Protocol Design with Concolic Snippets

Rajeev Alur, Jyotirmoy V. Deshmukh, Sela Mador-Haim,
Milo M. K. Martin, Arun Raghavan, and Abhishek Udupa

University of Pennsylvania, Philadelphia PA 19104, USA

Abstract. With the maturing of computer-aided verification technology, there is an emerging opportunity to develop design tools that can transform the way systems are designed. In this paper, we propose a new way to specify protocols using *concolic* snippets, that is, sample execution fragments that contain both concrete and symbolic values. While the purely symbolic extreme is simply an alternative representation of the traditional communicating extended finite-state-machines, and the purely concrete extreme is an instantiation of the "programming by examples" paradigm, our specification language allows the designer to specify the desired protocol using a mixture of symbolic state machines and concrete scenarios. Our synthesis engine generalizes the snippets into a transition function, which is then analyzed using a model checker with respect to high-level temporal-logic correctness requirements. We describe a prototype implementation for design of cache coherence protocols built using (1) a straightforward enumeration of all expressions for transition functions, (2) a check for consistency with respect to concolic snippets using the SMT solver CVC3, and (3) a check for correctness using the model checker Murϕ. We discuss our experience in designing classical cache coherence protocols using the proposed methodology.

1 Introduction

Over the last two decades, computer-aided verification technology has matured, and has witnessed growing adoption in industry. However, verification tools have been confined to discovering bugs in systems that have already been designed. This raises the question: how can we leverage the advances in verification tools by interacting with the designer and assisting the production of a "correct" design? One approach in this direction is based on *synthesis*, where a synthesis tool maps a *declarative specification* to an *executable implementation* [3, 17, 24]. Another approach is based on *programming by examples*, where a synthesis tool maps a set of example input/output traces to either an executable finite-state machine [9, 10], or a functional program [13] operating on strings [8]. In both approaches, the designer is still required to express the complete logic of the system in the chosen level of abstraction. Further, while the input formats such as logical formulas and examples differ from traditional imperative programming languages, there is little compelling evidence that these alternative formats reduce the effort for getting the tricky details "right."

A perhaps more promising approach is the recent work on *sketching*, where a programmer writes a partial program with incomplete details. A synthesizer then fills in missing details using user-specified assertions as correctness specifications [19–21]. A key methodological innovation in sketching is that it allows the programmer to mix imperative and assertional styles. Therefore, high-level invariants can be used to find intricate details necessary for fine-tuning code, without needing the programmer to abandon familiar programming styles.

While sketching has been demonstrated for functional programs [19, 20], we extend the view of the synthesis tool as an *integrator* of multiple formats to develop a new approach for designing *reactive systems*, such as distributed protocols, focusing on cache coherence protocols in this paper. Such protocols are interesting candidates for synthesis because (a) they are challenging to design correctly due to corner cases arising from asynchronous concurrency [1, 25], with bugs being reported even in published protocols [12], (b) they possess a well-defined structure with relatively few variables and small bounds for values, allowing for well-scoped and tractable definition of the synthesis problem, and (c) previous work has been successful on using model checkers to find bugs in such protocols [4, 7], and was in fact instrumental for adoption of model checking by the electronic design automation industry.

In our proposed method, a designer describes a distributed protocol using high-level temporal requirements and *concolic* snippets, where a concolic snippet is a fragment of desired execution that contains 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 [18]). The synthesis tool then generates a complete protocol, in the form of a set of communicating extended finite-state machines (EFSMs), such that the synthesized description is consistent with the snippets. The synthesized protocol is further checked against temporal requirements; if any of these requirements are violated, the resulting counterexample traces are used by the designer to supply additional snippets. Every EFSM transition, usually described using a guard and update action, can be viewed as a symbolic snippet; thus, a traditional protocol description using EFSMs can be directly input in our format.

When all snippets use only concrete values, our methodology is an instantiation of the "programming by examples" approach, where the synthesis tool assists the designer by generalizing from concrete examples to symbolic expressions for guarded updates of transitions, with the model-checker in the design loop. However, it offers flexibility to mix the two styles. Use of symbolic expressions can reduce the number of examples required for suitable generalization, and also ensures that we do not make the design process more cumbersome by insisting on describing every detail with concrete values.

In our prototype implementation, the designer describes the protocol by listing the communication architecture, constituent processes with their control states and variables, concolic snippets corresponding to EFSM transitions, a typed grammar for allowed expressions in guards and updates, and temporal logic requirements. For each EFSM transition, the synthesis tool systematically enumer-

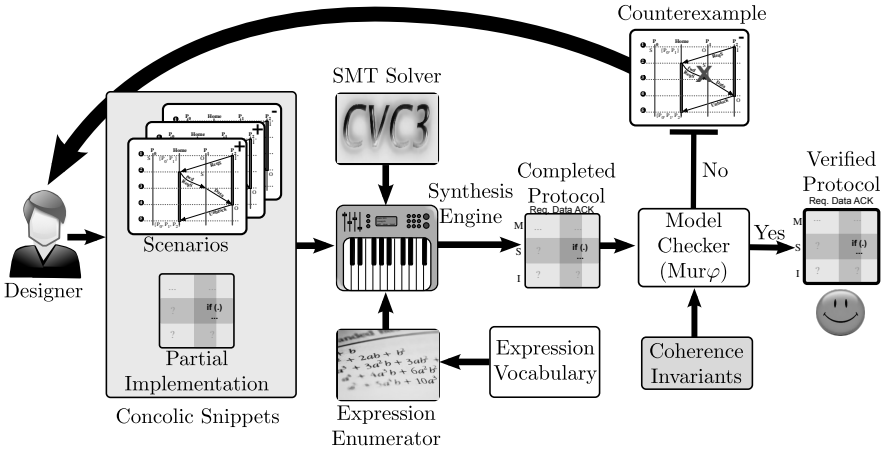


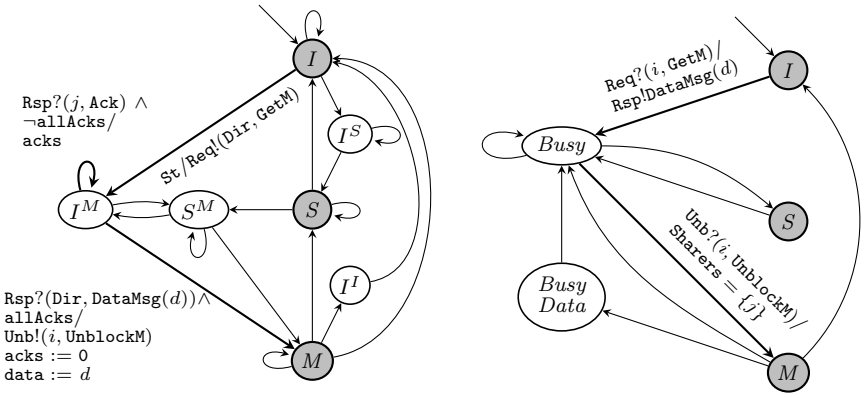
Fig. 1. Flowchart for Design Methodology using Concolic Snippets

ates possible expressions for guards and updates using the specified grammar, with optimizations to reduce the number of expressions that need to examined. Our straightforward enumeration mechanism was guided by preliminary investigations on the size of expressions in representative coherence protocols, including published broadcast-based protocols that are known to have a large number of states [14, 16]. Checking whether the completed description of a transition is consistent with all the snippets is formulated as a validity query discharged using the SMT solver CVC3 [2]. When consistent completions for all transitions are discovered, the resulting protocol is checked with respect to the temporal requirements using the model checker Murϕ [5]. If the check fails, the designer debugs the reported counterexample with a visualizer, and adds more snippets to rule out the erroneous behavior. Figure 1 illustrates the proposed methodology.

We evaluated the feasibility of our approach by first specifying protocols previously defined in SLICC [15], with the help of concolic snippets. We found that even the largest expression (size 8) was synthesized within tens of seconds by systematic enumeration. Encouraged by our feasibility results, we performed two case studies where designers with no experience with coherence protocols used the prototype tool to design two canonical cache coherence protocols from textbook illustrations. Both case studies resulted in generating protocols that were successfully verified by the model checker. We found that the final, correctly generated protocol required an average of less than 2 snippets per transition. Further, actually synthesizing the protocols from snippets took less than an hour, while the entire iterative design process required at most 15 hours of manual effort for either case study.

2 Background: Protocols as Communicating EFSMs

In this section, we review the traditional design methodology for communication protocols. The standard description of protocols uses the model of communicat-



(a) The i^{th} Cache Process **(b)** Directory Process

Fig. 2. The MSI protocol specified as communicating EFSMs

ing sequential processes, where each process is described as an extended finite state machine (EFSM). For instance, Promela models that serve as inputs to the Spin model checker [11], the inputs to the Murϕ model checker [6] and domain specific languages such as SLICC [23] have constructs that enable specification of protocols as communicating EFSMs. Protocol correctness is verified against temporal logic requirements by model checking.

2.1 Protocol Design

We explain the basic terminology for communicating EFSMs using the example of an invalidation-based directory cache coherence protocol. A cache coherence protocol ensures that data in private caches of processors in a shared memory multiprocessor configuration remain consistent; the value read by a *load* operation must be the value written by the last preceding *store* operation to that address. Most commodity server, desktop and even mobile processors today tend to have multiple cores and commonly implement cache coherence in hardware.

Coherence protocols typically enforce the consistency constraint using permissions — each address in shared memory may have either multiple readers or a single writer at any given time. Read and write permissions are associated with every address and are determined by exchanging messages between processes. In a simple directory-based protocol, a global directory process (D) tracks all processes that cache a memory block (denoted $C_1, \ldots, C_n$). Processes communicate using point-to-point messages to maintain coherence.

Communication model. Coherence protocols typically use an *asynchronous, message-passing* based model of communication, *i.e.*, processes communicate by sending messages on channels or *networks*. Each process description includes a list of input networks to receive messages, and output networks on which messages are sent. For example, the input network for the cache process C_i is `Rsp`, and its output networks are `Req` and `Unb`, while for the directory, `Req`

and `Unb` are input networks and `Rsp` is an output network. Note that some networks can be both input and output networks. A network can be thought of as a multiset; to send a message process C_i inserts it into the multiset, and if the destination field for the message is C_j, then C_j will eventually receive this message. This essentially enables any process to communicate with any other process.

EFSM description. Figure 2a and Figure 2b show how the cache and directory processes respectively are specified as EFSMs. Due to space constraints, we only specify the labels for the bold edges in the figure. An EFSM is specified as a tuple consisting of the following components:

Control states identify a logical state of the protocol. In Figure 2, the control states for each cache C_i are M, S, I, I^M, S^M, I^I, and I^S, while those for the directory are M, S, I, *Busy*, *Busy Data*. The M, S and I states in the cache are *steady states*, and define the read-write permissions; in M state, a cache block can be read or written, in S, it can be read but not written, while in I, a cache block can be neither read nor written. It follows that when a cache process is in M, all other cache processes must remain in I. States such as I^S or I^M are *transient* states and are used when a process wishes to transition between two steady states to acquire or relinquish permissions.

Process variables are variables local to a process, such as the `Sharers` variable for the directory and `Acks` and `Data` variables for the cache. Each process variable has a defined type, *e.g.*, the `Sharers` variable has the type `Set` and the `Acks` variable has type `Int`.

Input events either correspond to reception of messages or external events. E.g., in Figure 2a the transition from I^M to I^M is triggered by the event `Rsp`?(`i`, `Ack`), denotes that `Ack` was received on the `Rsp` network for C_i, while the transition from I to I^M is triggered by the external store (`St`) event.

Guards are Boolean predicates over fields of received messages and process variables. For instance, the receipt of a `Data` message in the I^M state, causes a transition to M if the `allAcks` predicate is true and leaves the state unchanged otherwise.

Output events correspond to sending messages. E.g., as a result of `St` event in the I state, process C_i sends a $GetM$ message on its output network `Req`.

Actions correspond to updates to the process variables. For instance, after receiving an `Ack` message in state I^M, the process variable `Acks` is appropriately decremented. Similarly, upon receiving an `UnblockM` message in state *Busy*, the `Sharers` variable of the directory process is set to the sender's process id.

2.2 Protocol Verification

We check the safety property of coherence by assigning read/write permissions to the control states and maintaining a copy of every stored data value in a global *oracle*. We then check if in every readable state the value read is the same as the value maintained by the oracle. Absence of deadlocks can be viewed as a safety property: a control state in which none of the guards is enabled should

```
Transition(CurrentState, InputEvent){
  [guard] => (NextState, OutMsg1, ...){
      antecedent_1 ==> {
          var_11 = expr_11
          var_12 = expr_12
          ...
      }
      ...
  }
}
```

Fig. 3. Structure of a concolic snippet. *CurrentState* and *NextState* are the start and end control states. The snippet specifies zero or more outbound messages that may be sent conditionally. It also specifies a *guard-action* block for each *guard* containing a set of conditional updates. The expression antecedent_i specifies the condition (on process variables and the fields of a received message) under which variables var_{ij} are updated with expressions expr_{ij}.

be unreachable. In addition to safety invariants, a well-behaved coherence protocol also provides forward progress guarantees such as deadlock and starvation avoidance to ensure that every request eventually succeeds, or every process in a transient state eventually returns to a steady state.

3 Protocol Design using Concolic Snippets

Rather than designing a coherence protocol by fully specifying communicating EFSMs, we propose an alternative approach based on the observation that it is often natural to describe a protocol in terms of its behavior in different scenarios. Figure 1 illustrates the various steps in our approach. The designer specifies a protocol using a set of *snippets* derived from scenarios describing the protocol behavior. In addition to the snippets, the designer also provides contextual declarations of the terms used in the snippets, such as the number of processes and an enumeration of possible control states.

Our synthesis tool (called TRANSIT) decomposes the problem of synthesizing a complete EFSM description from snippets into smaller sub-problems of: (a) synthesizing process-variable update expressions (*i.e.*, RHS of assignments), and (b) synthesizing guards for transitions. The central component of TRANSIT is an expression enumerator that generates candidate expressions for guards and updates which are checked for consistency with the snippets by an SMT solver.

The completed protocol (a fully specified set of communicating EFSMs) synthesized by TRANSIT is then translated to a formal model and verified for coherence and liveness properties by the Murϕ model checker. An error in model checking results in a violating trace, or counterexample. Upon inspecting the violating trace, the designer can add a *positive* example that illustrates the correct behavior for the violating trace to the original set to of snippets. The newly generated protocol completion would therefore no longer result in the earlier incorrect behavior. The entire process is repeated until the model checking phase verifies the generated protocol to be correct.

3.1 Structure of Protocol Snippets

A protocol snippet describes the behavior of a single transition of a process in the protocol. In particular, given a specific control state, input event (such as receiving a message or an external event), and guard (Boolean condition on

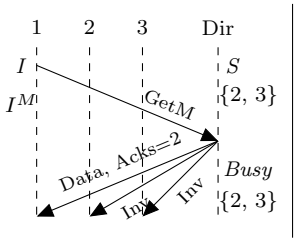
variables and received message fields), the designer specifies the actions executed by the protocol, such as updating the process variables, sending messages, and finally, change in control state. Unlike an EFSM, where a transition is specified symbolically, *i.e.*, the input event, the guard, and actions are fully specified, in a snippet, we allow a combination of concrete, symbolic, and in some cases unspecified terms. A concolic snippet is formally specified using the template shown in Figure 3.

3.2 Specifying Snippets

In any snippet, *CurrentState*, *InputEvent*, *NextState*, and *OutMsg* are concretely specified. In a concrete snippet, antecedents and update expressions are also specified concretely. A concrete snippet therefore constitutes a single transition in the actual execution of the protocol. Figure 4b shows a concrete protocol snippet describing the expected actions of the directory in control state S, when it receives an input request (`GetM`) from the cache process 1 (illustrated in Figure 4a). The designer concretely specifies the fields of the outgoing response message (`RspMsg`), indicating the number of `Ack` messages the requester must expect. Similarly, the designer specifies that invalidation messages must be sent to cache processes 2 and 3.

The goal of the synthesis tool is to generate expressions for these fields in the outgoing messages from the given snippets. Figure 4c shows the corresponding expressions generated by TRANSIT. The key difference between the generated expressions and the expressions in the snippet is that the generated expressions are *symbolic*. As the synthesis tool accepts symbolic input snippets as well, the generated expression is itself a valid input snippet. In the limit, a fully specified protocol can be expressed as a set of symbolic snippets, where each snippet fully specifies a transition corresponding to an arc in Figure 2. This observation allows for incremental protocol design by iteratively adding snippets to previously generated protocols, or the snippets that were used to generate them. We define the term *concolic snippet* for snippets in which expressions can contain a mix of concrete and symbolic expressions.

To illustrate the iterative refinement process, we continue with the example in Figure 4. We find that the protocol completion using the generated expressions results in a deadlock when verified with Murϕ. The counterexample from the resulting execution indicates a scenario where the requesting cache process itself was a member of the directory's sharer set (called an *upgrade* miss). The conventional EFSM approach to fixing such a bug would require the designer to reason about all the expressions that are affected by such a scenario and deduce the changes required to fix them symbolically. Instead, with our approach, the designer adds an example that indicates the desired behavior in this scenario. Figure 5a and Figure 5b show the example case and the corresponding snippet respectively. When TRANSIT is invoked with this additional snippet, it generates the new expressions shown in Figure 5c. The protocol completion obtained with these newly generated expressions is successfully verified by Murϕ.



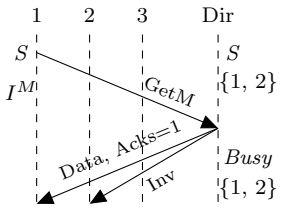

```
Transition(S, ReqMsg){
  [] => (Busy, RspMsg, InvMsg){
     (ReqMsg.Sender=1  ∧
      ReqMsg.Type=GetM ∧
      Sharers = {2, 3}) ==>{
          RspMsg.Acks=2;
          ...
          InvMsg.Dest={2, 3};
     }
  }
}
```

```
RspMsg.Acks := |Sharers|;
```

```
InvMsg.Dest := Sharers;
```

(a) scenario **(b)** snippet **(c)** generated expression

Fig. 4. Figure 4b shows the example snippet given by the designer for the Directory process transition corresponding to the scenario shown in Figure 4a. Figure 4c shows the expressions generated by the synthesis tool for two message fields.




```
Transition(S, ReqMsg){
  [] => (Busy, RspMsg, InvMsg){
     (ReqMsg.Sender=1  ∧
      ReqMsg.Type=GetM ∧
      Sharers = {1, 2}) ==>{
          RspMsg.Acks=1;
          ...
          InvMsg.Dest={2};
     }
  }
}
```

```
RspMsg.Acks :=
  |Sharers∪Req.Sender|-1;
```

```
InvMsg.Dest :=
  Sharers \ Req.Sender;
```

(a) scenario **(b)** snippet **(c)** generated expression

Fig. 5. Figure 5a and Figure 5b show an additional snippet illustrating the expected behavior for a counterexample corresponding to a deadlock resulting from the generated expression in Figure 4c. The newly generated expression Figure 5c eliminates the deadlock.

Problem Definition. In order to generate a protocol completion, *i.e.*, the symbolic snippets which are consistent with the concolic snippets provided by the designer, we need to solve two sub-problems:

- Infer the symbolic expressions e_i' for the RHS of each update expression.
- If a guard g is concolic or unspecified, then infer the guard g from the antecedents provided by the designer.

The above problem statements imply that the generated expressions are consistent with the concolic examples for updates and their respective antecedents specified in every input snippet. As input snippets can be either concrete or symbolic in nature, we employ an SMT solver to check this implication.

4 Implementation of the Design Tool

To solve the problems described in § 3.2, our implementation of TRANSIT is based on generating candidate expressions and checking if they are compatible with the given concolic snippets. The tool systematically enumerates expressions of increasing sizes based on the types of variables and constants used in the given

snippets, and the operators on these types that are defined in our expression grammar.

4.1 Expression Grammar

An expression grammar G is defined as the tuple $(\mathcal{F}, V, C)$ over a set of types $\mathcal{T}$, where $\mathcal{F}$ is a set of typed function symbols with specified arities, V is a set of typed variable symbols and C is a fixed set of typed constant symbols. Let $e_1, \ldots, e_k$ be expressions, let $f_k \in \mathcal{F}$ be a function symbol of arity k, let $v \in V$ be a variable, and $c \in C$ be a constant symbol. Further, assume that the types of the expressions $e_1, \ldots, e_k$ are respectively the same as the types of the arguments of f_k. A well-formed expression e is then inductively defined as: $e ::= f_k(e_1, \ldots, e_k) \mid v \mid c$. The size of an expression is the number of symbols belonging to $\mathcal{F}$, V or C appearing in the expression. For example, consider the expression grammar G with $\mathcal{T} = \{\texttt{Int}, \texttt{Set}\}$, $\mathcal{F} = \{+, \cup, \texttt{SetOf}\}$, $V = \{x, y\}$, and $C = \{0\}$. G includes the expressions $\texttt{SetOf}(+(+(x, x), y))$ (of size 6) and $\cup(\texttt{SetOf}(0), \texttt{SetOf}(+(x, x))$ (of size 7). We use G_i to denote a restriction of the expression grammar G that only contains expressions that have a size exactly equal to i.

To synthesize expressions for cache-coherence protocols, we found that we needed the following set of types:

$$\mathcal{T} = \{\texttt{Int}, \texttt{Set}, \texttt{Bool}, \texttt{PID}, \texttt{Address}, \texttt{Value}\} \cup \texttt{EnumTypes}$$

`Int` and `Bool` have the usual meaning of integer and Boolean types. Each EFSM in a protocol has a unique identifier, and the type `PID` specifies its type. Any set of `PID` types belongs to the `Set` type. The `Address` type represents addresses in the memory and the `Value` type represents values stored in the memory. `EnumTypes` represents the set of user-defined enumerated types, primarily used to specify message types.

4.2 Synthesizing Update Expressions

The goal of the synthesis tool is to compute expressions for the variables in guard-action blocks as shown in Figure 3. We assume the standard parallel assignment model (static single assignment, or SSA form) so that expressions for each variable, renamed as *primed variables*, can be synthesized independently. For each output variable v, if $\mathit{type}(v) \in \mathcal{T}$ denotes the type of v, then each concolic snippet specifies a (conditional) concolic assignment to the variable v of the form: $[a_j \implies v = e_j]$, where a_j is the antecedent condition, and e_j is a concolic expression. Assume that G is an expression grammar where $\mathcal{F}$ and C are user-specified, and V_{in} is the set of all unprimed process-variables and fields of the received message. The synthesis algorithm systematically enumerates expressions from each G_i (starting from $i = 1$), up to some user-specified bound. For each enumerated expression g, we check if g is a valid generalization of the concolic assignments. Concretely, we check the validity of the formula:

$$\bigwedge_{j=1}^{n} ((a_j \wedge (v = g)) \implies (v = e_j))$$

If the formula is valid, then the expression g is the *smallest* expression consistent with each of the concolic RHSs for v. This process is repeated for all the variables $v \in V_{out}$ and for every guard-action.

To illustrate the discussion presented above, consider the example shown in Figure 4b. Suppose that we desire to check if the expression $|\texttt{Sharers}|$, fits the examples for the update to `RspMsg.Acks`, then the proposition to be checked for validity is

$$\left(\begin{array}{r} \texttt{Sharers} = \{2,3\} \wedge \\ \texttt{ReqMsg.Sender} = \quad 1 \quad \wedge \\ \texttt{ReqMsg.Type} = \texttt{GetM} \end{array}\right) \wedge (\texttt{RspMsg.Acks} = |\texttt{Sharers}|) \implies \texttt{RspMsg.Acks} = 2$$

It can be verified that this proposition is indeed valid given that the value of `Sharers` is constrained in the antecedent. Thus, this expression fits the examples provided for the update to `RspMsg.Acks`.

Now, when the additional example in shown in Figure 5b is added, and the expression $|\texttt{Sharers}|$ is tried, the following proposition will be checked for validity

$$\left(\begin{array}{r} \texttt{Sharers} = \{2,3\} \wedge \\ \texttt{ReqMsg.Sender} = \quad 1 \quad \wedge \\ \texttt{ReqMsg.Type} = \texttt{GetM}) \end{array}\right) \wedge (\texttt{RspMsg.Acks} = |\texttt{Sharers}|) \implies \texttt{RspMsg.Acks} = 2$$

$$\wedge$$

$$\left(\begin{array}{r} \texttt{Sharers} = \{1,2\} \wedge \\ \texttt{ReqMsg.Sender} = \quad 1 \quad \wedge \\ \texttt{ReqMsg.Type} = \texttt{GetM}) \end{array}\right) \wedge (\texttt{RspMsg.Acks} = |\texttt{Sharers}|) \implies \texttt{RspMsg.Acks} = 1$$

This proposition is invalid, since $\texttt{Sharers} = \{1,2\} \implies |\texttt{Sharers}| = 2$, which violates the consequent in the second example. It can be verified that the expression $|\texttt{Sharers} \cup \{\texttt{ReqMsg.Sender}\}| - 1$ is valid for these set of examples.

4.3 Synthesizing Guards

A guard can be simply viewed as a Boolean-valued expression. However, the key difference between synthesizing guards and synthesizing update expressions is that for a given control state and input event, guards cannot be inferred independently of each other. To ensure deterministic execution, we require that synthesized guards are mutually exclusive. For a set of guard-actions $B_1, \ldots, B_n$ from a set of concolic snippets with the same starting control state and input event type, the j^{th} guard-action block B_j consists of a sequence of examples conditioned by antecedents $a_{j1}, \ldots, a_{jk}$. The synthesis algorithm sequentially synthesizes the guards for the blocks B_1 to B_n, 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.

Similar to the synthesis procedure for update expressions, we systematically enumerate Boolean-valued expressions compatible with the grammar G_i (beginning with $i = 1$), and check if the enumerated expression g is (1) compatible with the disjunction of the antecedents in the current guard-action block, (2) incompatible with the guards synthesized previously, and (3) incompatible with the antecedents in the subsequent guard-action blocks. Any expression g that satisfies these three conditions is guaranteed to be mutually exclusive with all the other guards for the given control state and input event. Concretely, we define $\mathfrak{A}_i = \bigvee_{i=1}^{k} a_{ji}$ to be the disjunction of the antecedents in B_j, and check the validity of the following formula:

$$(\mathfrak{A}_j \implies g) \wedge \left(\bigwedge_{i=1}^{j-1} (\varphi_i \implies \neg g) \right) \wedge \left(\bigwedge_{i=j+1}^{n} (\mathfrak{A}_i \implies \neg g) \right)$$

Again, to illustrate the discussion presented, consider the following set of antecedents which occur in a real world protocol, where two guards need to be synthesized. Note that the guard-action blocks have been labeled with B_i and antecedents of the examples have been labeled with a_{ij} for future reference.

```
Transition(SM, RspNet InMsg) {
      B1: [] => (SM) {
          a11 : (InMsg.MType=ACK_I) ∧ (InMsg.Acks=-1) ∧ (Pending=-2) ==> {···};
          a12 : (InMsg.MType=ACK_I) ∧ (InMsg.Acks= 3) ∧ (Pending= 0) ==> {···};
          a13 : (InMsg.MType=ACK_I) ∧ (InMsg.Acks=-1) ∧ (Pending= 0) ==> {···};
      }
      B2: [] => (M) {
          a21 : (InMsg.MType=ACK_I) ∧ (InMsg.Acks=-1) ∧ (Pending=-1) ==> {···};
          a22 : (InMsg.MType=ACK_I) ∧ (InMsg.Acks= 3) ∧ (Pending= 3) ==> {···};
      }
}
```

Suppose that we were trying to synthesize an expression for the guard of the guard-action block B_1. In this case, we have $\mathfrak{A}_1 = a_{11} \vee a_{12} \vee a_{13}$. No guards have yet been synthesized, so none of the φ_i propositions are defined for this example. Also, we have $\mathfrak{A}_2 = a_{21} \vee a_{22}$. Consider the case where the expression `InMsg.Acks` $\neq$ `Pending` is being evaluated as a candidate expression for the guard of B_1. The proposition that will be tested for validity is

$$\begin{array}{rcl} a_{11} \vee a_{12} \vee a_{13} & \implies & (\texttt{InMsg.Acks} \neq \texttt{Pending}) \\ & \bigwedge & \\ a_{21} \vee a_{22} & \implies & \neg(\texttt{InMsg.Acks} \neq \texttt{Pending}) \end{array}$$

It can be seen that `InMsg.Acks` $\neq$ `Pending` is consistent with a_{11}, a_{12} and a_{13}, while the negation `InMsg.Acks` $=$ `Pending` is consistent with a_{21} and a_{22}. Thus the expression `InMsg.Acks` $\neq$ `Pending` is a valid guard for B_1.

4.4 Optimizations for Expression Enumeration

As several generated expressions from our grammar can be equivalent, we eliminate common causes for equivalent expressions, to avoid redundant invocations

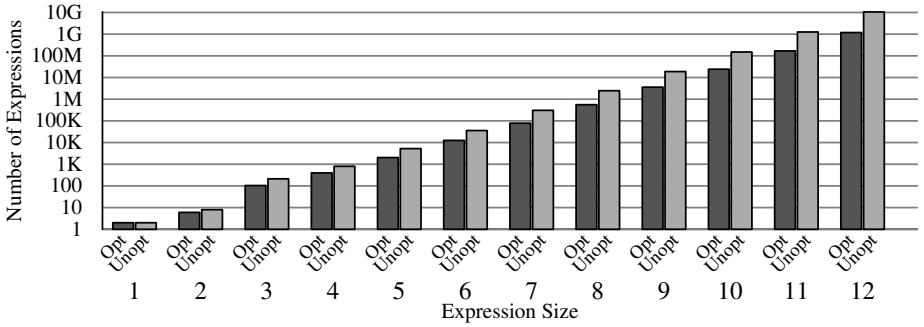


Fig. 6. Number of expressions per expression size, with and without optimizations enabled.

of the SMT solver. The protocol designer defines operators in the grammar to be associative and commutative when appropriate, and the expression generator chooses a canonical representation for such compositions. Similarly, we allow for annotation of operators whose repeated composition can lead to equivalent expressions (such as Boolean negation and idempotent functions). We also eliminate Boolean expressions of size greater than one consisting solely of constants. We found that these optimizations reduced the run-time of systematic enumeration by an order of magnitude, and allowed for the synthesis tool to consider expressions with larger sizes, within a given time budget.

5 Evaluation of the Proposed Design Methodology

In this section, we evaluate the performance of our synthesis tool TRANSIT and discuss the experience that we have had in designing coherence protocols using our approach.

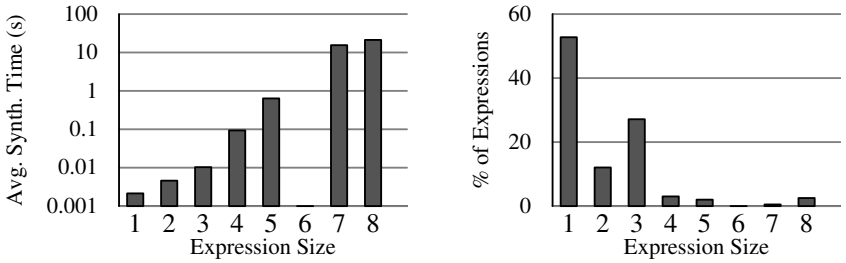
5.1 Performance evaluation

We first measure the limits of systematic expression enumeration. Figure 6 shows the number of Boolean-valued expressions enumerated for each size. The numbers for other types are similar. As expected, the number of generated expressions increases exponentially with expression size. Our optimizations reduce the number of enumerated expressions, particularly for large expression sizes, allowing for expressions of larger size — for example, the number of expressions of size 12 enumerated with optimizations is the same as the number of expressions of size 11 without optimizations.

Figure 7a shows the performance of the expression synthesis algorithm when synthesizing completions for the MSI protocol. Most of the synthesis time was spent in evaluating queries with the SMT solver. Figure 7b shows the distribution of the sizes of the expressions synthesized. Most of the expressions are of size 3 or lower, with fewer than 6% of the expressions having a greater size. As the average synthesis time for smaller expressions is several orders of magnitude lower, most of the time is dominated by the few larger expressions which needed to be synthesized.

Table 1. Performance of snippet-based design

Protocol	Snippets required	Synthesis: Updates Num. synth.	Updates Exprs. explored	Updates Time (secs)	Guards Num. synth.	Guards Exprs. explored	Guards Time (secs)	States explored by Murϕ
VI	19	49	88	0.1	17	3.1K	5	140K
MSI	77	157	1.8K	3.8	45	44.5K	134	854K



(a) Average synthesis time per size. **(b)** Distribution of expression size.

Fig. 7. Expressions synthesized when completing the MSI protocol from snippets.

To evaluate the feasibility of specifying cache coherence protocols with snippets, we manually expressed working versions (fully specified and verified EFSM descriptions) of two protocols — the Valid-Invalid (VI) protocol and the MSI protocol as concolic snippets. In each case, the protocol had 4 cache processes and one directory. The key results are summarized in Table 1 and Figure 7a. As the number of expressions explored during guard synthesis was much higher than that for updates, the synthesis time was dominated by guard synthesis.

5.2 Case Studies

Methodology. We investigated the effectiveness of designing protocols with concolic snippets by conducting two case studies, where we designed two representative coherence protocols presented in the synthesis lectures on computer architecture [22]. The lectures describe directory-based protocols with the help of illustrations of common-case scenarios, accompanied by a textual description of some key aspects. To simulate the experience of an inexpert user, we chose two of the co-authors of this work (henceforth called designers) who were inexperienced in cache coherence protocol design.

Case Study A: Non-blocking-Directory based MSI Protocol. In this experiment, we designed the MSI protocol described in the synthesis lectures from concolic snippets with TRANSIT. In contrast to the blocking-directory protocol in Figure 2b, this protocol has a non-blocking directory, *i.e.*, the directory responds to concurrent requests (for the same cache block). Consequently, there are several additional race conditions that result from this change. Another key difference is that this protocol includes *non-silent clean line eviction*, *i.e.*, when

Table 2. Effectiveness Metrics for Protocol Design from Concolic Snippets

	Case Study A	Case Study B
Snippets in the first/last version	19/86 snippets	96/108 snippets
Writing first set of snippets	2 hours	6 hours
Total manual effort	6 hours	13 hours
Number of iterations	13 iterations	8 iterations
Number of traces inspected	5 traces	6 traces
Synthesis time for last version	52 mins.	15 mins.
Number of updates/guards synthesized	175/80 expressions	260/74 expressions
States in verified protocol	7.7M states	1.5M states

it is in the S state, if a cache receives a replacement request, it does not silently transition to the I state, but has to introduce additional states and messages. In contrast, the protocol in Figure 2 implements silent evictions.

The designer observed that the scenarios described in the synthesis lectures 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 designer had to add 67 more snippets over 13 iterations to converge to a correct protocol. A summary of other key metrics of the case study is presented in Table 2.

Case Study B: Adding the *E* State to a Blocking MSI Protocol. The goal of the second case-study was to augment the MSI protocol in Figure 2 with an "E" state to arrive at the MESI protocol. The E state (shorthand for *exclusive*) is an optimization that allows the protocol to immediately grant write permissions for read requests to memory locations to which any other cache process has neither read nor write permissions. The synthesis lectures describe the MESI protocol in terms of new scenarios, and modifications to scenarios for the MSI protocol. Our approach was to add the corresponding snippets to the existing set of snippets specifying our baseline MSI protocol. As the examples describe a MESI protocol with non-silent clean line evictions, we had to modify our baseline MSI protocol correspondingly.

The extended protocol contained 5 new states (4 for the cache, 1 for the directory), and 7 new message types. In the first iteration, we added 19 snippets to specify transitions involving the E state and non-silent clean line evictions. As in Case Study A, the description from the text under-specified corner cases and scenarios involving transient states, which manifested as errors reported by the model checker. The designer was able to obtain a fully verified protocol by adding 12 additional snippets over 8 iterations. (Please see Table 2 for more metrics.)

5.3 Discussion

Ease of use. We note that the manual effort required in our case studies was modest - 6 hours for an inexperienced designer to produce a fully verified MSI protocol, and less than 13 for a fully verified MESI protocol. On an average, we

needed less than 2 snippets to specify a transition, and the designers converged to a correct protocol in a reasonably small number of iterations. In both protocols, the designers had to visually inspect only a handful of counterexample traces to identify positive examples to refine the protocol specification. As TRANSIT searches for the smallest compatible expression, the designers found cases where the obtained expressions differed (were smaller in size) from expected results. In fine-tuning TRANSIT, we found that identifying the right expression grammar involves a trade-off between synthesis time and the verbosity of expressions generated. For instance, adding the constant symbol denoting an empty set can improve readability of expressions, but can impact the number of expressions enumerated (and hence synthesis time) adversely. The expression grammar that we used is presented in the appendix.

Ease of debugging. After synthesizing all guards for a control state, TRANSIT generates an *error guard* that accounts for any condition has not yet been covered. Consequently, any unexpected input events are detected immediately upon occurrence. Adding snippets to remove such errors was a straightforward, and often mechanical process. With the help of a trace-visualization tool that we developed, we were able to graphically explore error scenarios, and add positive examples to refine the protocol specification. Thus, fixes did not involve changing portions of the code, but instead required adding more snippets. In one case, each added snippet resulted in a number of disjoint antecedents. Such a scenario presents a challenge for guard synthesis as the generalization is an unwieldy disjunction. However, we observed that each term in the disjunction corresponded to a distinct message type, and by simply splitting a single guard-action block into multiple blocks grouped by message type, we were able to synthesize the guard. In fact, this suggests that for protocols, message type could be made a language construct when specifying snippets.

6 Conclusions

In this paper, we have proposed a new approach for designing communication protocols by adopting verification tools that interact with the protocol designer. Our approach builds on the intuition that protocols can be naturally expressed in terms of their behavior in example scenarios. We introduced the concept of *concolic* snippets to allow a designer to specify these behaviors as a mix of concrete examples and symbolic partial transitions. To demonstrate the feasibility of our approach, we developed a prototype tool based on expression generation, which generates complete protocol specifications from concolic snippets, that are then verified using a model checker. Our preliminary case studies using this tool allowed inexperienced designers to correctly synthesize representative cache coherence protocols of modest complexity with several hours of human effort. Encouraged by the initial feasibility results, our next steps are to further extend designing with snippets by exploring alternate computation strategies for expression synthesis, synthesizing EFSM descriptions from distributed scenarios such as message sequence charts, and techniques to automatically analyze counterexamples returned by the model checker.

References

[1] Abts, D., Lilja, D.J., Scott, S.: So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In: IPDPS (Apr 2004)
[2] Barrett, C., Tinelli, C.: CVC3. In: CAV (Aug 2007)
[3] Church, A.: Logic, Arithmetics, and Automata. In: Proceedings of the International Congress of Mathematicians (1963)
[4] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
[5] Dill, D., Drexler, A., Hu, A., Yang, C.: Protocol Verification as a Hardware Design Aid. In: ICCD (Oct 1992)
[6] Dill, D.L.: The Murphi Verification System. In: CAV (Aug 1996)
[7] E.Clarke, O.Grumberg, H.Hiraishi, S.Jha, D.Long, K.McMillan, L.Ness: Verification of the Futurebus+ Cache Coherence Protocol. Formal Methods in System Design 6, 217–232 (1995)
[8] Gulwani, S.: Automating String Processing in Spreadsheets using Input-output Examples. In: POPL (Jan 2011)
[9] Harel, D.: Can Programming be Liberated, Period? IEEE Computer (2008)
[10] Harel, D., Marelly, R.: Come, Let's Play: Scenario-based Programming using LSCs and the Play-engine. Springer (2003)
[11] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
[12] Komuravelli, R.: Verification and Performance of the DeNovo Cache Coherence Protocol. Master's thesis, University of Illinois (2010)
[13] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional Synthesis for Linear Arithmetic and Sets. Software Tools for Technology Transfer (STTT) (2012)
[14] Martin, M.M.K., Hill, M.D., Wood, D.A.: Token Coherence: Decoupling Performance and Correctness. In: ISCA. pp. 182–193 (Jun 2003)
[15] Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News (2005)
[16] Raghavan, A., Blundell, C., Martin, M.M.K.: Token Tenure: PATCHing Token Counting Using Directory-Based Cache Coherence. In: MICRO (Nov 2008)
[17] R.Bloem, S.Galler, B.Jobstmann, N.Piterman, A.Pnueli, M.Weiglhofer: Interactive Presentation: Automatic Hardware Synthesis from Specifications: A Case Study. In: DATE (Mar 2007)
[18] Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: FSE (2005)
[19] Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.A.: Sketching Stencils. In: PLDI (Jun 2007)
[20] Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching Concurrent Datastructures. In: PLDI (Jun 2008)
[21] Solar-Lezama, A., Rabbah, R., Bodik, R., Ebcioglu, K.: Programming by Sketching for Bitstreaming Programs. In: PLDI (Jun 2005)
[22] Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. Morgan Claypool (2011)
[23] Sorin, D.J., Plakal, M., Hill, M.D., Condon, A.E., Martin, M.M.K., Wood, D.A.: Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. IEEE TPDS 13(6), 556–578 (Jun 2002)
[24] Thomas, W.: On the Synthesis of Strategies in Infinite Games pp. 1–13 (1995)
[25] Vantrease, D., Lipasti, M., Binkert, N.: Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In: HPCA (Feb 2011)

Appendix

For synthesizing protocol completions with TRANSIT, we used the list of function symbols $\mathcal{F}$ shown in Table 3. The functions were chosen as a trade-off between expressiveness and the time required for expression synthesis. Functions with arity 0 represented constants (*e.g.*, the 0-ary function *Zero*).

Table 3. Expression Grammar used for Cache Coherence Protocol Synthesis

Function	Description
Plus (`Int, Int`) → `Int`	Integer Addition
Minus (`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`	Compute the cardinality of a Set
SetUnion (`Set, Set`) → `Set`	Set Union
SetInter (`Set, Set`) → `Set`	Set Intersection
SetDiff (`Set, Set`) → `Set`	Set Difference
SetOf (`PID`) → `Set`	Create a singleton Set from a PID
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
Equals (`T, T`) → `Bool`, $\forall$ `T` $\in \mathcal{T}$	Equality Test
GE (`Int, Int`) → `Bool`	Test if an Integer is greater than or equal to another
GT (`Int, Int`) → `Bool`	Test if an Integer is strictly greater than another
GetLeast (`Set, Int`) → `PID`	Get the smallest element from a Set
IfThenElse (`Bool, T, T`) → `T`, $\forall$ `T` $\in \mathcal{T}$	Conditional Expression
NumCaches (`Void`) → `Int`	Constant denoting the number of caches in the system
Zero (`Void`) → `Int`	Constant value of 0.