Transaction Flows and Executable Models: Formalization and Analysis of Message-passing Protocols

Muralidhar Talupur
murali.talupur@gmail.com

Sandip Ray
Strategic CAD Labs
Intel Corporation
sandip.ray@intel.com

John Erickson
MIC Methods, Tools, and Verification,
Intel Corporation
john.erickson@intel.com

***Abstract*—The lack of appropriate models is often the biggest hurdle in applying formal methods in the industry. Creating executable models of industrial designs is a challenging task, one that we believe has not been sufficiently addressed by existing research. We address this problem for distributed message passing protocols by showing how to synthesize executable models of such protocols from transaction message flows, which are readily available in architecture descriptions. We present industrial case studies showing that this approach to creating formal models is effective in practice. We also show that going the other way, *i.e.*, extracting flows from executable models, is at least as hard as the model-checking problem. These results indicate that transaction flows may provide a superior approach to capture design intent than executable models.**

I. Introduction

Over the last decade, significant advance has been made in the formal verification of industrial-scale designs. Formal tools scale to hardware designs with millions of gates and software systems with millions of lines of code [1], [2], [3]. However routine applications of formal methods have been confined to certain niche areas, *e.g.*, floating-point units, device drivers, etc. A key factor constraining the use of formal methods is the unavailability of appropriate models [4]. Constructing formally analyzable models of industrial designs is a complex enterprise, requiring significant expertise both in the artifact being modeled and in the formalism used. The model must be small and abstract to be tractable, while preserving the behaviors of interest from the original design for the analysis to be meaningful. Furthermore, there is a significant cost to maintaining models to keep up with the design evolution. Not surprisingly, most successful adoptions of formal methods have been in areas where the target was the implementation itself (*e.g.*, Register-Transfer Level (RTL) designs in hardware, or C/C++ implementations of software). Unfortunately, this means that verification occurs late in the design life-cycle, after these artifacts have been implemented. Moreover, for such low-level implementations, formal tools do not scale to complete designs [5]. The situation is exacerbated with shrinking time-to-market schedules, that make it infeasible to fix late-discovered deep errors which warrant significant design change. The result has been complex patches, point-fixes, and systems shipped with errors and vulnerabilities. To address these issues it is critical to facilitate easy creation of maintainable, high-level models early in the design life-cycle; the models can then serve as (1) targets for early analysis for catching architecture-level bugs, and (2) specifications driving later phases of development.

In this paper, we address the problem of efficiently developing high-level executable models for asynchronous message-passing protocols. Such protocols include cache coherence, resource allocation, bus locking, etc., and form the bedrock of modern multicore and multiprocessor systems as well as SoC designs. Errors in these protocols tend to be particularly difficult to detect since they involve unanticipated, subtle interleavings of concurrent communications that are difficult to exercise through simulation. Furthermore, protocol errors discovered late are difficult to fix, since they typically involve design changes in a number of participating components.

Our approach is based on automatic generation of executable models of the protocols from artifacts already created by architects during the system design process. These artifacts typically take the form of diagrams specifying different message transactions. Fig. 1 shows two such diagrams for a toy cache coherence protocol. Observe that they provide a "transaction-centric view" of the protocol: executions are broken into transaction scenarios, and a diagram specifies the message communications for each transaction. A traditional distributed system model, on the other hand, provides an "agent-centric view": for each participating agent *agt*, it specifies the behavior of *agt* under all possible system scenarios. We will refer to these models as *executable models*. They have a closer correspondence to downstream implementations (*e.g.*, RTL or software) which are also developed on agent-by-agent basis. Traditionally executable models are manually created by formal methods experts after studying architecture documents.

The main insight for our work is that, with a little additional information, the seemingly informal transaction descriptions created by architects can be used to synthesize executable models. The paper makes four contributions. First, we develop a formal foundation for specifying protocol transactions. Two key ingredients of this foundation are (i) *transaction message flows* (or simply, *flows*); and (ii) a definition of *compliance* that formalizes correspondence between transaction-centric and an agent-centric views. Second, we develop a framework for synthesizing executable models from flows. This makes it feasible for architects, having little familiarity with formal

This work was done when the first author was at Intel Corporation.

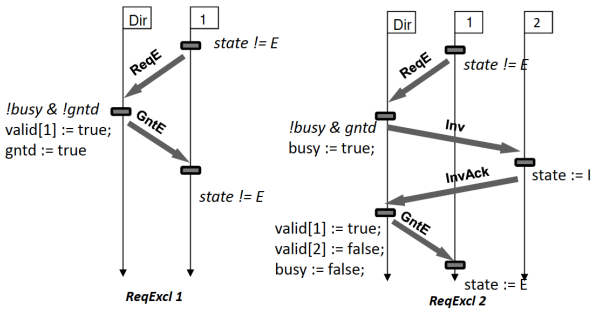

Fig. 1: Two architecture diagrams showing ReqExcl (request for exclusive line access) transaction for a toy cache coherence protocol. Agents are the clients (numbered as 1 and 2) and a memory controller or *Directory*. (a) The scenario in which no other agent has a copy of the cache line. So the line is granted immediately. (b) The scenario where another agent has a copy of the cache line. This copy must be invalidated (and its data written back) before the same line can be granted to 1.

methods, to develop, and analyze executable models of complex industrial protocols, *e.g.*, our tool has been used by architects at Intel to synthesize highly complex protocols in Intel's many-integrated-core (MIC) processors. Third, we report industrial case studies showing that protocol specifications via flows is effective in practice: model generation could be accomplished at *a fraction of the time* required by a formal methods expert to manually build a model. Finally, we show that while generation of models from flows is relatively simple, identifying flows from models is at least as hard as model-checking. This indicates that flows provide a strictly richer semantic information than an executable model.

The rest of the paper is organized as follows. Section II defines flows and formalizes the notion of compliance between flows and models. In Sections III and IV we discuss our approach to synthesize models from flows. In Section V we discuss the converse problem, *viz.*, extracting flows from models, and show that extracting (an appropriate notion of) compliant flows from a model $\mathcal{M}$ is as hard as model-checking $\mathcal{M}$. In Section VI we discuss application of automated synthesis of flows to models in practical case studies. We discuss related work in Section VII and conclude in Section VIII.

II. Systems, Transactions, and Executable Models

A. Executable Model

Our formalization of an executable model is based on guarded transitions. Such formalisms are well-known in concurrency literature [6], [7], and form the basis of system modeling in model-checking tools like Murphi [8] and SPIN [9].

A distributed system involves coordinated computation by a collection of *agents* with indices or *ids*. In our formalism ids are numbers $\{1, \ldots, n\}$. The system state is given by the local state of each agent and the states of communication channels. The local state of agent i is specified by the value of a finite collection *vars*(i) of *state variables*. For each pair of agents i, j, where $i \neq j$ *chans*(i, j) is a finite set of *channels* from i to j. To send a message m to j, agent i places it in some $c \in$ *chans*(i, j). All variables (both state variables and channels) are assumed to take values from a fixed, bounded, finite set $\mathcal{S}$. We assume that $\mathcal{S}$ includes a special "empty" value $\perp$ to represent an empty channel. For each agent i, denote the set *vars*$(i) \cup (\bigcup_{j \neq i}$ *chans*$(i, j))$ by Π_i. An *assignment* of a variable $v \in \Pi_i$ is given by $v :=$ *exp* where *exp* is an expression over $\Pi_i \cup \mathcal{S}$. Unless otherwise noted we keep the syntax of operations involved in expression exp unspecified, but assume that any expression in this paper can be evaluated over $\mathcal{S}$. A guard g_i for agent i is a Boolean expression over $\Pi_i \cup \mathcal{S}$.

Definition 1 (Rule): A *rule* for agent i is a construct of the form $r : g_i \rightarrow a_i$ where r is a symbol called *rule name*, (1) $g_i$is a guard for i and (2) a_i is a collection of assignments of some variables $v \in \Pi_i$.

Definition 2 (Executable Model): An *executable system model* (or simply, *model*) M of n agents is a pair $M \triangleq \langle R, I \rangle$ where R is a set of rules with unique rule names and I is an initial set of assignments to all variables in Π_i, for $i \in \{1, \ldots, n\}$ to constants in $\mathcal{S}$.

We assume that each $c \in$ *chans*(i, j) is assigned to $\perp$ in I. A *system state* s is an assignment to all the variables in Π_i, for all $i \in \{1, \ldots, n\}$. Given a system state s and a rule $r : g_i \rightarrow a_i$ for agent i, we say that r is *enabled* at s if and only if g evaluates to true in state s.

Definition 3 (Rule Firing): Given a rule $r : g_i \rightarrow a_i$ for agent i, we say that s' *is the result of firing of* r *from* s if the following two conditions hold

- r is enabled in s; and
- s' is derived from s as follows: If there is no assignment to v in r then v is assigned the same value in s' as in s. Otherwise, if there is an assignment $v := exp$ in r then v is assigned the value obtained by evaluating exp in s.

Definition 4 (Execution Trace): A sequence of rules $\tau \triangleq [r_1, \ldots, r_k]$ is called an *execution trace* of model $M \triangleq \langle R, I \rangle$ (where $r_i \in R for i = 1 \ldots k$) if there exists a sequence $[s_0, \ldots, s_k]$ of system states with $I = s_0$ such that following two conditions hold.

- r_i is enabled in s_{i-1}; and
- s_i is the result of firing r_i in s_{i-1}.

B. Flows and Flow Model

The notion of message flows is similar to Message Sequence Charts (MSCs) [10] used in the specification of multi-agent transaction systems. In particular, a flow is a partially ordered set of *events*, specifying the transactions of a protocol. An *event* is a 5-tuple $\langle$AGT, GD, RECV, SEND, UP$\rangle$ as described below. Fig. 2 shows the formalization of the *ReqExcl1* flow in Figure 1.

- AGT $\in \{1, \ldots, n\}$ specifies the index (or *id*) of the agent executing the event.
- GD is guard for $\Pi_{\text{AGT}} \cup \mathcal{S}$ and UP is set of assignments to variables in Π_{AGT}.

1)	$\langle 1, \texttt{true}, -, [\langle \text{ReqE}, \mathsf{Dir}\rangle], -\rangle$
2)	$\langle \mathsf{Dir}, \neg \text{busy} \wedge \neg \text{gntd}, [\langle \text{ReqE}, 1\rangle], [\langle \text{GntE}, 1\rangle], [\text{valid}[1] := \texttt{true}; \text{gntd} := \texttt{true}]\rangle$
3)	$\langle 1, \texttt{true}, [\langle \text{GntE}, \mathsf{Dir}\rangle], -, [\text{state} := \text{E}]\rangle$

Fig. 2: Formalization of three events in *ReqExcl1* flow of Fig. 1. Although process ids are restricted to be numbers in the formalization, we use the symbol Dir for the directory process for pedagogical reasons.

- SEND and RECV are lists of messages. Each message is a tuple $\langle \text{MSG}, \text{ID}\rangle$ where MSG is the actual message and ID is the index of the receiving agent (in case of SEND) or the sending agent (in case of RECV).

We use $e.\text{GD}$, $e.\text{AGT}$, etc. to denote the individual components of event e. As with rules, meaning is assigned to events via guarded commands: e is enabled whenever $e.\text{GD}$ holds and the list of messages specified in $e.\text{RECV}$ are available in the incoming channels of $e.\text{AGT}$; the execution causes the updates to the local state of $e.\text{AGT}$ as specified by $e.\text{UP}$ and the list of messages in $e.\text{SEND}$ to be sent through the outgoing channels of $e.\text{AGT}$.

Based on the above semantics, we impose following syntactic requirements and restrictions on events: (1) $e.\text{GD}$ is a Boolean expression over the state variables of $e.\text{AGT}$; (2) $e.\text{SEND}$ provides a list of assignments to the outgoing channels of $e.\text{AGT}$ where the right hand side of each assignment is a tuple $\langle \text{MSG}, \text{ID}\rangle$ specifying the message and receiver id; (3) $e.\text{RECV}$ is similarly a list of tuples $\langle \text{MSG}, \text{ID}\rangle$ containing the message and sender id.

Definition 5 (Transaction Flows and Flow Model): A *transaction Message Flow* (or simply, *Flow*) is a pair $\langle f, \prec\rangle$, where f is a set of *events* and $\prec$ is a partial order relation over f.

Informally, $\prec$ specifies the temporal ordering on the events in a flow. For Fig. 1, we can view each diagram as a linearly ordered sequence of events; the two diagrams represent two flows describing the two different ways in which a request for exclusive access can be handled. Note that for this and many common cases, the events in a flow f form a sequence, *i.e.*, the partial order $\prec$ is in fact a total order. Nevertheless, there are situations where the generality of partial order is necessary, *e.g.*in the "diamond transaction" shown in Fig. 3. For the rest of the paper, we use f instead of $\langle f, \prec\rangle$ to refer to a flow when the relation $\prec$ is clear from context.

1) Mapping between rules and events: There is an direct connection between rules and events. For the purpose of formalization, we assume fixed syntactic mappings $rl2ev$ and $ev2rl$ that translate a rule into an event and vice versa. The mapping $ev2rl$ maps an event e to a rule of the following form. Here the right hand side of the rule, specified as a list of items enclosed by $\langle\rangle$ represents a sequence of assignments to local and channel variables.

$$e.\text{GD} \wedge (\mathit{chans}[(e.\text{RECV}).\text{ID}, e.\text{AGT}] = (e.\text{RECV}).\text{MSG}) \rightarrow \langle e.\text{UP}; (\mathit{chans}[e.\text{AGT}, (e.\text{SEND}).\text{ID}] := (e.\text{SEND}).\text{MSG}); \mathit{chans}[e.\text{AGT}, (e.\text{RECV}).\text{ID}] := \bot\rangle$$

Correspondingly, the mapping $rl2ev$ takes a rule rl and extracts the five fields above to get an event ev. The only possible source of ambiguity in a rule rl is about the identity of the sender of messages in case rl involves multiple agents. In our model, messages are communicated using fixed unidirectional channels, and it is simple to identify the sender.[1] This agent can be viewed as the executing agent of the event.

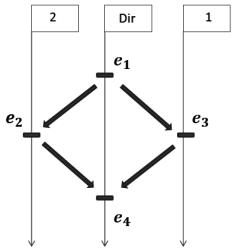

Fig. 3: A transaction requiring $\prec$ to be a partial order. The execution of e_1 enables both e_2 and e_3, and both these events must be executed before e_4 can be enabled.

2) Flows as templates: Given the correspondence between events and rules, a flow provides a template or a pattern for system execution, grouping together related rules with a temporal ordering on their firing. A flow can be invoked or *instantiated* several times, even concurrently, during a run of the system. To make precise the relation of an execution trace with flows, we need to disambiguate between these instances. The notion of tagging accomplishes that by augmenting a flow with a "tag". Here we assume that we have an unbounded set T of *tags* (which is different from all the previously defined sets, *viz.*, variables, values, events, rules, etc.).

Definition 6 (Tagged Events and Flows): A *tagged event* is a pair $[e, t]$ where e is an event and t is a tag. If $\langle f, \prec\rangle$ is a flow, then a *tagged flow* $\langle [f, t], \prec\rangle$ is obtained by replacing each event $e \in f$ with the corresponding tagged event $[e, t]$.

Definition 7 (Legal Tagging): Given a set of flows F and a set T of tags, a set $[F, T] \subseteq \{[f, t] : f \in F, t \in T\}$ is a *legal tagging* if and only if for $f, f' \in F$ such that $f \neq f'$, if $[f, t]$ and $[f', t']$ are members of $[F, T]$ then $t \neq t'$.

Informally, we want a *unique* tag to be associated with each instance of a flow in an execution trace of the system. The definition of compliance makes this notion explicit.

Definition 8 (Precedence-Preserving Mapping): Let $\tau \triangleq [r_1, \ldots, r_k]$ be an execution trace of model M, F be a set of flows, and $[F, T]$ be a legal tagging of F. Let $rl2ev_{\#}$ be a one-to-one mapping from rules in τ to tagged events in $[F, T]$. We say $rl2ev_{\#}$ is *precedence preserving* if for each r_i, $i = 1 \ldots k$

[1] In practice, the message names usually gives away the sender identity.

there exists a tagged flow $[f,t] \in [F,T]$ and a tagged event $[e,t] \in [f,t]$ such that the following conditions hold:

- $rl2ev_{\#}(r_i) = [e,t]$
- $rl2ev(r_i) = e$
- for each $p \in f$ such that $p \prec e$, there exists $k < i$ such that $rl2ev_{\#}(r_k) = [p,t]$.

Definition 9 (Compliance): Let $\tau \triangleq [r_1, \ldots, r_k]$ be an execution trace, and let F be a set of flows. We say that τ is compliant with F if there exists a precedence preserving mapping $rl2ev_{\#}$ from members of τ to events in a legal tagging $[F,T]$. A model M is compliant with flows F if every trace of M is compliant with with F.

Flows provide a generalization of control flow graph to a distributed setting. The definition of compliance essentially stipulates that the trace τ can be viewed as a composition of a collection of flow instances. The requirement of precedence preserving mapping guarantees that each such instance can be uniquely identified with a tag and respects the precedence constraints imposed by flows.

We end this section by briefly comparing the notion of flows introduced here with a related notion in previous work [11], [12] which was also called "flows". Flows in previous papers did not consider state annotations and updates. The more refined notion used here is necessary for synthesizing protocols, while previous work only used flows to infer invariants. Nevertheless, we use the same name in this paper for two reasons. First, the definition here strictly supersedes the previous notion, *viz.*, the previous usages can be accomplished with the current notion. Second, the notions are similar, both in structure and in "spirit", *e.g.*, both aim to exploit the transaction-centric view of protocols.

III. Synthesizing Models from Flows

Fig. 4 shows the high-level steps for applying our framework for synthesizing executable models. The user starts from an initial (possibly incomplete) set of flows F that captures the algorithmic aspects of protocol being designed, and progressively refines this set in response to feedback from a model-checker. In more detail, we automatically synthesize a (compliant) model M from F, and model-check M against a set of user-provided assertions I and a collection of sanity conditions. If model-checking detects deadlock or invariant violation, or a sanity condition fails, a counterexample is provided together with diagnostic information (cf. Section VI). The user modifies F, possibly by adding more flows or adjusting some existing one, so that the erroneous behavior is ruled out. The process is iterated until model-checking succeeds. Note that all the steps are automatic other than the obviously creative step of "fixing" F to rule out the counterexample.

The approach requires a set of assertions. The assertions we use are simple and straightforward, *e.g.*, for cache coherence protocols the obvious assertion is the coherence property itself. Sanity checks are also included to ensure that the generated model M (and, by implication, F) exhibit certain desired behaviors. We discuss some generic sanity checks in the next section. In addition, domain-specific sanity conditions can be added by the user. For instance, a simple sanity check for cache coherence protocols is that for every agent a there is a trace where a can get into *shared* and *exclusive* states.

```
while !(done) do
    Synthesize model M compliant with set of flows F.
    if M does not deadlock, M ⊨ I and passes sanity
    checks then
        M is the required executable formal model.
        done := true.
    else
        Modify F. In case of deadlock or invariant
        violation use the counterexample trace as a
        guide.
    end
end
```

Fig. 4: Synthesis-Refinement Loop for Interacting with Flows

A. Synthesizing M

Consider synthesizing a model from a set of flows F. A first naïve approach may be simply mapping each event e in each flow $f \in F$ to $ev2rl(e)$. To see why this does not work, note that $ev2rl$ is a function of e alone and not the preceding events of e in f. Hence we are not guaranteed that rule $ev2rl(e)$ is enabled respecting the ordering relation "$\prec$". The key task in synthesizing a model M is to create rule guards in M such that for any execution trace τ each rule $r \in \tau$ respects "$\prec$". To achieve this, we augment the state variables for each agent a by the following additional components:

- A "local tag list" T_a for each agent a, that disambiguates concurrently executing flows (including different instances of the same flow) involving a.
- A mapping $\eta_a : T_a \rightarrow Id \times \cup_{b \neq a} T_b$ that associates every local tag of a with a set (possibly empty) of local tags of other agents b that a communicates with.
- A set of "history variables" $H_a[f,t] \subseteq E_f$, where E_f is the set of events in flow f, for each agent a, flow f, and each possible tag value t. The history variable $H_a[f,t]$ records the firing of events in the instance of flow f with tag t. For example, if event $e \in H_a[f,t]$ then it means event e occurring in f has been fired by agent a with associated local tag t.

The local tags are different from the global tag introduced in Section II. Since each agent has only local visibility, it is impossible to ensure that the tags chosen for concurrently executing flows are globally unique. The crux of the synthesis consists of "book-keeping" to disambiguate between different active transactions based on local history and tags.

Suppose that f is a flow which includes an event e. Then we generate a rule $ev2rl^*(e,f)$ by extending the rule $ev2rl(e)$ with (1) additional assignments updating the relevant tag and history variables and (2) additional conjuncts on the guard. Let a be the agent executing e, that is, $e.\text{AGT} = a$.

Updates to History and Tag. Fig. 5 shows how these variables are updated. Note that we must update history variables of a each time a rule $ev2rl(e)$ fires.

if *e is the first event for agent a in flow f* **then**
  Pick a new tag t and add it to T_a.
  Add e to history variable $H_a(f,t)$.
  Let $m = e.\text{SEND}$. Replace m with a new message m' by augmenting it with the identifier (a,t,f).
  if *there is an incoming message msg for a in e* **then**
    Let (b,t',f) be the identifier associated with msg. Add the mapping $t \rightarrow (b,t')$ to η_a. This indicates that agent b uses local tag t' for that particular instance of flow f.
  end
else
  Then there exists $e' \prec_f e$ such that $e'.\text{AGT} = a$. The local tag t already picked for e' is used as local tag for e as well. Update $H_a(f,t)$, η_a and message $m = e.\text{SEND}$ exactly as above.
end

Fig. 5: Updating History and η Variables when agent a fires event e

Additional Guard. The additional guard conjunct is given by the expression $g_a[e,f] \triangleq \exists t : \forall e' : e' \prec e \Rightarrow e' \in H_a[f,t]$. That is, $g_a[e,f]$ is true only when all events preceding e' in flow f have been executed. Note that although we wrote the guard as a quantified first-order expression, for any system involving a finite set of flows and active tags, it can be written as a Boolean expression by enumeration.

Fig. 5 defines an algorithm for each agent to keep track of the local tag it uses to communicate with other agents in specific flow instances. This information is sufficient to create a global tag as required by compliance definition (cf. Theorem 1). Indeed, the elaborate tagging and history updates mirror typical RTL implementation of protocols with multiple, concurrently executing flow instances. Although essential for correctness, this part of protocol design is typically boring to humans while still being tricky to get right. Indeed, many errors in protocols arise by incorrect handling of such disambiguation procedures. By synthesizing it automatically, we free the human to focus on the algorithmically interesting parts.

The history and tag variables can make the synthesized model unbounded, *e.g.*, we need a history variable $H_a[f,t]$ for each possible tag value t. In practice, we impose finiteness by setting an upper bound to the set of possible tag values. This is reasonable in our case since our protocols are implemented in hardware with finite resources; consequently, most protocol definitions already include an upper bound on the possible tag values to impose implementability. Nevertheless, the model is more restrictive than flow descriptions. In particular, the set of possible tags constrains the number of possible concurrent instantiations of a flow. We can avoid this constraint to some extent by reusing tags of completed flows.

The correctness of the procedure is given by Theorem 1. Here $ev2rl^*$ and $rl2ev^*$ are augmentations of $ev2rl$ and $rl2ev$ respectively so that the domain of $ev2rl^*$ (and range of $rl2ev^*$) include additional guards and updates to the history variables.

Lemma 1: Let $\tau \triangleq [r_1, \ldots, r_i]$ be any execution trace of the model M obtained by applying the above procedure to F. Then there exists $f \in F$ such that $rl2ev^*(r_i)$ is an event in f and for each $e \prec_f rl2ev^*(r_i)$ there exists $k < i$ such that $rl2ev^*(r_k) = e$.

Proof sketch: The proof follows from induction on the length of τ. In the induction step, we note that using the guard specified for *Additional Guard* and rules for history variable update, for each event e such that $e \prec rl2ev^*(r_i)$, $ev2rl^*(e,f)$ must be executed for r_i to be enabled. ■

Theorem 1: If M is an executable model synthesized from a set of flows F, then M is compliant with F.

Proof sketch: Let $\tau \triangleq [r_1, \ldots, r_k]$ be any execution trace of M. It is sufficient to show that there is a legal tagging $[F,T]$ and precedence preserving mapping $rl2ev_{\#}$ from rule firings in τ to $[F,T]$. Below we provide a construction of T. The result then follows from Lemma 1.

We construct $[F,T]$ inductively. Recall that each rule firing r in τ is associated with a unique $\langle \text{AGT}, \text{TAG}, \text{FLOW} \rangle$ triple. We have to map every tuple $\langle \text{AGT}, \text{TAG}, \text{FLOW} \rangle$ seen in the execution τ to a global tag so that all local tags used for the same instance of a flow f by different agents get mapped to the same global tag.

For the base step, the unique tuple of the first rule r_1 is mapped to global tag of 1. In the inductive step, suppose the tuples for rules $[r_1, ..., r_{i-1}]$ have been mapped to global tags $T_{i-1} \triangleq \{1...g\}$ such that T_{i-1} is a legal tagging. We then consider the following cases for r_i.

Case 1: There exists $e \prec rl2ev^*(r_i)$ such that both e and $rl2ev^*(r_i)$ are events in f and $e.\text{AGT} = (rl2ev^*(r_i)).\text{AGT}$. Then by Lemma 1 there exists a $k < i$ such that $ev2rl^*(e,f) = r_k$. By induction hypothesis, $ev2rl^*(e)$ is tagged by T_{i-1} and this global tag can be used for r_i.

Case 2: If there is no e satisfying *Case 1* then either $rl2ev^*(r_i)$ is an initial event of f or all preceding events of $rl2ev^*(r_i)$ are executed by an agent different from $(rl2ev^*(r_i)).\text{AGT}$. In the former case, we can augment T_{i-1} with any unused tag and map that to r_i. In the latter case, $(rl2ev^*(r_i)).\text{AGT}$ must have received a non-empty set P of tuples $\langle \text{AGT}, \text{TAG}, \text{FLOW} \rangle$ from preceding rule firings. If all these are mapped to the same global tag g we use that for the current rule as well. Otherwise, we pick an unused global tag g and map the tuple of the current rule r_i and all the tuples in P to g. ■

IV. Additional Checks to Ensure Realizability

The notion of compliance only ensures that each trace in the model M can be viewed as an interleaving of flow instances, but not that all flows in F can be exercised by some trace. Consider the trivial model $\mathcal{M}_{tr}$ in which the guard for each rule is the logical false. Since the only trace of $\mathcal{M}_{tr}$ is the empty trace, $\mathcal{M}_{tr}$ is compliant with F. To ensure that every flow is "necessary", we introduce the following additional sanity check.

Non-Triviality Check. We say that a flow $f \in F$ is *exercised* by M if there is a trace τ in M such that τ is *not complaint* with the set of flows $F \setminus \{f\}$. M is *non-trivial* with respect to F if every flow $f \in F$ is exercised by M.

The non-triviality check ensures that without f, there is no way to decompose τ into interleaving instances of flows

from F. In general, it can require exhaustive exploration of M. However, it is efficient in practice since model-checking quickly discovers traces exercising all flows (cf. Section VI-A).

In addition to non-triviality, we perform two other checks which catch frequently observed mistakes in flows. These checks are done on the flows F directly, not on the synthesized model M. First check is that for every message m sent by some event e in flow f, there is some other event e' in f, $e \prec e'$ such that e.RECV includes m. The second check, called *prefix consistency* is more subtle. Here we assume that for flow $\langle f, \prec \rangle$, the relation $\prec$ is a total order over the events $\{e^1_{(f,a)}, \ldots, e^2_{(f,a)}\}$ executed by agent a (so that they form a sequence). This is a general restriction and follows well-known tradition of distributed system definitions [7].

Prefix Consistency Check. Let f_1 and f_2 be two flows such that the event sequences executed by agent a are $[e^1_{(f_1,a)}, \ldots, e^k_{(f_1,a)}]$ and $[e^1_{(f_2,a)}, \ldots, e^l_{(f_2,a)}]$ respectively. Then f_1 and f_2 are *prefix consistent* if the following holds for $n = 2, \ldots, \min(k, l)$: If the first $(n-1)$ events of both sequences is identical, and the GD and RECV fields of $e^n_{(f_1,a)}$ and $e^n_{(f_2,a)}$ are identical, then so must be the UP and SEND fields.

The motivation for prefix consistency comes from the fact that when two or more flows share a prefix there may be "misunderstanding" among agents regarding which flow is being executed (usually leading to a deadlock). For instance, suppose $e^n_{(f_1,a)}$ and $e^n_{(f_2,a)}$ cause the same message m^b_a to be sent from agent a to b, but local updates on a and the expected response from b are different. Suppose a executes $e^n_{(f_1,a)}$. Receipt of message m^b_a can enable b's response to $e^n_{(f_2,a)}$ in addition to $e^n_{(f_1,a)}$. That is, from the perspective of b, the flow its participating in is f_2 whereas from the perspective of a it is f_1. Consequently, the response sent by b is discarded by a which continues to wait for a response to $e^n_{(f_1,a)}$, leading to a deadlock. Indeed, we introduced this check after observing that this phenomenon is quite common for many industrial protocols and leads to subtle errors.

Finally, note that the checks described in this section only ensure that flows pass a minimum quality screening; they cannot ensure that only correct models are synthesized. Indeed, the quality of synthesized model is only as good as the flows. In practice, particularly in the initial iterations of the refinement loop of Fig. 4, flows do contain errors that have to be fixed by the user through analysis of model-checking counterexamples. The main advantage of using flows over directly writing executable models from scratch is that it provides an easier and more intuitive way of creating models.

V. Extracting Flows from Models

In this section we consider the inverse problem of the preceding section, *i.e.*, extracting a set of compliant flows from a given executable model. In addition to being of theoretical interest, *e.g.*, for comparing the semantic richness of flows vis-a-vis models, an efficient flow extraction algorithm has practical utilities. For instance, flows can yield powerful invariants facilitating formal verification [11], [12]. Moreover, there are legacy models of industrial protocols which are large and hard to understand; extracting flows may facilitate understanding by exposing the underlying transaction structures.

Flow Extraction Problem. Let M be any executable model. Construct a finite set F of flows such that (1) M is compliant with F, and (2) each flow $f \in F$ is exercised by M.

Condition 2 ensures that for each f in F, an instantiation of f occurs in some trace τ of M. This rules out trivial cases, *e.g.*, defining each rule in M to be a flow with a single event and calling the rule set to be the set of "extracted" flows.

Unfortunately, the flow extraction problem as stated is as hard as model-checking M. To see that, let I be an arbitrary predicate over M, and consider the model M^+ obtained by extending M with the single rule $r : I \rightarrow$ NOP, where NOP does not involve any updates. Let F be a set of flows satisfying Conditions 1 and 2 above. Then $\neg I$ is an invariant of M if and only if no $f \in F$ includes the event $rl2ev(r)$. This follows by noting that r occurs in some trace if and only if I is true of some reachable state of M.

In spite of the result above, it is often possible to extract approximate flows from a sufficient set of bounded executions of M by heuristically inferring event precedence. Indeed, in many cases, we empirically found such an approach to produce flows similar to those created by architects. Thus, for legacy protocols with no available flows, such approximations can be used as substitutes to mine invariants. However, the result shows that extracting "perfect" flows is an intractable reverse-engineering problem.

VI. Applications

A. Flows2HLM Synthesizer

We developed a tool, *Flows2HLM*, to synthesize models from flows. The tool implements the framework in Section III, augmented with the following facets to facilitate adoption.

Parameterized Flows. The definition of flows required each event to include the id of the executing agent. But in practice, ids only specify the type of the executing agent (*e.g.*, *Directory* agent vs. *Cache* agent). Two or more agents of the same type occurring in the same flow are disambiguated by using numbers along with type. Our tool supports such parametric flows. The key change required in the algorithm is to add agent ids to the local tags to disambiguate same events from parametric flows executed by different concrete agents. Dealing with parametric flows keeps the synthesized model small and manageable.

Murphi Types. The model generated by Flows2HLM is a Murphi [8] model. Apart from adding tagging information and event precedence, it also adds a Murphi header file declaring the types of variables. Type inference in this case is simple, since all variables are finite enumerated types.

Flows2HLM follows the refinement loop of Section III, using Murphi as the model-checker. In addition to generic assertions (*e.g.*, cache coherence) and the sanity checks discussed in Section IV, the user can write project-specific sanity checks, constraints, etc. The generated model and any assertion violations are reported to the user, together with diagnostic

information culled from the counterexample. The crucial diagnostic information is the interaction of flows leading to the failure. Typos and "shallow" errors are typically identified (and easily repaired) in initial synthesis iterations. For example, an error in message type manifests in a deadlock, as follows. Suppose agent a is inadvertently declared to send b a message of type E instead of correct type C; then b, which expects C, waits indefinitely, leading to a deadlock. Finally, while there is no guarantee, the initial iterations coupled with random simulations are enough to see that all flows are being exercised; thus, the non-triviality check of Section IV is easily satisfied.

B. Synthesizing Industrial Protocols

Flows2HLM has been used to synthesize several cache coherence protocols for Intel's next-generation many-integrated-core (MIC) processors; recently, it has been applied to bus lock and credit management protocols as well (cf. Table I). For pedagogical reasons, we also synthesized two academic cache protocols, German [13] and Flash [14], and their flows are publicly available [15]. We invite the reader to explore them, to appreciate the intuitive nature of flow specifications.

We elaborate a bit on our experience with Intel 2, since it is the most complex protocol synthesized by Flows2HLM so far (and perhaps the most complex cache coherence protocol formally analyzed). Interestingly, the synthesis was done fully by an architect with no prior formal modeling experience. The initial definition took two days and constituted 40 flows. Several sanity checks were added by the architect, *e.g.*, that the cache for each agent can get to an exclusive state. This description contained many shallow errors which were detected by Flows2HLM, including the deadlock scenarios discussed above. Subsequently subtle bugs were exposed, including an unexpected response to a local snoop message which required adding a new flow to fix. Overall *six major bugs were found*, which required significant modification of flows; any *one* of them, if leaked into RTL, would have led to a costly RTL churn. The effort took a month, including modification to Flows2HLM; in contrast, an earlier effort developing a hand-written Murphi model of similar complexity by a formal verification expert had required 6 months.

In addition to facilitating use of formal models by architects, a major gain of our approach is easy maintenance and modification of protocols. For Intel 2, the architect subsequently made major changes to flows to create a derivative protocol in matter of days, something that would be very hard with hand-written executable models. Furthermore, the architect used counterexamples returned by model-checker during the synthesis effort to estimate downstream RTL validation complexity. If model-checking counterexamples involve complex interleaving of several flow instances with a number of agents, one may expect bugs in further elaborated RTL implementations to also exhibit similar characteristics and hence require significant validation effort. Based on this insight, the architect simplified the initial protocol to keep the RTL validation complexity manageable.

Finally, since annotations are written manually as part of flow description, our approach provides reduction in specification detail over hand-written executable models only if they are small. This has been the case for most message-passing protocols we have seen, including academic cache coherence protocols as well as SoC and MIC protocols in Intel. Annotations account for less than 10% of model size in all our examples, and less than 5% for the larger protocols. This is not due to bloating from auto-generated code; hand-written models of protocols of comparable complexity were typically of a similar size. Note that for *microarchitectural* protocols, which involve lower level details like message buffering and arbitration, annotations may form a larger part of the description. Investigating application of flows for such protocols is an interesting future work.

VII. Related Work

Several protocol description techniques have been created in recent years, *e.g.*, table-based methods, message sequence charts (MSC), etc. [16], [4], [17], [10]. However, they have not been widely adopted in industrial practice. We speculate that a key bottleneck is the need to think about protocols in terms not natural to the architects. For instance, table-based methods require manually projecting system-level transactions to each agent and carefully tracking the set of transactions that the agent can participate in concurrently. Furthermore, a local change in one transaction requires modification of multiple tables. The comparison with MSCs [10] is more interesting. MSCs capture a diverse range of distributed computing artifacts, including interface protocols and real-time systems; consequently, they include several bells and whistles. While graphical formats provide a more intuitive visualization than text for small protocols, they become unwieldy and inflexible for protocols with 40 flows. Rather than identifying a subset of MSCs for specifying protocols, we found it easier to develop a simple language analogous to MSCs directly based on artifacts we observed the architects to use in informal specifications.

There has been recent related work on synthesizing distributed protocols. Udupa *et al.* [18] synthesize models from concolic execution snippets via user-guided iterative refinement using counterexamples from a model checker. Concolic snippets are analogous to tabular specification, with each snippet corresponding to a row. Synthesis based on snippets cannot account for the context of an event execution, *i.e.*, preceding events in a flow. We found flows to provide a more effective and natural starting point. Alur *et al.* [19], use *scenarios* and temporal properties to synthesize protocols; they address the problem of automatically synthesizing a model even when the number of scenarios is inadequate. There is superficial correspondence between scenarios and flows. However, while flows are self-contained, redundancy-free descriptions of protocol transactions, scenarios are sample executions. The problem addressed by Alur *et al.* is to automatically synthesize a model when the number of scenarios is inadequate. In contrast, we take the set of flows F itself to be the description of system transactions; thus, assertion violation in our refinement loop identifies inconsistency between the flows in F but no automatic repair mechanism. The analogous repair problem in our setting, *e.g.*, repairing protocol model by supplying a completely missing flow, would be unsolvable: if a flow is missing it cannot in general be inferred from other flows. Finally, using temporal assertions to capture behaviors of complex distributed protocols is hard and requires significant expertise in formal logic, making it unusable for architects. In our experience, flow-based design capture is more closely aligned to industrial design development.

Protocol	Type	No. of Flows	Murphi Model LOC	State Annotation LOC
German	Cache Coherence	4	600	30
Flash	Cache Coherence	10	1400	100
Intel 1	Cache Coherence	36	5000	200
Intel 2	Cache Coherence	43	6500	200
Intel 3	Bus Lock	3	1600	120
Intel 4	Credit Management	3	200	20

TABLE I: Some protocols synthesized with our tool

VIII. Conclusion

We formalized the notion of transaction message flows, and provided a method to synthesize executable models. We also showed that flows contain strictly richer semantic information than models: while generating models from flows is easy, extracting flows from models is intractable. To our knowledge, ours is the only technique for automated protocol modeling that has found consistent use in industry. The cache coherence protocols analyzed via Flows2HLM are some of the most complex ones to ever undergo formal verification. Note from Table I that the industrial cache coherence protocols have about an order of magnitude more flows than German, Flash, etc. that constitute representative benchmarks for state-of-the-art automatic formal verification techniques.

The problem of synthesizing models arose out of the need to analyze these highly complex industrial protocols early in the design life-cycle. Previous work [11], [12] addressed this problem by mining invariants from transaction-level descriptions to facilitate formal verification of protocol models at this scale. Nevertheless, a limiting factor for its practical adoption was the complexity of creating these formal models. This bottleneck, together with the observation that flow diagrams in architecture documents often represent "authoritative" source of protocol descriptions for designers and validation engineers, motivated the approach presented here. The results of this paper, together with the previous results [11], [12], suggest that flows provide a powerful and efficient method for modeling, analysis, and understanding of protocols.

In future work, we plan to exploit flows further in protocol modeling and analysis. One application is generating test harness for RTL simulation, *e.g.*, flows can be used to encode environment behaviors when exercising the RTL implementation of an agent. Another future work is to use repair techniques to fix missing annotations in user-provided flows, perhaps through a user-guided refinement loop. Finally, tabular specifications, although difficult for architects, are useful for downstream RTL designers; it will be interesting to explore synthesis of tabular specifications from flows.

References

[1] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in intel coretm i7 processor execution engine validation," in *21st International Conference on Computer-Aided Verification*, 2009.

[2] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, 2001.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Communications of the ACM*, vol. 5, no. 2, 2010.

[4] D. James, T. Leonard, J. O'Leary, M. Talupur, and M. Tuttle, "Extracting Models from Design Documents with Mapster," in *Proceedings of the 27th Annual ACL Symposium on Principles of Distributed Computing (PODC 2008)*, R. A. Bazzi and B. Patt-Shamir, Eds., 2008, p. 456.

[5] S. German, "Tutorial on Verification of Distributed Cache Memory Protocol," in *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, A. J. Hu and A. K. Martin, Eds., 2004, http://www.cs.utah.edu/~ganesh/presentations/fmcad04_tutorial2/german/steven-tutorial.pdf.

[6] S. S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, 1976.

[7] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. 3, no. 2, pp. 125–143, 1977.

[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," in *IEEE International Conference on Computer Design*, 1992.

[9] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, 1997.

[10] ITU-TS Recommendation Z.120, "Message Sequence Chart (MSC) Annex B: Algebraic Semantics of Message Sequence Charts, ITU-TS, Geneva," 1995.

[11] M. Talupur and M. R. Tuttle, "Going with the Flow: Parameterized Protocol Verification using Message Flows," in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008)*, A. Cimatti and R. B. Jones, Eds., 2008.

[12] J. W. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol Verification using Flows: An Industrial Experience," in *Proceedings of the 9th Internation Conference on Formal Methods and Computer Aided Design (FMCAD 2009)*, A. Biere and C. Pixley, Eds., 2009, pp. 172–179.

[13] A. Pnueli, S. Ruah, and L. Zuck, "Automatic Deductive Verification with Invisible Invariants," in *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, T. Margaria and W. Yi, Eds., 2001.

[14] J. Kustin and et. al, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture (ICSA 1994)*, 1994.

[15] M. Talupur, S. Ray, and J. Erickson, "Flows and Murphi Models for German and Flash Protocols," 2014, See URL http://www.cs.cmu.edu/~tmurali/flow_examples.

[16] C. L. Heitmeyer, M. Archer, R. Bharadwaj, and R. D. Jeffords, "Tools for constructing requirements specifications: the SCR Toolset at the age of nine," *Computer Systems: Science & Engineering*, vol. 20, no. 1, 2005.

[17] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[18] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, "TRANSIT: Specifying Protocols with Concolic Snippets," in *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, 2013.

[19] R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing Finite-state Protocols from Scenarios and Requirements," 2014.