Flexible Runtime Verification Based On Logical Clock Constraints

Daian Yue, Vania Joloboff, Frédéric Mallet

Flexible Runtime Verification Based On Logical Clock Constraints

Daian Yue*†, Vania Joloboff*†, Frédéric Mallet‡*

*MOE Trustworthy Software International Joint Lab, East China Normal University, China, ENS Rennes, France

†Inria, France, e-mail: vania.joloboff@inria.fr

‡Université Cote d'Azur, CNRS, Inria, I3S, France.

Abstract

We present in this paper a method and tool for the verification of causal and temporal properties of embedded systems, by analyzing the trace streams resulting from virtual prototypes that combines simulated hardware and embedded software. The proposed method makes it possible to analyze different kinds of properties without rebuilding the simulation models. Logical clocks are used to identify relevant points to put observation probes and thus also reducing the trace streams size. We propose a property specification language, called PSML, and based on behavioral patterns that does not require knowledge of temporal logics. From a given PSML specification, simulation is instrumented to generate a trace and the code is dynamically loaded by the simulator. The resulting trace stream is analyzed by parallel automata generated from the specification. The experiments, developed over the SimSoC virtual prototyping framework, show flexibility, possibility of using multi-core platforms to parallelize simulation and verification, providing fast results.

Keywords – Runtime verification, trace analysis, property specification, clock constraint specification language, simulation, debugging, model driven engineering, virtual prototyping

I. Introduction

Virtual prototypes provide good solutions to model and verify systems combining hardware components with embedded software, hardware-in-the-loop, real-time operating systems, various devices and their drivers. Even though exhaustive verification techniques can address increasingly larger parts of the system, integration requires complementary techniques that consider the system as a whole. Virtual prototypes also provide a good framework to test alternative hardware architectures or develop the embedded software while the hardware architecture is not yet fully available.

We promote the use of a Model-based verification approach. The model provides a high-level abstraction to detect early problems, it contains a set of expected properties from the different components of the system. From this central model, several artifacts can be generated, including, in our case, verification modules to ensure that a component implementation satisfies the expected system properties. For known required properties, assertion-based verification techniques can be used [1], [2], typically by running many simulations with varying parameters and varying interactions.

However, most often during the design phase the engineers discover a system failure that is not captured by the assertions as it was not anticipated. Investigating the cause of the failure may not be easy because it is a long chain of causality events that eventually lead to failure. To investigate the failure cause, simulation trace files are generated that record a lot of information which can be exploited with trace analysis tools. A trace stream usually consists of the sequence of mixed events, states and transition steps with variable values, annotated with time stamps. The engineers eventually discover the problem by successive iterations, typically searching in the trace stream at which point it deviates from the model. They also often discover new properties that should be verified but were not formulated as initial requirements, which can be added as new assertions. Trace analysis is thus very useful as long as the trace analysis tool is convenient enough to be handled by seasoned engineers and powerful enough to discover intricate properties.

Two issues in trace analysis are that firstly it may be impractical or not feasible to re-compile and re-build the simulator to check new properties; secondly that trace files have become huge in industrial applications (hundreds of Gigabytes) as systems have become fairly complex, although it is often the case that a lot of the trace stream data is actually irrelevant to the failure cause.

The purpose of this work is to make it easy to check for properties by quickly analyzing trace streams with a powerful tool able to verify causal and temporal properties, also able to reduce the size of trace streams. We are proposing a new approach for trace analysis that does not require rebuilding the simulator when adding new properties. Raw binary data are abstracted into a formal trace that can be used either dynamically as a run time verification tool to detect property violations (without slowing down the simulation) or statically by filtering out unnecessary data with respect to the properties.

The paper is structured as follows. The next section presents background work in the trace analysis and property specification areas. Section III presents our global approach, decomposed into several steps, based on a modeling approach and model

Daian Yue's work has been funded by ECNU Fund for Short Term Overseas Academic Visit, ENS Rennes and Inria, e-mail: 51141500055@stu.ecnu.edu.cn

transformation using Domain Specific Languages (DSL). Section IV presents the details of the tools function and an example drawn from a System on Chip simulator.

II. Background and Related work

A. Related Work

Even though model-driven engineering approaches with model verification techniques are used, the engineers still face the problem of debugging simulation models [3] and runtime verification is still needed in most industrial applications [4]. Trace analysis has been a topic of work for some time in the simulation community. Several languages in the literature allow to express temporal properties, among which Linear Temporal Logics [5]. LTL abstracts the system behavior as an infinite sequence of states and establishes either properties of *invariance* or *eventuality*. All the properties are expressed relative to the steps of execution, using modal temporal operators, and not relative to an external (physical or not) clock measure. Several efforts have been made to provide real-time extensions for temporal logics (RT-LTL and others).

Later, some efforts were made to renounce to the full expressiveness of some logic in favor of a representation more natural to the designers and more computationally tractable. This lead to the notion of Logic Of Constraints (LOC) initially proposed by Balarin and al. [6] and used for trace analysis by Chen and al. [7]. This system allows to express functional and performance constraints containing event names, instances of events, index variables allowing to express generic formulas. For example, the formula $t(Stimuli[i+1]) - t(Stimuli[i]) < T$ denotes the property that two successive occurrences of the event $Stimuli$ must occur within a given time frame T.

As SystemC is widely used by the embedded systems community, the specific VCD (Value Change Dump) trace format has been defined for signal tracing and many commercial or non commercial tools exist for tracing wave forms. A more flexible trace generator is proposed by CULT [8], but there is no associated tool to analyze the traces produced. A SystemC analyzer like [9] can be used to verify temporal properties holding in the SystemC processes, but it requires access to the SystemC modules and cannot check information originating from embedded software running over the simulated hardware. Although the experiments related in this article are based on SystemC, nothing in the approach requires SystemC, and our DSL makes it possible to check value changes on any variable, not restricted to signals.

The COSITA tool [10] analyzes traces resulting from co-simulation between SystemC and Matlab models for automotive applications. The tool permits to notice differences between simulation and emulation to investigate the models but it does not include a property specification language. The Meta-Event Definition Language (MEDL) [11] is used to verify properties in traces of packets in network simulation, using the Monitoring and Checking (MAC) framework [12]. It is based on a logic for events and conditions, which allows to consider instances of events by means of counters, similar to LOC.

Regarding property specification, the seminal paper from Dwyer and al [13] has inspired many subsequent works. In particular the pattern specification language proposed by [14] can capture causality and conditional properties with timing conditions, but does not seem to capture counting occurrences.

A similar though different approach from trace analysis consists in considering usage scenarios and either generate timed automata that can be model-checked such as [15] or generate tests associated with verification of pre and post conditions [16]. The work from [17] has extended this mechanism to actually implement trace analysis to verify some temporal relations. Yet another approach is to replay execution of a trace over the model to investigate the example at stake [18].

Regarding the tooling aspect, the Open Trace Format [19] provides a flexible trace format that inspired our trace mapping work. Our mapping system is using the notion of meta model for defining a trace format introduced by [20]. Our specification language is based on the Clock Constraint Specification Language (CCSL) [21]. CCSL is uncomparable to LTL and CTL (see [22]), it provides a set of safety patterns regarding both causal and temporal properties. It combines purely boolean expressions with unbounded counters, thus being more expressive than regular languages and adequate for properties targeted here. Although our tool does not use the software library from TimeSquare [23], it relies on the notion of logical clocks. In the sequel we use the word *tick* to mean an occurrence of a logical clock.

B. The SimSoC Framework

The work described here is based and integrated into SimSoC [24], an open-source simulation framework developed at Inria [25]. The code generated by the tools described below is compiled and linked with the SimSoC Instruction Set Simulator to generate traces. Although our work is based on SimSoC, it is independent and can be re-targeted to any other simulator.

III. Runtime Verification

A. Global Architecture

The global software architecture of our approach is shown on Figure 1. It is decomposed into four independent steps. The first step is the trace generation, based on a trace meta model so that all traces are conforming to some well defined model. The second step is a model transformation process, using a DSL called STML, in order to map the actual simulation binary trace data into an abstract trace of logical clocks. The third step is the property specification by means of another DSL called

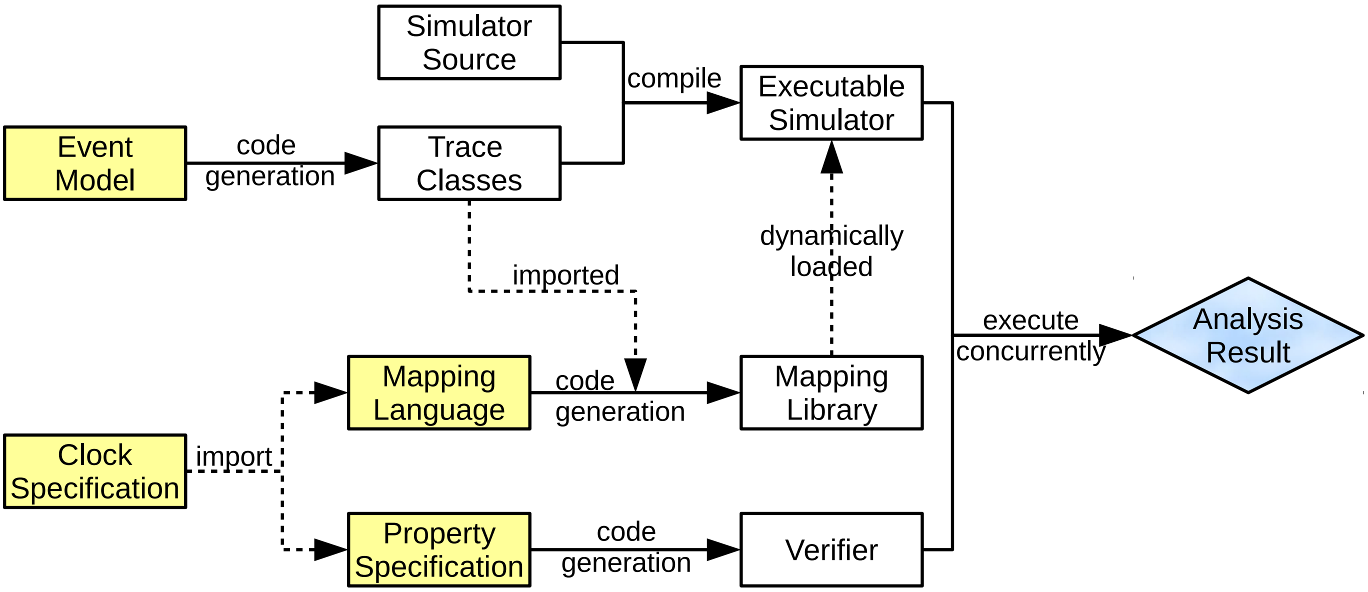

Fig. 1: Global architecture

TPSL, an English-like pattern-oriented specification language. The final step is the property verification over the mapped trace, using code generated by the property specification compiler.

The individual steps, described in more details in the following sections, are motivated and structured as follows:

- We initially place a light constraint on the simulator: The simulator can generate arbitrary binary data into what is called the *raw trace* as long as each data item dumped into the trace can be modeled as a structure that can be described itself using the meta-model that is defined and explained below in sectionIII-B. Because the trace corresponds to such a model, it can be parsed and understood for extraction and transformation. Another constraint on the simulator implementation is that it must comply with the trace interface that we provide to produce trace items.
- The runtime verification is based on the logical clock notion from CCSL. The verification process eventually operates on logical clocks. Thanks to the meta-model, a mapping process can be defined, which maps raw binary data into logical clock ticks. For example the fact that the variable value of a sensor has changed can be transformed into a specific clock tick.
- This mapping of raw data is achieved using a domain specific language (DSL) called STML (Simulation Trace Mapping Language), defined using the Eclipse Modeling Framework. The engineers can take advantage of STML to filter out useless data from the trace and significantly reduce the trace file size, or map complex data structures into clock ticks. The STML compiler generates the mapping code as a dynamically loadable module implementing the trace interface.
- After the raw trace has been mapped to a trace of logical clocks, it can be analyzed to verify the required properties, expressed in another DSL named TPSL (Trace Property Specification Language). TPSL has been designed to be a simple language usable by engineers that do not require knowledge of temporal logics or logical clock algebra. It has English keywords and simple common mathematical expressions. A TPSL module consists of an arbitrary number of properties. The TPSL compiler compiles each module into a set of parallel automata corresponding to the properties.

The resulting automata are applied to the mapped trace file and result into a PASS or FAIL status. In case of a FAIL an explanation is provided upon the failed expression in TPSL. Since the automata are parallel and no backtrack is ever operated, the verifier performance is linear with the trace file size.

A key notion in CCSL is the notion of simultaneity. In this framework, it is a parameter of the system. The simulator must provide a timestamp for all data item produced. These timestamps must be expressed in some unique real time clock, typically the hardware clock, the resolution of which is arbitrary. For verification of properties, a *trace clock* is used. The mapping transformer is parameterized by the trace clock frequency, which must be smaller than the real clock frequency. The trace emitter finally emits logical clock ticks annotated with tags related to the trace sampling frequency. Therefore, trace items that may have a different timestamp relative to the real time clock may translate into simultaneous clock ticks as observed by the trace clock, as they carry the same time stamp. The trace clock frequency should be chosen by the engineers depending on their use cases.

B. Trace Emission

To avoid recompiling and rebuilding the simulator when checking different properties, the trace generation is done inside a dynamically loaded module that is provided when the simulation starts as a parameter. The simulation must call this module using the provided interface. A key element of this interface is the `trace_emit()` function, an abstract polymorphic function that the simulation engineers must call from their model.

The interpretation of the raw trace data is defined by a Ecore/UML model. This model must comply with a meta-model that is essentially the following: each data item in the trace must carry a timestamp and belong to some class. Each class has a name and an arbitrary number of attributes. Each attribute has a name and a data type, which must be a basic value type, pointers and references are not supported. Embedded structures are not supported either at the moment, except fixed size arrays. These data items can represent anything, sensor values, states, transitions, events, transactions, no specific requirement is made.

A trace model following the meta model must be provided by the engineers constructing the simulator (using Eclipse Modeling Framework) who can output any piece of information described in the model. In our experiment, the simulator is the SimSoC simulator from Inria coded in SystemC, hence using C++ code. From this model, it is possible to derive trace generation code that captures and generates the simulation data into a well understood format, such as XML technology. For example, the trace model can describe a SystemC TLM transaction to have an initiator, a target, an operation, an address and argument data. The data types of those attributes are described using abstract type names. The code generator is parameterized to map these abstract types into the target programming language data types, also provided in the trace model.

A call to function `trace_emit(transaction)` generates a transaction data item in the trace whoseformat is the one derived from the trace model. As the trace generator is using code generated from the model, it can arrange the layout of the trace data as most convenient for later processing, while maintaining the integrity. For example the N^{th} bit of a register can be translated into a boolean flag if deemed appropriate. The role of function `trace_emit()` is to decide how the trace data should be emitted.

C. Trace Mapping and Filtering

The next step in the process consists in mapping the trace events to logical clocks such as defined by CCSL [21]. The STML was devised to define such a mapping. A STML module takes as input two parameters, on one hand the trace model described above, on the other hand a set of clock definitions. This clock specification actually comes from the TPSL language described below. The mapping module then contains instructions that specify the mapping of the trace data into either nothing (the data is eliminated from the trace) or into one or more clock ticks.

A STML module consists of a set of rules, each one introduced by the keyword `for`. There must be one and exactly one rule for each trace item class described in the trace model passed as the first parameter. Each rule describes how the trace items of this class translate (or not) into the logical clock occurrences of the second parameter, using a syntax close to the C language for expressions.

In the sequel of this paper we illustrate the approach with a virtual prototype example that consists of a simulator of a platform with a multi-core Power architecture processor. Each core has its own MMU (Memory Management Unit), and is connected to various peripherals, including a OpenPIC multi-core interrupt controller and a transmit controller with a FIFO queue. The hardware is controlled by a real time software that has drivers monitoring the peripherals. The hardware FIFO queue receives items asynchronously from the software but it re-sends these items synchronously to the outside world. The hardware should (i) emit a $FifoEmpty$ signal whenever the queue is empty (including at initialization phase), a $FifoFull$ signal when it is full, (ii) send the data items with limited jittering (iii) have a reasonable latency. On the other hand, the software should (i) enable interrupts and service them, and (ii) it should not put data into the queue when it is full.

In STML example on Fig. 2, only data items related to sending and receiving and related interrupts to the FIFO are mapped in order to analyze the FIFO handling behavior. All other trace data items are simply thrown away. Only the items related to the FIFO are mapped into clock ticks. Regarding interrupts, the trace items emitted by the interrupt controller translate into different clock ticks depending on the successive trace items as the interrupt controller is tracing both whether it receives a signal and whether the corresponding interrupt is enabled. In the latter case, the interrupt may stay pending (e.g. because there is a higher priority interrupt) or it is raised to some core. These different situations map to different clock ticks.

The STML language helps reducing the trace file size by either filtering out irrelevant data with regards to the properties to be verified or by combining complex binary data into a single clock tick. Another feature reducing the trace and facilitating property verification is the ability to emit clock ticks only when a value changes, eliminating the need to repeatedly store the same value. The STML compiler generates a dynamic linking library, loaded dynamically by the simulator. This dynamic library is called by `trace_emit()` interface and implements the specified mapping. Therefore one can dynamically generate as many traces as desired from the same simulator, without rebuilding the virtual prototype (which may not be an easy and quick task). Conversely, using static analysis, a huge file can be dumped using no filtering, but the library can be stored and reused to transform that huge file into as many clock mappings as desired.

```
for Transaction t { ignore }
for Signal s {
  (s.signal == FifoIsEmpty) ? emit FifoEmpty
  : (s.signal == FifoIsFull) ? emit FifoFull : ignore
}
for Send s { emit Send }
for Receive r { emit Received }
for Control c {
  (c.operation==WRITE && c.target == Fifo
  && c.value == 1 ) ? emit Flush : ignore
}
for MPIC_Request r {
  when MPIC_Pending p:
  either MPIC_Raised r : emit Raised
  else emit Pending
  else ignore
}
```

Fig. 2: STML example

Any raw trace can be reduced to a stream of logical clock ticks. Since the STML compiler requires to handle all possible classes specified in the trace model, the user must specify for each trace item whether it translates into clock ticks or whether it is ignored. Users may have to specify a fairly large number of clocks to distinguish specific cases. When specifying TPSL properties as shown below, it is possible to either specify specific properties for specific clocks or to group them using the union operator.

D. The Property Specification Language

Following the same reasoning as Balarin et al. [6] or Drechler et al. [26], we advocate for a specification language close to the domain and easy to compute rather than as expressive as temporal logics but failing to get a wide adoption among engineers. Another domain-specific language, named TPSL, is dedicated to expressing properties on simulation traces. TPSL[1] also uses the Eclipse Modeling Framework for easy integration in development tools so that engineers can express properties that should be verified on traces. This language first defines the list of logical clocks (that are exported to be generated by the mapping language). We have captured in this language the properties that cover a large part of the CPS engineering applications:

- causality: the fact that when some event occurs or some state is reached, there should be a consequence in the future.
- delay: the fact that two events should not be spaced by more or less than a given (time) value.
- counting occurrences of events and comparing counts.
- specific well known properties such as jitter, latency and drift.

The property specification language in its current version focuses mostly on exploring causality factors. It is our experience that many failures occur because some complex causality fails. Verifying a TPSL property boils down to checking expressions in the CCSL clock algebra. In this paper, we do not re-introduce the formal definition of CCSL algebra. The reader is referred to [28] to obtain the formal definitions. TPSL so far only uses a subset of these clock operators, namely:

- the **union** (noted $+$ in CCSL) of two clocks, that ticks whenever one of them ticks.
- the **intersection** (noted $*$ in CCSL) of two clocks, that ticks whenever both of them ticks simultaneously (as defined above in our context).
- **sampling** (noted $\searrow$ in CCSL) of a clock A by another clock B, which ticks whenever the second clock B ticks after some clock tick of A.
- **subclocking** (noted $\subset$ in CCSL) of clock by some frequency divider.

A TPSL module must first declare the clock identifiers and the constant values that are used in subsequent properties. These declared clocks are called the *basic clocks*. The users may define additional new clocks from the basic ones using one of the following operators:

- **X = A or B** to define X as the union of the two clocks A and B.
- **X = A and B** to define X as the intersection.
- **X = A by N** to define X as the CCSL subclock of A corresponding to frequency divider N.

After the clock definitions the users may define properties related to these clocks. The language has a few English keywords to construct properties. The property language considers only two relations from the clock algebra the alternation (CCSL notation $\sim$) of two clocks, that ticks alternatively, and the precedence of two clocks (CCSL notation $\prec$ or $\preccurlyeq$).

[1] not to be confused with PSL [27]

The properties follow a generic pattern, which is:

```
Scope Clock_Expr Relation Clock Constraints
```

The scope can be either **always** if the property must hold at any time (the default value), or **never** if something must never happen. A Clock Expression results either directly from clock operators or from pattern matching. TPSL allows some kind of regular expression pattern matching on clock ticks in clock expressions, using two operators noted $<$ and $<>$. $A < B$ means that a precondition of a relation is that one or more ticks of A were encountered since the property is active, followed by a tick of B. The expression $A <> B$ is matched whenever A is followed by B or B is followed by A. These pattern matching expressions must not be confused with the precedence relation $\prec$ of CCSL. The relation following a clock expression is either **causes** or **alternates** to mean that two clocks must constantly alternate.

The causality relation is refined in TPSL because it is often necessary to distinguish in practice between necessary and sufficient condition. For example, in a train scenario, it is sufficient that the alarm is raised to stop the train, but the train may stop for other reasons. On the contrary, in a embedded software context, whenever an interrupt is raised, there should be a reason for this interrupt, otherwise it is spurious. TPSL uses the keywords 'causes' to express the sufficient condition and the condition becomes necessary by adding the '!' symbol, thus in TPSL, `Alarm` **`causes`** `TrainStop` and `Signal` **`causes!`** `Interrupt`.

Another distinction in causality is uniqueness. It may be the case that multiple ticks of some clock should result in one single consequence (several reasons can cause the alarm but the train stops only once) or conversely that each action will result into another action (each message sent should be received). The keyword **each** is used to specify the latter.

In addition to the relationship, there may be additional constraints on the property and we distinguish three of them:

- additional conditions that must be satisfied. These conditions may be related to the respective timing of the clock ticks, or the count of tick occurrences. The keyword **satisfies** expresses such conditions. It means that for the property to be verified, not only the clock ticks must obey the rules but there are additional constraints on these clocks, typically a time delay between some ticks.
- suspending condition for the causality, introduced by keyword **unless**, to express that a property has to be verified unless something happened, for example a Cancel order (e.g. `Signal` **`causes`** `Interrupt` **`unless`** `Disabled`). The semantics are that the property is void if the unless clause did occur between the first and the last relevant clock tick in the trace. The unless clause makes it possible to deal with absence in a number of cases, using a property template of the form `Something` **`causes`** `Error` **`unless`** `OK` such as the coffee machine that should return the coin when the system goes wrong `Coin` **`causes`** `ReturnCoin` **`unless`** `CoffeeReady`
- activation condition for the property, introduced by keyword **if**. The semantics are that the property holds only if the condition is true. This is different from unless clause as the predicate is not a clock expression but a Boolean expression on counting occurrences or time stamps. For example, writing into the FIFO may cause the FIFO to be full only if the previous items were not sent already.
- Causality properties specifying **each** for some clock C have a slightly different behavior. No **if** clause can be specified then as it would be contradictory; and the unless clause, if present, only applies to the most recent occurrence of C, but it does not void the causality of previous occurrences.

IV. Implementation and Tests

A. STML

STML is a DSL described in Eclipse Modeling Framework [29] using the Xtext tool and an abstract syntax model also using Ecore. In the current implementation we have chosen to express properties based solely on logical clocks. Therefore STML currently offers basically two choices: a raw trace data item can simply be ignored because it is not relevant to the properties under investigation, or it can be translated into one or more logical clock ticks. Many irrelevant data can thus be simply eliminated from the trace.

To further reduce trace size and articulate related subsequent data items in the trace, STML allows to reduce several successive trace data items into one logical clock event if they meet some condition. Emitting a logical clock tick can be conditional based on some variable value. For example, the $Pending$ interrupt clock can be emitted when some particular bit of the interrupt register is set to 1, but not when it is set to 0. But it can be also conditional to the occurrence of subsequent trace data. A question is thus how long can possibly be that sequence. The STML mapping library does consider only these simultaneous events (relatively to the trace clock) to map them into a single logical clock event.

To that end the mapping algorithm constructs a queue of the trace data items ordered by their timestamp in the simulation clock until the next period of the trace clock (the queue size increases as the trace clock frequency decreases). It then considers the earliest data and checks the STML rule for that item. Either it is simply ignored, or it directly maps to a logical clock tick, or there is a conditional path leading to some other conclusion. That conditional path can be either based on variables extracted from the simulation data (e.g. the N^{th} bit of a hardware register), or based on the eventuality of a later event. In which case that event is searched in the queue. When such a conditional path successfully leads to a clock emit, or to an ignore, all of the matching events are removed from the queue.

Hence, any trace data in STML is considered only once. The same data item cannot participate twice or more into generating a clock tick. However it may be that simultaneous trace data (with respect to the trace clock) generate simultaneous logical clock ticks. As a result, the trace mapping is reducing a potentially huge trace stream into a much smaller one, by eliminating irrelevant data, or by synthesizing complex data possibly expressed on many bytes of data into a single logical clock tick. The logical clock trace stream can then be analyzed with automata resulting from TPSL compiler.

The Java code generator of STML generates the C++ code of the dynamic mapping library. This dynamic library is loaded by the simulator that generate trace files (in our case SystemC models). One can change the STML rules to generate different clocks to check different properties without recompiling the simulator.

B. TPSL Implementation

TPSL is the Trace Property Specification Language. It is also described as a DSL in the Eclipse EMF framework. The TPSL compiler generates code for the Verifier, which consists mainly of the dispatcher and the parallel automata. The dispatcher dispatches basic clock ticks to the parallel automata. Since the TPSL compiler knows which clock operators and which automata depends on which clock ticks, it can directly dispatch the basic clock input to their destination. A clock tick may be dispatched to multiple automata as the rules are verified in parallel.

The automata generated by TPSL compiler are symbolic finite state transducers (SFTs) [30], [31], which are an extension of traditional automata by allowing transitions to be labeled with arbitrary formulas in a specified theory, in our case Boolean expressions over clock ticks.

The alphabet for the transducers is the clock alphabet, but the transducers include predicates that operate over a finite set of clock ticks. When taking transitions, the transducers add information to the *history* of the automaton. The history is a summary of what happened during the evaluation of the property. It maintains the timestamps and the counters of the first and last tick of each kind involved in the automaton, or the absence thereof. The TPSL language has been designed precisely such that maintaining the history is sufficient to evaluate the transducers predicates and no other data is required. The history has enough information to guarantee the decidability of the predicate bearing on the **unless** or **if** or **satisfies** clauses. The history is reset to void each time an automaton enters its initial state.

By construction of TPSL, the automata all follow some particular template. The TPSL compiler has several such templates that it can instantiate with the particular clock expressions and predicates from the source code. This can be implemented using C++ template facility combined with a functional style since C++ 11 standard supports lambda expressions. Figure 3 provides the diagram of the automaton for the template`<ClockExpr> causes <Expected Clock> if <Condition> unless <Cancel Clock> satisfies <Constraint>`

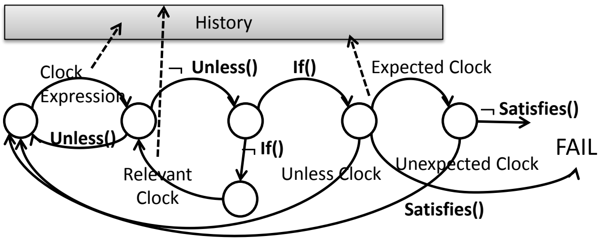

Fig. 3: Automaton template with if and unless clauses

Each template automaton has a specific behavior, but they all follow the same generic pattern: from the initial state (when it is active), each transducer reaches a checking state when the clock expression has been encountered. The clock expression is actually a dependent automaton, also generated from a template. The transition is taken in the property automaton when the clock expression automaton terminates. Based on the predicates result, the transducer will either return to the initial state, or stay in the same state, or move to the expecting state. It then expects some clock tick to occur to meet the required causality. If it does, it will transition to the satisfying state, and then, depending on the required condition to be satisfied (which also bears on the history) it returns to its initial state or fails. While in the expecting state, other ticks may occur, for example a tick that would make true the unless clause, or ticks that modify the count in the history. Such clock ticks are told to be *Relevant Clocks*. An automaton may take a transition when some such relevant tick occurs. The TPSL compiler knows what these relevant ticks are and can generate code that dispatches only such relevant ticks to each automaton, and generate appropriate tests in the automata to take the transition.

The **each** automaton template is particular. As shown on Fig. 4, it is a potentially infinite automaton that waits for N expected clocks after N causing clocks. It will return to initial state only when the counts of occurrences are equal.

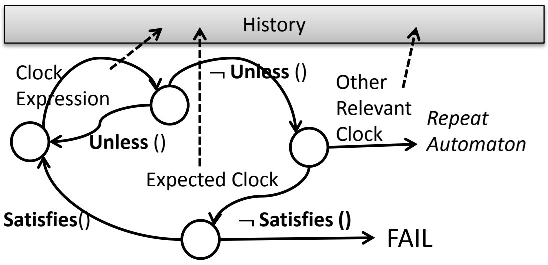

Fig. 4: Automaton template for each clause

C. The Verifier

The verifier takes as input a trace file and outputs either a success message or one or more failures in the properties verification. Some part of the verifier is pre-defined (manually coded), essentially a library of base functions, the automaton templates, and the skeleton of the main program. This predefined code is linked with the code generated by the TPSL compiler to form a program that can read trace streams. It works essentially as follows: each line of a trace file has a time stamp (in the trace clock) followed by all clock ticks that occured on that timestamp.

The verifier first calls a setup function that creates all of the parallel automata generated by the TPSL compiler and it activates those that are in the **always** or **never** scope. It then repeatedly reads the trace stream line by line and calls the dispatcher, which does dispatch the clock ticks to each individual active automata. During this process some automaton may become active or inactive, and some may report failure. At the end of the trace stream, the automata are checked again to know if they have reached their final state. For any property, if the end of the trace is reached while a transducer is still in expecting state, it is a failure.

D. Tests

We have instrumented the SimSoC PowerPC Instruction Set Simulator and several simulation modules to generate traces according to our model. The main experiment simulates a multi-core PowerPC processor connected to various peripherals, including a OpenPIC multi-core interrupt controller and a transmit controller with a FIFO queue, controlled by a real time monitor that has drivers monitoring the peripherals. The hardware FIFO queue receives items asynchronously from the software but it re-sends these items synchronously. The hardware should (i) emit a $FifoEmpty$ signal whenever the queue is empty (including at initialization phase), a $FifoFull$ signal when it is full, (ii) send the data items with limited jittering (iii) have a reasonable latency. On the other hand, the software should (i) enable interrupts and service them, and (ii) it should not put data into the queue when it is full.

The embedded software is a test program that emits data to the outside world, using the drivers and receiving interrupts. The software must enable interrupts, service them and acknowledge the interrupts with the hardware. On a multi-core configuration like the one considered, this induces a cascade of acknowledgments and signals. The full simulation trace contains data regarding hardware states, transactions between the cores and the controllers and events happening consequently to software actions.

The TPSL code regarding our example is:

In this case, the trace reduction is very significant, as we are only verifying properties related to the FIFO queue and consequently many other hardware states are simply ignored. The trace file emitted by the test on SimSoC for a duration of 27 seconds is 21.162 Megabytes, the size of the logical clock trace is only 1.737 Megabytes, a reduction of over 90%. The trace is analyzed by our tool in less than 1 second.

V. Conclusion

We have demonstrated the feasibility of runtime verification of system properties for investigating system failure from simulation trace analysis, by abstracting raw binary simulation trace files into logical clock trace files. Thanks to a DSL that we have defined based on a simple yet powerful trace meta model, the huge binary trace files can be reduced and mapped to much smaller abstract traces relating occurrences of logical clock ticks. A language has been defined to capture temporal properties, compiled into parallel automata executed by a runtime verifier.

Our goal is not to formally offer more expressive power than any variant of temporal logic. Our tool is meant to be a convenience tool for engineers that have limited knowledge of formal methods but still can express temporal and causal properties in a easy-to-use language that is supported by a user-friendly environment, namely the Eclipse platform.

```
clock Put Send FifoFull FifoEmpty;
clock Signal Interrupt Pending ACK;
constant int FifoSize = 16;

//the hardware must not send when fifo empty
never Send between FifoEmpty Put;

// the software must not put when fifo full
never Put between FifoFull Send;

// each put must be balanced by one send
Put each causes Send
Put causes FifoFull
      if count Put - Send = FifoSize;
Send causes FifoEmpty
     if count Send - Put = FifoSize;
FifoEmpty causes Signal;

// after Signal has been detected as Pending
// there must be an interrupt interrupt handled
// by service routine that must ACK
Signal < Pending causes! Interrupt;
Interrupt causes! ACK;
```

Fig. 5: TPSL code for our example

Another advantage of our technique is that it does not require to recompile the simulator to change the mapping and verify different types of properties. Different mappings can be achieved and different properties verified from a single trace. This provides a mechanism for the engineers to explore several paths for investigating a failure cause.

The SimSoC implementation has been modified so that a new argument can specify the trace mapping module to be loaded. In such case, the trace stream is generated such that it can be pipelined with the verifier. The verification can occur at runtime, concurrently with the simulation (possibly using at least two of the cores on a multi-core platform, introducing no penalty on the runtime verification), somehow like an extension of an assertion verifier, except that the assertions can be modified and checked without recompiling the simulator. In addition, our system can also work as a static analyzer for trace data stored into files.

In the current implementation, the mapping language translates binary data into genuine clock ticks. We are considering in the future to associate attributes to the clock ticks, then the predicates in the transducers could also evaluate Boolean expressions over such attributes, which would make it more powerful.

References

[1] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamidem, and Y. Lahbib, "Combining system level modeling with assertion based verification," in *Sixth Int. Symp. on quality electronic design (isqed'05)*, pp. 310–315, March 2005.

[2] H. Foster, "Assertion-based verification: Industry myths to realities," in *Computer Aided Verification*, pp. 5–10, Springer, 2008.

[3] D. Krahl, "Debugging simulation models," in *Proceedings of the Winter Simulation Conference, 2005.*, pp. 7 pp.–, Dec 2005.

[4] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[5] A. Pnueli, "The Temporal Logic of Programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, (Providence, RI, ISA), pp. 46–57, 1977.

[6] F. Balarin, J. Burch, L. Lavagno, Y. Watanabe, R. Passerone, and A. Sangiovanni-Vincentelli, "Constraints specification at higher levels of abstraction," *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, vol. 0, p. 129, 2001.

[7] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Automatic trace analysis for logic of constraints," in *40th Design Automation Conference*, (Anaheim, CA, USA), pp. 460–465, IEEE, 2003.

[8] W. Hong, A. Viehl, N. Bannow, C. Kerstan, H. Post, O. Bringmann, and W. Rosenstiel, "Cult: A unified framework for tracing and logging c-based designs," in *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pp. 1–6, Sept 2012.

[9] D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman, "A temporal language for systemc," in *Formal Methods in Computer-Aided Design, 2008. FMCAD'08*, pp. 1–9, IEEE, 2008.

[10] M. Khlif, M. Shawky, and O. Tahan, "Co-simulation trace analysis (cosita) tool for vehicle electronic architecture diagnosability analysis," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pp. 572–578, 2010.

[11] K. Bhargavan, C. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan, "Verisim: Formal Analysis of Network Simulations," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 129–145, 2002.

[12] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally Specified Monitoring of Temporal Properties," in *11th Euromicro Conference on Real-Time Systems*, (York, UK), pp. 114–122, 1999.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Int. Conf. on Software Engineering*, pp. 411–420, IEEE, 1999.

[14] S. Konrad and B. H. Cheng, "Real-time specification patterns," in *27th Int. Conf. on Software Engineering*, pp. 372–381, ACM, 2005.

[15] S. Some, R. Dssouli, and J. Vaucher, "From scenarios to timed automata: Building specifications from users requirements," in *Asia Pacific Software Engineering Conference*, pp. 48–57, IEEE, 1995.

[16] W.-T. Tsai, L. Yu, F. Zhu, and R. Paul, "Rapid embedded system testing using verification patterns," *Software, IEEE*, vol. 22, no. 4, pp. 68–75, 2005.

[17] A. M. Tokarnia and E. P. Cruz, "Scenario patterns and trace-based temporal verification of reactive embedded systems," in *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 734–741, IEEE, 2013.

[18] Á. Hegedüs, I. Ráth, and D. Varró, "Replaying execution trace models for dynamic modeling languages," *Periodica Polytechnica. Electrical Engineering and Computer Science*, vol. 56, no. 3, p. 71, 2012.

[19] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries.," in *PARCO*, vol. 22, pp. 481–490, 2011.

[20] A. Hamou-Lhadj and T. C. Lethbridge, "A metamodel for the compact but lossless exchange of execution traces," *Software & Systems Modeling*, vol. 11, no. 1, pp. 77–98, 2012.

[21] F. Mallet, C. André, and R. de Simone, "CCSL: specifying clock constraints with UML/Marte," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 309–314, 2008.

[22] R. Gascon, F. Mallet, and J. DeAntoni, "Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL," in *Int. Symp. on Temporal Representation and Reasoning, TIME'11*, pp. 141–148, September 12-14 2011.

[23] J. DeAntoni and F. Mallet, "Timesquare: Treat your models with logical time," in *50th Int. Conf. on Objects, Models, Components, Patterns, TOOLS'12*, pp. 34–41, Springer Berlin Heidelberg, 2012.

[24] C. Helmstetter and V. Joloboff, "SimSoC: A SystemC TLM integrated ISS for full system simulation," in *IEEE Asia Pacific Conference on Circuits and Systems, APCCAS*, Nov. 2008. http://formes.asia/cms/software/simsoc.

[25] INRIA, "SimSoC open source software." http://gforge.inria.fr/projects/simsoc.

[26] R. Drechsler, M. Soeken, and R. Wille, "Formal specification level: Towards verification-driven design based on natural language processing," in *Forum on Specification and Design Languages, FDL'12*, pp. 53–58, Sept 2012.

[27] IEEE, "Property Specification Language (PSL)," 2010.

[28] C. André, "Syntax and Semantics of the Clock Constraint Specification Language (CCSL)," Research Report RR-6925, INRIA, 2009.

[29] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[30] G. van Noord and D. Gerdemann, "Finite state transducers with predicates and identities," *Grammars*, vol. 4, no. 3, pp. 263–286, 2001.

[31] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic finite state transducers: Algorithms and applications," *SIGPLAN Not.*, vol. 47, pp. 137–150, Jan. 2012.