that is consistent with the given set of concolic examples, we presented an algorithm that significantly prunes the space of expressions that need to be considered, and also limits the number of calls to an SMT solver needed to check consistency with respect to symbolic constraints. To demonstrate the feasibility of our methodology, we developed a prototype tool based on the algorithm for expression inference that generates complete protocol specifications from concolic snippets, which are then verified using a model checker.

Our preliminary case studies using this tool allowed inexperienced programmers to correctly synthesize representative cache coherence protocols of modest complexity with several hours of human effort. We were also able to translate an incomplete flow-based description of an industrial-strength protocol into a working implementation by effectively exploiting the flexibility afforded by concolic specifications. Encouraged by the initial experimental results, our next steps are to explore techniques to automatically analyze counterexamples returned by the model checker and alternate strategies for expression inference, and synthesizing EFSM descriptions from distributed scenarios such as message sequence charts.

References

[1] Intel Core2 Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 and E4000 Sequence — Specification Update, 2003. URL `http://www.intel.com/design/processor/specupdt/313279.htm`.

[2] D. Abts, D. J. Lilja, and S. Scott. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS '03, pages 1–11, 2003.

[3] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. .Long, K. McMillan, and L. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[4] N. Dave, M. C. Ng, and Arvind. Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. In *Formal Methods and Models for Codesign*, MEMOCODE '05, pages 25–34, 2005.

[5] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of the International Conference on Computer Design*, ICCD '92, pages 522–525, 1992.

[6] B. Finkbeiner and S. Jacobs. Lazy Synthesis. In *13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '12, LNCS 7148, pages 219–234, 2012.

[7] S. Gulwani. Automating String Processing in Spreadsheets using Input-output Examples. In *Proceedings of The 38th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, 2011.

[8] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, 2011.

[9] D. Harel. Can Programming Be Liberated, Period? *IEEE Computer*, 41(1):28–37, Jan. 2008.

[10] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag New York, 2003.

[11] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100, Jul. 2012.

[12] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, 2010.

[14] G. Katz and D. Peled. MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming. In *8th Internation Symposium on Automated Technology for Verification and Analysis*, ATVA '10, LNCS 6252, pages 359–364, 2010.

[15] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software Synthesis Procedures. *Communications of the ACM*, 55(2):103–111, Feb. 2012.

[16] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 241–251, 1997.

[17] N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[18] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, Nov. 2005.

[19] L. D. Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08, pages 337–340, 2008.

[20] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '05, pages 263–272, 2005.

[21] S. A. Seshia. Sciduction: Combining Induction, Deduction, and Structure for Verification and Synthesis. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 356–365, 2012.

[22] R. Singh and A. Solar-Lezama. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM Symposium on Foundations of Software Engineering*, FSE '11, pages 289–299, 2011.

[23] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by Sketching for Bitstreaming Programs. In *Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI '05, pages 281–294, 2005.

[24] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching Concurrent Data Structures. In *Proceedings of the SIGPLAN 2008 Conference on Programming Language Design and Implementation*, PLDI '08, pages 136–148, 2008.

[25] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan Claypool, 2011.

[26] S. Srivastava, S. Gulwani, and J. S. Foster. From Program Verification to Program Synthesis. In *Proceedings of the 37th annual ACM Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, 2010.

[27] M. Talupur and M. R. Tuttle. Going with the Flow: Parameterized Verification using Flows: An Industrial Experience. In *Proceedings of the 2008 Internation Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 1–8, 2008.

[28] W. Thomas. Facets of Synthesis: Revisiting Church's Problem. In *Foundations of Software Science and Computational Structures, 12th International Conference*, FOSSACS '09, LNCS 5504, pages 1–14, 2009.