TRAP: A Platform For Flexible Runtime Verification of Temporal Properties

Daian Yue

RESEARCH REPORT
N° 9006
December 2016
Project-Teams TEA

TRAP: A Platform For Flexible Runtime Verification of Temporal Properties

Daian Yue*

Project-Teams TEA

Research Report n° 9006 — December 2016 — 43 pages

Abstract: The TRAP Trace Runtime Analysis Platform provides a model-based framework and implements the corresponding tool chain to support runtime analysis and verification of traces generated by virtual prototypes or cyber physical systems. The main goal is to make it easy for engineers to define system properties that should be satisfied and verify them at system runtime (or from a recorded session). The property verification tools proposed do not require a detailed knowledge of the system implementation, do not require any modification or recompilation of the system to investigate different properties, and do not require the engineers to be familiar with temporal logic. TRAP proposes Domain Specific Languages (DSL's) integrated within the Eclipse Modeling Framework to express the properties. The DSL toolchain uses the concept of Logical Clock defined by CCSL and takes advantage of CCSL clock algebra as the underlying formal support. The DSL's compilers eventually generate C++ code to verify the properties at run time, making usage of dynamically loaded code.

Key-words: cyber physical systems, embedded systems, formal methods, verification, temporal properties, Eclipse Modeling Framework, automata, simulation

* Intern from East China Normal University under Prosfer program with ENS Rennes

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Verification Dynamique de Propriétés Temporelles

Résumé : La plateforme d'analyse de trace en runtime TRAP fournit un outillage dans le cadre de l'ingénierie dirigée par les modèles qui permet l'analyse et la vérification de traces pour des prototypes virtuels et des systèmes cyber physiques. Le principal objectif est de faciliter la définition de propriétés qui doivent être satisfaites, et leur vérification soit pendant le fonctionnement du système, soit sur des traces enregistrées. Les outils de vérification proposés n'exigent pas de connaitre dans les détails l'implémentation du système, et aucune recompilation ou reconstruction du système n'est nécessaire pour explorer différentes propriétés. Les langages proposés ne requièrent pas non plus de connaissance préalable de la logique temporelles pour ceux qui les manipulent. TRAP propose des Langages Spécifiques de Domaine (DSL) intégrés dans les outils Eclipse. Ces langages reposent sur le formalisme CCSL avec sa notion d'horloge logique et l'algèbre d'horloges associée. Les compilateurs des langages génèrent du code C++ qui vérifie les propriétés temporelles en faisant usage de chargement dynamique de code.

Mots-clés : cyber physical systems, embedded systems, formal methods, verification, temporal properties, Eclipse Modeling Framework, automata, simulation, property specification, CCSL, temporal logic

1 Prologue

The Ecole Normale Supérieure de Rennes (ENS Rennes), one of the French ENS schools and East China Normal University (ECNU) have created under the PROSFER agreement, a joint education program proposing courses taught at ECNU to selected Chinese students with possibility for these ECNU students to come to France for internship or possibly registering in the ENS PhD cursus.

This project was initially started by Dr. Vania JOLOBOFF when he was invited scientist at ECNU, where I am studying as a master student and joined his project. I have been accepted as an exchange student in the ENS/ECNU program, but the research work has been carried out at the IRISA/Inria/ENS joint laboratory, working in the INRIA TEA Project Team.

2 Introduction

Since System-On-Chip (SoC) has been widely used in many places in daily life like cars, subways, airplanes etc. Modeling and simulation of SoCs gained lots of attraction from both academic and industry world. Among the popular approaches, Virtual Prototyping (VP) provides a good solution to model SoC and makes it possible to simulate and test hardware components or build and execute embedded software inside SoC simulator.

Since there are usually constraints for both hardware and software designs to satisfy, one of the traditional ways to find bugs from VP simulator is to analyze the traces generated during the execution of programs. However, during the execution of VP simulators, it often generates huge amount of traces which makes it really hard to extract useful information even with the help of regex tools, not to speak of the analysis of property violation.

This project, Trace Runtime Analysis Platform (TRAP) is started to provide a model-based framework and implement corresponding tool chain to support runtime analysis and verification of traces generated by VP simulators. The overall goal is to reduce the size of traces generated, and make it easy for engineers to define the properties, then to automatically analyze the properties according to the execution of simulators.

3 SimSoC Simulator

3.1 Overview

The development of embedded systems platforms requires increasingly large pieces of software running on complex System On Chips (SoC). It is necessary for engineers to have a simulation environment to simulate the system under design so that software developers can test the software and hardware developers investigate alternative designs.

For the embedded software developers, the simulation environment must achieve full system simulation: it must have exactly the same behavior at least when executing binary inputs; also it must run fast enough for interactive testing and fast software verification cycles. However, a full system simulation at very low level of hardware detail (e.g. cycle accurate) is much too slow for software testing. These requirements call for an integrated, modular full simulation environment which can simulate proven hardware components quickly whereas new IP under design can be tested separately and thoroughly.

So in around year 2008, Vania JOLOBOFF together with many other people brought up such an simulation environment - SimSoC [1], which is an open source software, copyrighted by INRIA in France, and distributed under the GNU LGPL license.

SimSoC is a Virtual Prototyping (VP) framework to fully simulate hardware platforms, and more generally Cyber Physical Systems. The cornerstone of SimSoC is the simulator. This simulator is meant to be used by either software developers who can validate their software or by hardware developers who want to validate an architecture and/or calibrate hardware design to meet the application requirements. SimSoC is not a cycle accurate simulator. It is a bit accurate simulator. It simulates the hardware at so-called "programmer's view", that is a hardware abstraction level that given some input provides the same output as the real hardware, but the computation and communication techniques inside the virtual device might be different from the real one. The simulator is based on simulation models implemented in SystemC and using Transaction Level Modeling (TLM) to communicate among each other.

3.2 Definitions

The idea is to run the exact application binary software on the simulation framework as described in diagram below.

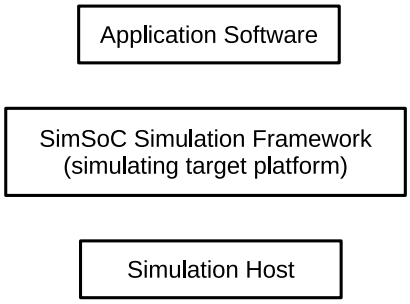

Figure 1: SimSoC execution diagram

The *target* platform is simulated on a *host* machine. For example, the ST Spear600 platform is simulated on a PC. One must simulate the *instruction set* of the target, the *state* and *behavior* of the processor. For example, a comparison instruction usually positions a condition code that must be emulated in the machine state, and the simulation must simulate that.

The target instruction all together form the *instruction set*. The instruction set is simulated by the Instruction Set Simulator or *ISS*. An ISS is *instruction accurate* when the state of the CPU is known between each instruction; an instruction accurate ISS can abstract the micro-architecture of the CPU (e.g. pipelines).

SimSoC consists of two parts: the simulation of some hardware components in SystemC/TLM, and the simulation of other components, in particular off-the-shelf processors, as illustrated in diagram 2.

The simulator of a particular processor is named a CPU simulator. A machine simulation therefore includes three parts: simulation of each individual instruction, simulation of state and behavior, simulation of communication with devices.

To simulate accurately some processors it may be necessary to also simulate one or more specific co-processors.

Communication with peripheral devices, or peripherals, must be simulated too. SimSoC assumes that all simulated processors are carrying I/O's through memory-mapped I/O's, that is, communication with devices is carried through load-stores on specific memory addresses, also called I/O ports.

To simulate state, a data structure is defined called the state. The environment contains in particular the simulated CPU registers, both general purpose registers and state registers of the

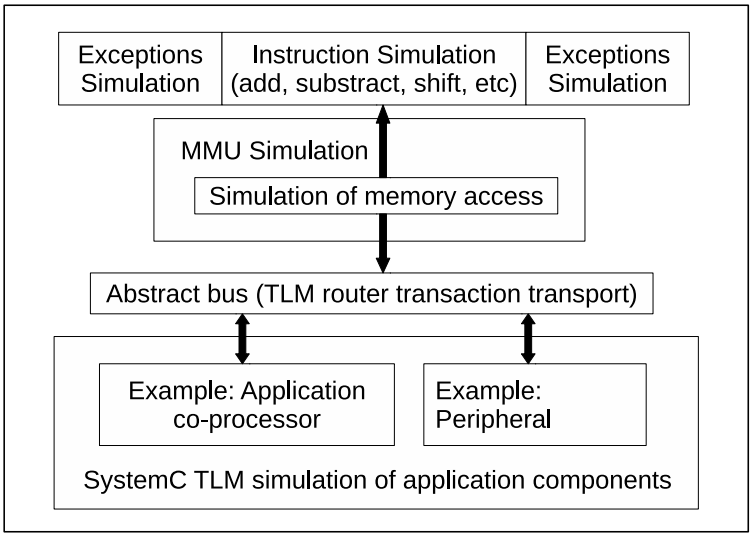

Figure 2: SimSoC architecture

machine. At any time, it must reflect accurately the state of the simulated CPU.

Processors execute instructions that are referred to by the register of Program Counter (PC).

Most processors nowadays use a Memory Management Unit (MMU). In particular, the mechanism to translate a virtual address into a physical address is usually handled by the MMU. The MMU must be simulated; the simulation includes the behavior of virtual memory management.

3.3 Overview

SimSoC is activated with a command line that takes a number of arguments, and possibly a configuration file. These arguments are given to the SystemC elaboration code, which build the architecture of target platform.

Each hardware component of the target platform is simulated by a SystemC module. In particular, the CPU simulator with its MMU is included in a SystemC module. The MMU generates SystemC-TLM transactions to the other devices trough its initiator TLM port. The CPU simulator implements a function to receive interrupts, which are forwarded by the interrupt controller (IRQC).

Target memory is simulated with host memory. Memory is implemented as a TLM target module. It can receive transactions from any module in the system but it also provides a direct access to memory for the CPU simulator. At start-up time, a large memory space is allocated on the host, equal to the size of the simulated memory. Therefore, assuming the beginning address of that space on the host is "a", then the translation of physical address "b" of the target is simply "$a+b$" on the host. Also, It is assumed that virtual memory is paged.

4 Modeling Frameworks

In this section some modeling frameworks are introduced. The tool chain mainly uses modeling frameworks related with Eclipse that will be introduced in detail.

4.1 Eclipse Modeling Framework

Eclipse is an Integrated Development Environment (IDE) which has been widely used for various development for more than 10 years. It was initially designed for Java development, and because of its plug-in system, its function has been expanded into a full-featured IDE for other programming languages. In addition, Eclipse can be used for building models by using Eclipse Modeling Framework (EMF).

By the definition from its official website `http://www.eclipse.org/emf`, "The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor." In addition to be a complement modeling framework, EMF has also been used as a stable standard by many other technologies in the Eclipse world.

The core EMF framework consists of three fundamental parts:

- EMF Core: The core of EMF framework includes a meta model (Ecore), which can be considered as a UML dialect, for describing models, with a runtime which has a set of class system and support for reflective API of EMF objects and some other features.
- EMF Editor: A programmable framework which allows developers to program on the behavior of modeling editor and customize some features such as content and label provider etc.
- EMF Code Generator: The EMF code generation facility is able to generate everything needed to build a complete editor for EMF model. It includes a GUI which can be used to specify generation options and invoke generators.

Apart from being used directly as a modeling tool, Ecore file format introduced by EMF is also used by other modeling frameworks to hold information. Ecore becomes a generally used exchangeable file format between frameworks built upon Eclipse environment.

4.2 Textual Modeling Framework

Textual Modeling Framework (TMF, also known as Xtext) is both a framework and a tool for developing Domain Specific Language (DSL). After defining the grammar with a meta language, Xtext will generate lots of commodities for the DSL such as a parser, a syntax checking editor, and utilities to facilitate type checking and code generation from the DSL. Xtext uses EMF to construct an Abstract Syntax Tree (AST) and generate a fully featured Eclipse syntax directed text editor, that may have many features like text coloring, error checking, formatting, refactoring and so on. Reusable Java classes are generated by Xtext and DSL developers may manually write code to provide additional functions operating on the generated structures. Here lists some Xtext features mainly used by the platform:

- The AST. The parser automatically generated from the grammar (using the ANTLR technology from http://www.antlr.org/) constructs an AST, the nodes of which are customizable in the grammar definition. By default, it constructs an AST that directly reflects the concrete syntax, but one may specify otherwise in the grammar.
- Validator. The Xtext framework contains a validator which can be customized to check if the DSL instances contain any error. The validator can be set to be triggered automatically after the user finishes typing, or when the user saves the file, or by manually clicking. The

reflective feature of Xtext framework makes it possible for the validator to access the current context of each grammar element to be checked. When errors are detected, the validator may provide error marks and quick fixes programmed inside the validator. Also, some validation code can be specified to be called only to check some sub-tree of the AST, using the "@Check" annotation mechanism.

- Generator. when the user saves a DSL file, the Xtext framework calls the validators and if no errors are reported, it then calls the generator The generator can be used to generate code from the DSL, or whatever function is needed after the user saves the DSL file. Inside the generator, it is possible to access the AST. After the generator finishes execution, Xtext restarts its current loop.
- IDE support. The Xtext framework offers support for several different IDEs to edit the DSL instances. For Eclipse, DSL developers can provide many nice features like content consist, labeling, outline etc. These features can be very helpful when the DSL is complex and the user may get code template or other convenience. For example, when the user presses `ctrl` + `space`, Xtext shows a pop-up window to let the user select from possible completions. This feature can be programmed and provide many customized completions.

The Xtext framework also has some other facilities such as cross reference and scope, unit tests. It simplifies the development of DSL and makes it easier to integrate with other facilities.

4.3 Xtend the Language

In the Xtext framework, the programming language can be either Java or Xtend. Xtend is a statically-typed programming language based on top of Java. In fact, the Xtext framework will parse Xtend codes and generate corresponding Java codes from Xtend. The grammar of Xtend looks similar to Java, with less redundant and some helpful concepts which may help speedup the development:

- Operator overloading. Like C++, Xtend make it possible to overload operators for any type by implementing the right method signature.
- Type inference. Xtend has a strong type inference system. Type of variables or methods can be automatically inferred according to the context. So that developers do not have to specify the type explicitly.
- Multiple dispatch. It is also called *polymorphic method invocation*, meaning that methods can be invoked according to the inheritance of method arguments. In Java, polymorphic only works with objects which invoke methods instead of arguments. Xtend supports keyword *dispatch* which will add multiple dispatch support to a trivial function. The translated Java is actually a series of if-else clauses with `instanceof` checking.
- Powerful switch expressions. *switch* expression in Xtend not only works with trivial value types like Java, but also works with complex data types such as String and other abstract data types. Xtend uses *equals* method to compare the values. Instead and in addition to *case* guard, Xtend supports type guards which can check the type of *switch* argument. Note that a type guard and normal predicate can be used together which will be satisfied when both guards match.
- Expressions instead of statements. Xtend has no concept for *statements* because everything in Xtend is an expression so that can return a value. Together with type inference, this would save a lot of time.

Xtend adds some new concepts from other languages together with some syntax sugar. In Xtext framework, it is somehow equivalent to use Xtend versus Java. But Xtend code is usually more readable and expressive compared with original Java code.

5 Project Goal

When a user is designing a hardware component or developing embedded software, it is usually helpful to test the product on a simulator first. In both cases, the system usually has some constraints to satisfy. If the developers wants to find the bugs inside the system where constraints are violated, they have to investigate the actual execution of the hardware/software system in order to analyze it. A widely used method consist in recording data during execution of simulation. The recorded information is called the *trace*.

Running many executions of a VP simulator with different input conditions, it generates a lot of traces that record the various simulation sessions. The traces are usually dumped into trace files, and the traditional way to analyze errors is to look into the trace file and study what has happened during the simulation sessions. For example, if users are running software upon SimSoC and want to tell if there is any violated constraint, they have to dig into the trace file and compare it with the correct behavior. There are two major problems during manual analysis:

1. It is hard to extract useful information from a huge trace file. A trace file generated by a VP simulator often contains very detailed information about the system, including some internal states, and the trace file can reach gigabytes of data, which is too huge to handle manually.

2. It is difficult to reason about the detailed traces. Because the trace often contain raw binary data and not symbolic information, it may not be obvious to reason about a long chain of causality events that eventually lead to the failure. The too low level detailed trace information makes it hard to reason at abstract level.

In order to simplify the complexity during trace analysis, we propose a platform that makes it possible to reason at a higher level of abstraction, and does automated verification of the trace to find violated requirement. Our platform relies on the concept of *logical clocks* as proposed in the Constrained Clock Specification Language (CCSL). Our platform is ultimately verifying properties expressed as logical clocks formulas that must be satisfied, the logical clocks being extracted from the trace files.

In a first step, we want to reduce the size of traces generated during one execution of simulation and express the trace streams at a higher level of abstraction. The platform must provide a way for modeling the trace, filter out the low-level details, and possibly synthesize complex data structures from the raw trace into a single logical clock tick (reduce many bytes of data to a few). We provide to that end a concept of *event model*, derived from a meta-model, which can be *mapped* in an automated way to the abstract level of logical clocks.

Next, the platform provides a specification language to express the property constraints of the hardware/software design. The language should be carefully designed to be both simple and powerful:

1. For engineers, it should be simple enough to learn so that engineers can use it to express the system properties very quickly. We do not want to set as a pre-requisite for the engineers to be experts either in linear temporal logic, or in CCSL algebra.

2. Considering the function considered, it should be powerful enough to cover most popular cases and provides possibility to express some non-common but useful situation. Note that, because trace files are by construction finite, we do not need to reason on infinite traces...

3. For future extension, it should be flexible enough to allow future development of the language to add more expressions.

Finally, the platform should have a *verifier* which takes the property specifications and traces generated by one execution, and verify if any property is violated during the execution. The verification should be automatically done with an easy-to-use user interface and gives analysis result in a readable way. Also, one should be able to invoke the verifier in two modes: offline mode in which the verifier takes a trace file and analyzes it; runtime mode in which the verifier takes the traces generated by simulator on the fly. If any error is detected during the verification, the platform should provide a GDB-like *debugger* which makes it possible to analyze the trace piece by piece to find out the root of problem.

In addition, some other features are also interesting for the developers. In real embedded hardware/software applications, there are most often different phases/stages during an execution. The platform should support the notion of *scope* so that some properties only take effect in some specific scopes. The scopes should be able to overlap and be nested. Different scopes do not affect each other, and are independent from trace events.

Another aspect is that recompiling a simulator can be tedious, and sometimes impossible if all of the source code is not available. In order to facilitate usage of our property checker, we have added the constraint that modifying the trace generation for checking new properties should be fully dynamic and should not require recompiling the simulator.

In conclusion, the goal of this project is to provide a platform that let users to define trace event, how these events map to logical clocks and scopes, what are the properties that should be verified, and then automatically analyze and verify the traces generated by the simulator on the fly, and possibly debug the traces if any errors are detected; all of this without having to recompile the simulator when checking different properties based on difference clock specification.

6 Related Works

As mentioned above, the goal of this project is to overcome the difficulty of analyzing simulation traces and verifying the properties. Trace analysis has been a topic for some time in the simulation community, so that many people have worked on it and gave some approaches.

Several languages in the literature allow to express temporal properties. Linear Temporal Logic (LTL) [2] abstracts the system behavior as an infinite sequence of states and establishes either properties of *invariance* of *eventuality*. All the properties are expressed relative to the steps of execution, using model temporal operators, and not relative to an external clock. There are also some other formal specification language designed for trace verification such as Computational Tree Logic (CTL) [3] and Graphic Interval Logic (GIL) [4]. However, all of these specification languages cannot be used to specify real-time properties because none of them support quantitative reasoning about time. Later, several efforts such as Metric Temporal Logic (MTL) have been made to provide real-time extensions for LTL.

However, languages like LTL and its extensions can be very complex in practical use. The users must know well about the formal theory behind the language to use it well. In order to provide a higher level of abstraction which can be used to express constraints in real-time systems, Balarin and al. [5] introduced a new specification language, named *Logic Of Constraints* (LOC). Later on, Chen [6] has used LOC as theoretical basis and built an algorithm to generate model

checker in C++ to verify the properties specified in LOC on simulation trace. After using LOC to express annotations including sequential index, time and causes relation, the generated checker can verify properties such as rate, latency, jitter, etc.

As SystemC is widely used by the embedded system community, the specific Value Change Dump (VCD) trace format has been defined for tracing and many commercial or non-commercial tools have been developed for analyzing it. The C-based Unified Logging and Tracing (CULT) [7] system has introduced a flexible trace generation framework for C-based designs of embedded hardware/software systems. The framework is highly configurable and scalable, but there is no associated tool to analyze the traces produced. A SystemC analyzer like [8] can be used to specifically verify temporal properties holding in the SystemC processes, but it requires access to the SystemC modules and cannot check information produced by embedded software running over the simulated hardware.

The COSITA tool [9] makes it possible to analyze traces resulting from co-simulation between SystemC and Matlab models for automotive applications. The tool uses VCD to dump the execution trace of their platform and feed the trace to both simulator and emulator to compare the difference, but there is no property specification language defined from the paper. This paper [10] is used to verify properties in traces of packets in network simulation, using the Monitoring And Checking (MAC) framework [11]. It is based on a logic for events and conditions, which allows to consider instances of events by means of counters, similar to LOC.

Referring to property specification language, Dwyer *et al* published in 1999 a classic paper [12] which did a survey from many sources and developed a pattern system. Their pattern system covers most popular situations. Based on their research, this paper [13] builds a new pattern system containing some real-time extension. Also, they studied many formal specification languages and provides an English-like grammar which can be rather convenient. However, their paper does not refer to the implementation, thus there is no information of how to really apply their pattern system to the real simulators or on the real simulation traces.

Regarding the tool aspect, the Open Trace Format 2 [14] provides a flexible trace format that inspired our mapping work.

7 System Design

7.1 Architecture

The global architecture of this project is decomposed as shown in figure 3. One complete execution instance of TRAP contains four independent steps:

1. First the simulator runs and generates well-formed *trace items*, which are data structures defined inside trace item model.

2. In the second step, the trace items are transformed into corresponding *clock ticks* according to the mapping rules that is expressed by a pre-defined DSL. After that, simulator is able to generate stream of logical clock ticks in some fixed format.

3. In the third step, the trace stream is fed into the verifier and system properties are expressed by another pre-defined DSL.

4. Finally the verifier analyzes the stream of clock ticks and checks if properties are satisfied and gives the analysis result.

In the figure 3, the gray rectangles represent the input of the platform that will be implemented by the end user. All the DSL languages for specifications are defined using Xtext

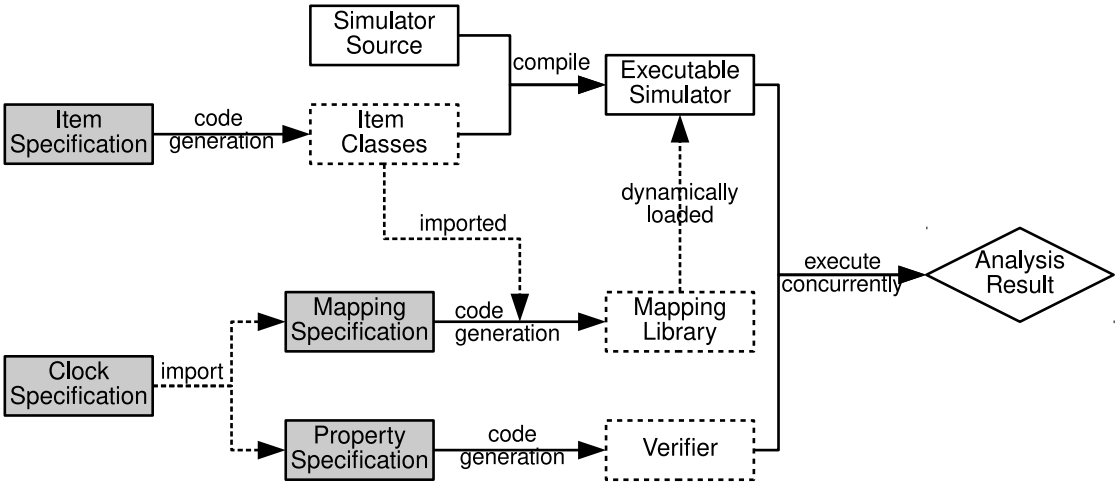

Figure 3: Platform architecture

framework. The dashed rectangles represents C++ components that will be generated according to the input. During the development of TRAP, SimSoC is used as the simulation back-end. However SimSoC and TRAP are loosely coupled and by definition, TRAP can be integrated with any simulator back-end as long as the simulator uses interface provided by TRAP to generate traces. The interface and requirements will be introduced in detail in section 14.

7.2 Main Components

Given any simulation platform, there must be a corresponding specification describing information structures. It represents what kind of information it can provide to its simulator user. TRAP defines a DSL for this purpose named Trace Item Specification Language (TISL). This language defines all the trace items and corresponding data types used in next process. Each trace item stores a basic information structure that can be a hardware interrupt, a designed event or a plain information block.TRAP generates C++ code according to the specification. The generated code together with some previously written code can be integrated into simulator and compiled together to get an new executable simulator. Inside this new simulator, the traces are generated via interface provided by TRAP will generate a stream of *trace items* instead of raw state changes.

Runtime verification is based on the *logical clock* notion from Clock Constraint Specification Language (CCSL) [15]. In this theory, the basic notion of trace is a infinite sequence of clock ticks. TRAP takes the notion *logical clock* to represent a basic "event" during the execution. Logical clocks in TRAP are then used to describe system properties and they are defined in the clock specification DSL named Logical Clock Specification Language (LCSL). Since TRAP analyzes the trace generated during one execution of simulator, it does not deal with the infinite traces and this is a big difference between TRAP and CCSL based tools.

After trace items and logical clocks are defined, the next step is to provide mapping rules telling how patterns of trace items are transformed into logical clocks. This step is inspired by Open Trace Format 2 [14] and it converts a trace item stream into a stream of logical clock ticks. Mapping rules are defined via Simulation Trace Mapping Language (STML). TRAP takes the DSL instance and generates code consisting dynamic library for mapping. This library will be loaded dynamically by executable simulator to do the transformation. Since dynamic

library and simulator works through an interface, modification of STML instance does not require recompilation of simulator. After this step, simulator is ready to run and generate clock tick stream.

Since the goal of this project is to provide runtime trace analysis, the most important part is to express system properties. Another pre-defined DSL named Trace Property Specification Language (TPSL) is used to specify specify system properties. TRAP takes DSL instance and generates C++ code that will compose with verifier and verify the traces for corresponding properties. Each time the properties are changed, the verifier must be re-compiled with newly generated code.

Verifier contains two parts: 1) the basic library providing template-like facilities; 2) code generated from TPSL that uses the library. Verifier can be invoked via command line interface. It can work in three modes:

1. Offline mode. Verifier takes a trace file generated by simulator and analyze it line by line. If there are any properties that are not satisfied, it gives an error. After all, it gives a summary of all the properties according to the trace file.

2. Pipe mode. Verifier analyzes the trace stream just like offline mode, except that under this mode, it runs simultaneously with simulator and reads clock ticks from a named pipe.

3. Debug mode. Verifier takes a trace file and waits for the user input via a command line interface. It provides tick-by-tick analysis and so on according to the user command.

After this step, the analysis result is given and one cycle of execution is finished.

7.3 Summary

In summary, TRAP is used as follows from a user's perspective:

1. Basic trace items and logical clocks are defined using TISL and LCSL respectively and the simulator is compiled together with code generated in this step.

2. Mapping rules are provided using STML and a dynamic library are compiled using code generated in this step.

3. System properties are specified using TPSL, verifier is compiled with generated code.

4. Simulator loads the mapping library and generates trace stream, and verifier takes the trace and analyze it.

In the chain of TRAP, simulator, mapping library and verifier are weakly connected with each other so that modification of one piece does not affect the others. This flexible way makes it possible for engineers to work in parallel and then merge their works. In the rest of this section, these components of this platform will be introduced in detail.

8 Trace Item Specification Language (TISL)

8.1 Overview

TISL contains every necessary information for trace emission and is closely related to simulation back-end, thus by definition should be provided by simulator developers. Grammar of TISL is similar to C++ declarations, specifying alias of data types, declaration of enumerations and trace

items. Each item is an abstraction of a series of internal state changes or information collections of simulator. These specifications are transformed into C++ `typedef`, `enum` and `class` using Xtext generator. As mentioned above, code generated by TISL generator should be put into `imports/` directory inside given C++ code.

All the specification language make use of Xtext IDE support and provide content assistant at the right time. For example, `<target-directory>` introduced in next sub-section can be completed with content assistant triggered by `ctrl`+`space` and let user choose desired directory in a pop-up window.

8.2 Grammar

The BNF-like grammar of TISL can be expressed as following:

⟨*item-specification*⟩ ::= '`generate`' ⟨*target-directory*⟩ ⟨*definition*⟩*

⟨*definition*⟩ ::= ⟨*type-definition*⟩
| ⟨*enum-definition*⟩
| ⟨*item-definition*⟩

⟨*type-definition*⟩ ::= '`type`' ⟨*type-name*⟩ ⟨*real-type*⟩

⟨*real-type*⟩ ::= ⟨*built-in-type*⟩
| ⟨*custom-type-name*⟩

⟨*built-in-type*⟩ ::= '`int`'
| '`int8`'
| '`int16`'
| '`int32`'
| '`int64`'
| '`uint`'
| '`uint8`'
| '`uint16`'
| '`uint32`'
| '`uint64`'
| '`bool`'
| '`float`'
| '`double`'

⟨*enum-definition*⟩ ::= '`enum`' ⟨*enum-name*⟩
'`{`' ⟨*enum-literal*⟩ ('`,`' ⟨*enum-literal*⟩)* '`}`'

⟨*item-definition*⟩ ::= '`item`' ⟨*item-name*⟩ ('`extend`' ⟨*item-name*⟩)?
'`{`' ⟨*attribute-definition*⟩* '`}`'

⟨*attribute-definition*⟩ ::= ⟨*type-name*⟩ ⟨*attribute-name*⟩ ('`[`' ⟨*array-size*⟩ '`]`')?

TISL by default supports some built-in data types, which maps to C++ built-in simple types. Other than that, some custom data types can also be defined by using keyword `type` and providing information of custom type name and string of real type. The built-in types and custom types can then be used in item declarations.

Some tokens in the grammar are explained below:

- `<target-directory>` tells TRAP where to put the generated C++ code. It must be an existing directory pointing to the `imports/` sub-directory in the pre-written code.
- `<type-definition>` declares a C++ type and its alias. `<real-type>` is a built-in type or a string representing a valid C++ type while `<type-name>` is an alias that can be used in other specifications.
- `<enum-definition>` is translated into C++ `enum` directly, so all the `<enum-literal>` will stay in the same `namespace`. Thus, each single `<enum-literal>` must be unique.
- `<item-definition>` contains a list of attributes. `<attribute-definition>` is a pair consisting of a built-in type name or `<type-name>` defined previously and its own name `<attribute-name>`. Also an attribute can be an array of elements and the size is specified by `<array-size>`.

8.3 Example

Suppose there is a chip designed for controlling train system and there is a simulator that simulates this hardware. During the door is closing, the light above the door will blink to warn passengers. Suppose the light has different colors and may blink with different rate for some time. Then the information item of "the light of some color blinks for some time with rate" may be abstracted as following:

```
type IntType "uint32_t"
enum Color {
  RED, YELLOW, GREEN
}
item LightBlink {
  Color color;
  Int rate;
  Int duration;
}
```

When the program built upon the simulator needs to serialize the information of light blinking, it should create an instance of generated C++ class defined by "`item LightBlink`", then set value of its attributes and call the TRAP emission interface.

9 Logical Clock Specification Language (LCSL)

9.1 Overview

As a notion taken from CCSL, *logical clock* is the basic block for expressing system properties. A *clock* is one kind of basic "event" happening during the abstract time line. Each clock may *tick* as many times during the time line. Each clock tick has mainly two attributes: timestamp and name of the clock. All of the clock ticks share the same physical real time clock, thus the timestamp represents the presence order of ticks in the trace stream.

In TRAP, logical clocks are defined by the clock specification, which provides a higher layer of abstraction than the item specification because it is isolated with the low-level detail of simulator and represents more on logical level. A clock specification defines basic items of behavior of the hardware component or software that will run upon the simulator.

9.2 Grammar

BNF-like grammar of LCSL can be expressed as below:

⟨*clock-specification*⟩ ::= '`scope`' ⟨*scope*⟩* '`clock`' ⟨*clock*⟩*

Compared with other languages, grammar LCSL is rather simple. However it is important because scopes and clocks defined inside LCSL will be used by both STML and TPSL. Logical clocks are declared using keyword `clock` followed by a list of names, and keyword `scope` declares a list of phases. During the execution of simulator, HW/SW system may have different phases. Inside different phases, there may be different system properties to verify. In TRAP, *phase* are called *scope* that is defined in LCSL. Scopes may overlap with each other, so nested scopes can be expressed simply by overlapping two scopes with different ranges.

9.3 Example

Consider the train control system example in TISL section. During its execution, the train may have several phases: *speed_up*, *uniform_motion*, *slow_down* etc. Also, there might be many actions like *door_open*, *door_close*, *light_blink_rapidly* and *light_blink_slowly*. It can be declared using LCSL shown below:

```
scope speed_up uniform_motion slow_down
clock train_start train_stop door_open door_close light_blink
```

10 Simulation Trace Mapping Language (STML)

10.1 Overview

Since item specification and clock specification provides two different layer of abstraction, there must be a specification defining how to convert trace items into clock ticks, i.e. a mapping language. In this platform, STML builds the bridge between items and ticks. This mapping language defines how the items are mapped into clocks in a pattern matching manner. It takes as input two parameters, on one hand the TISL instance file, on the other hand LCSL instance. STML takes another parameter as output, which defines the place where the platform put the generated C++ codes.

Since logical clocks represents system events in a larger scale, several trace items may be mapped into single logical clock. Thus the timestamp may be scaled so that trace items of different timestamp can map to one tick. Here a notion of *period* is introduced to describe the maximum interval of trace items timestamp (i.e. real system clock) that will be considered as one timestamp in logical clock side. All the timestamp within one period is considered to be simultaneous while mapping. This design reflects the fact that trace item models lower level properties than logical clocks.

An STML instance consists of a header importing the item specification and the clock specification, then global variable definitions, then a set of rules, each one introduced by the keyword `for` followed by a item name defined inside TISL instance that defines the rule for that trace item. For each final item (i.e. not inherited by other items) in the trace model, there must be a corresponding mapping rule defined for it inside mapping specification, so that the mapping is guaranteed to be fully defined.

For each matched pattern, the mapping consists in transformation the input items into one of the three possibilities:

- simply ignore the pattern because it is irrelevant to the properties considered in this analysis. It is introduced by keyword `ignore` meaning no output clock ticks in this case.
- enter or leave a scope defined in imported LCSL instance with keyword `enter` or `leave`.
- emit a list of clock ticks introduced by keyword `emit`.

The language also has constructions related to the trace model. Given the input item specification, it makes it possible to:

- specifify conditional action, with a boolean condition and two branches. The condition must bear on the values of the attribute of items, or on the global variables. Each branch can be a basic action, or another nested conditional action.
- check for a sequence of items that finally construct a pattern. This sequence of items is specified using a construction similar to switch instruction in C++ programming language, introduced by keyword `when` followed by candidate `case`, with a mandatory `else` clause in case the sequences of items specified are not matched. This construction builds a pattern tree used by the pattern matching algorithm described in section 14.
- assign some attribute value of an item to a global variable. This feature makes it possible to considerably reduce trace file sizes. Indeed in many cases the trace items contain state information that is duplicated over and over again in the trace, whereas the verification is only interested in state changes. Global variables in the language make it possible to only emit clock ticks when some particular global values have changed, and the conditional makes it possible to quantify that change. For example, it is possible to express the case "emit a clock tick if the train speed has increased by more than 10% compared to last time considered".

By defining the mapping rule for each trace item, any trace items stream can be mapped into clock ticks to generate a new trace. Since the mapping process may ignore some data, or reduce repeated state to state changes only, and reduce multiple items into a single clock tick, the size of trace stream can be remarkably reduced when studying a given set of properties. After mapping, the engineer is able to focus on the clocks and scopes that provide much higher abstraction for reasoning on properties.

10.2 Grammar

BNF-like grammar of STML can be expressed as follows:

⟨*mapping-specification*⟩ ::= '`mapping`' ⟨*item-specication-file*⟩
'`to`' ⟨*clock-specification-file*⟩
'`generates`' ⟨*target-directory*⟩
'`from`' ⟨*source-directory*⟩ '`;`'
⟨*global-variable-declaration*⟩* ⟨*rule*⟩+

⟨*global-variable-declaration*⟩ ::= ⟨*type*⟩ ⟨*global-variable*⟩
'`=`' ⟨*global-variable-initial-value*⟩

⟨*global-variable-initial-value*⟩ ::= ⟨*constant*⟩
| ⟨*global-variable-initializer*⟩ ('`,`' ⟨*global-variable-initializer*⟩)*)

⟨*global-variable-initializer*⟩ ::= ⟨*item*⟩ '`.`' ⟨*attribute*⟩

⟨*rule*⟩ ::= '`for`' ⟨*item*⟩ ⟨*variable*⟩ '`{`' ⟨*statement*⟩ '`}`'

⟨*statement*⟩ ::= ⟨*simple-stmt*⟩
| ⟨*sequence*⟩

⟨*simple-stmt*⟩ ::= '`ignore`' ⟨*global-variable-update*⟩*
| ⟨*emission*⟩ ⟨*global-variable-update*⟩*
| ⟨*conditional-stmt*⟩

⟨*global-variable-update*⟩ ::= '`update`' ⟨*global-variable*⟩
'`=`' ⟨*variable*⟩ '`.`' ⟨*attribute*⟩

⟨*emission*⟩ ::= '`emit`' ⟨*clock*⟩
| '`enter`' ⟨*scope*⟩
| '`leave`' ⟨*scope*⟩

⟨*conditional-stmt*⟩ ::= '`if`' '`(`' ⟨*boolean-expr*⟩ '`)`' '`{`' ⟨*simple-stmt*⟩ '`}`'
'`else`' '`{`' ⟨*simple-stmt*⟩ '`}`'

⟨*boolean-expr*⟩ ::= ⟨*boolean-expr*⟩ '`||`' ⟨*boolean-expr*⟩
| ⟨*boolean-expr*⟩ '`&&`' ⟨*boolean-expr*⟩
| '`(`' ⟨*boolean-expr*⟩ '`)`'
| '`!`' ⟨*boolean-expr*⟩
| ⟨*comparison*⟩

⟨*comparison*⟩ ::= ⟨*arithmetic-expr*⟩ ⟨*compare-operator*⟩ ⟨*arithmetic-expr*⟩

⟨*arithmetic-expr*⟩ ::= ⟨*arithmetic-expr*⟩ ⟨*arithmetic-operator*⟩ ⟨*arithmetic-expr*⟩
| '`-`' ⟨*arithmetic-expr*⟩
| '`~`' ⟨*arithmetic-expr*⟩
| '`(`' ⟨*arithmetic-expr*⟩ '`)`'
| ⟨*primary*⟩

⟨*compare-operator*⟩ ::= '`<`'
| '`<=`'
| '`==`'
| '`!=`'
| '`>=`'
| '`>`'

⟨*arithmetic-operator*⟩ ::= '`+`'
| '`-`'
| '`*`'
| '`/`'
| '`%`'
| '`&`'
| '`|`'
| '`^`'

| '«'
| '»'

⟨*primary*⟩ ::= ⟨*integer-constant*⟩
| ⟨*boolean-constant*⟩
| ⟨*enum-literal*⟩
| ⟨*global-variable*⟩
| ⟨*variable*⟩ '.' ⟨*attribute*⟩

⟨*sequence*⟩ ::= '`when`' ⟨*pattern*⟩* '`else`' ⟨*simple-stmt*⟩

⟨*pattern*⟩ ::= '`either`' ⟨*item*⟩ ⟨*variable*⟩ '`:`' ⟨*statement*⟩

As shown in the grammar, global variables are introduced by `<global-variable-declaration>` above the definition of mapping rules. Global variables can be initialized by a constant, or a list of initializers. Each initializer consists of a pair of trace item and its attribute. During the simulation, global variables will be initialized by the first trace item belonging to the initializer list when it shows up. This only work for the first time. Later, the update of global variables are specified by `<global-variable-update>` via assigning attribute value to the global variable. Update field is designed to appear after `<emission>` or `<ignore>` because global variables follows read-then-update pattern.

The element `<pattern>` in the grammar reflects the concept of `pattern matching` mentioned above. It can be nested to build several patterns. A pattern is a tree of trace items, with its root being the start point for matching. Any path of the tree (i.e. a sequence of items) represents a case that once matched, `<simple-stmt>` will be taken into account. This filters out patterns that is not interesting for analysis.

Also inside STML, `<variable>` has its *visible scope*. Notion *scope* here is different from LCSL, here it denotes the life time of a variable. Local variables inside STML are introduced by `<rule>` or `<pattern>`, and referred to by their identifier inside `<expressions>`. A local variable introduced by any grammar element can be seen inside the children element, but not sibling elements.

STML supports various ways to express arithmetic expressions and values. All the arithmetic operators in C++ programming languages is supported and the operator precedence is exactly the same as C++. Also, apart from decimal constants, different representation of integer constants including hex, octal, binary are supported by specifying `0x`, `0`, `0b` in front of the number respectively.

Some other tokens are described below:

- `<item-specification-file>` and `<clock-specification-file>` represents location of instance files of TISL and LCSL respectively.
- `<target-directory>` indicates the location to put generated code.
- `<source-directory>` is the directory of code prepared by TRAP including C++ classes generated by TISL.
- `<item>` is id of trace item defined inside event specification.
- `<clock>` and `<scope>` are valid names of clocks and scopes declared in the clock specification.

10.3 IDE Support

TRAP contains a validator that will detect any errors on-the-fly and give error marks when the user stops editing. Any mistake in the events, clocks, scopes and attributes, will be spotted. Also, the platform is loading the event model and clock specification right after the user defines them. If the imported models cannot be loaded properly, an error message will be given to the user about it. Every time the user changes the location of the model, the platform will reload it automatically.

An important feature is type checking. For example, comparison can only happen when left and right operands have the same type, and operands of boolean operators must both return boolean values. In STML, when an expression is taken into account, type checking works as follows:

- If both operands (sometimes one operand) are defined with built-in data types introduced in TISL section, type checking is triggered. In this case, operands must have the same data type as operator requires, otherwise an error is signaled. If two data types are in the same category but have different types, for example one `int16` variable is compared with one `int32` variable, a warning instead of an error is given.
- If any one operand has a custom type, type checking mechanism works in another way. Since a custom type is defined with string, STML validator does not know what the real type is. In this case, the type string is compared and if they do not match, a warning is given.

STML editor also contains some other features that makes development of instances easier:

- Event template. After the user specifies the imported event model, the platform will parse the model and extract all the final events, so the platform will provide a list of final events defined in event specification with an "`ignore`" as the default behavior.
- Name completion. The name of events, clocks and scopes can be automatically completed according to the event model and clock model.
- Attribute completion. Since STML allows comparison between values inside boolean conditions, it is possible to access the attributes of events. After typing '.', the code assistant will give a list of valid attributes of that event.
- Pop-up window for directory selection. Same as TISL.

10.4 Example

Consider the blinking light inside train controlling system. Suppose if the light blinks in green or yellow color, it is ignored; otherwise it is in red color, `light_blink_rapidly` and `light_blink_slowly` will be emitted according to the rate. This example can be expressed by following:

```
for LightBlink b {
  if (b.color == RED) {
    if (b.rate <= 0.5) {
      emit light_blink_rapidly
    }
    else {
      emit light_blink_slowly
    }
```

```
  }
  else {
    ignore
  }
}
```

11 Trace Property Specification Language (TPSL)

11.1 Overview

After mapping, the original trace has been transformed into clock ticks. Now the clocks and scopes defined inside clock specification can be used to specify the properties of the HW/SW system user wants to verify. Considering the fact that most commonly used patterns are causality, universality and absence [13], pattern system in TRAP was built covering these ones and others commonly used in the embedded system development. In order to make it easy for engineers, TPSL was built as an English-like language to express the patterns with the Xtext framework. This makes it possible for user to define properties without needing to know CCSL algebra behind TPSL.

The TPSL specification takes the clock specification as input, imports all the clocks and scopes defined inside it, and at last generates corresponding code for verifier. A valid TPSL file consists of three parts:

1. Constant declaration. Constants are defined at the beginning of TPSL files and then used inside the rules. Currently only integers with arithmetic operations including addition, subtraction, multiplication and division are supported.

2. Clock declaration. To make it clear, clocks defined inside clock specification are called *basic clocks*, while those defined in this section are called *virtual clocks*. Basic clocks is acquired after mapping and virtual clocks is a composition of other clocks using logical clock algebra operations like union, intersection etc. Both base clocks and virtual clocks can be used by the rules defined in the same TPSL file.

3. Rule declaration. This section consists of a list of rules that each one expresses one property of the system and will be verified by TRAP verifier. Each rule will be checked inside its own scope defined inside clock specification. If a rule contains more than one scope, then the intersection of scopes are taken into account in such cases.

Each rule can be one of the supported patterns. Some patterns are taken as a subset from CCSL and then extended to provide more functionality, while some others are commonly known in embedded HW/SW development. Given a TPSL file, TRAP generates C++ code consisting the verifier by compiling them together. Each verifier executable file is gain from a specific TPSL instance file.

11.2 Grammar

BNF-like grammar of TPSL is shown as follows:

⟨*property-specification*⟩ ::= '`import`' ⟨*clock-specification*⟩
'`generates`' ⟨*target-directory*⟩
⟨*constant-declaration*⟩* ⟨*clock-declaration*⟩* ⟨*rule-declaration*⟩*

⟨*constant-declaration*⟩ ::= '`define`' ⟨*constant-assignment*⟩
('`,`' ⟨*constant-assignment*⟩)*

⟨*constant-assignment*⟩ ::= ⟨*constant-variable*⟩ '`=`' ⟨*arithmetic-expression*⟩

⟨*arithmetic-expression*⟩ ::= ⟨*constant*⟩ ⟨*arithmetic-operator*⟩ ⟨*constant*⟩
| '`(`' ⟨*arithmetic-expression*⟩ '`)`'

⟨*constant*⟩ ::= ⟨*constant-variable*⟩
| ⟨*constant-value*⟩

⟨*arithmetic-operator*⟩ ::= '`+`'
| '`-`'
| '`*`'
| '`/`'

⟨*clock-declaration*⟩ ::= '`clock`' ⟨*clock-assignment*⟩ ('`,`' ⟨*clock-assignment*⟩)*

⟨*clock-assignment*⟩ ::= ⟨*variable*⟩ '`=`' ⟨*clock-expression*⟩

⟨*clock-expression*⟩ ::= ⟨*clock-union*⟩
| ⟨*clock-sub*⟩
| ⟨*clock-intersect*⟩

⟨*clock-union*⟩ ::= ⟨*clock-union*⟩ '`union`' ⟨*clock*⟩
| ⟨*clock*⟩

⟨*clock-sub*⟩ ::= ⟨*clock*⟩ '`by`' ⟨*constant*⟩

⟨*clock-intersect*⟩ ::= ⟨*clock-intersect*⟩ '`intersect`' ⟨*clock*⟩
| ⟨*clock*⟩

⟨*rule-declaration*⟩ ::= ⟨*scope-declaration*⟩ ⟨*rule*⟩

⟨*scope-declaration*⟩ ::= ⟨*global-scope*⟩
| ⟨*inside-scope*⟩
| ⟨*outside-scope*⟩

⟨*global-scope*⟩ ::= '`always`'

⟨*inside-scope*⟩ ::= '`inside`' ⟨*scope*⟩

⟨*outside-scope*⟩ ::= '`outside`' ⟨*scope*⟩

⟨*rule*⟩ ::= ⟨*alternates-rule*⟩
| ⟨*causes-rule*⟩
| ⟨*each-causes-rule*⟩
| ⟨*absent-rule*⟩
| ⟨*jitter-rule*⟩
| ⟨*throughput-rule*⟩

| ⟨*burstiness-rule*⟩
| ⟨*period-rule*⟩

⟨*alternates-rule*⟩ ::= ⟨*clock*⟩ '`alternates`' ⟨*clock*⟩

⟨*causes-rule*⟩ ::= ⟨*before-expression*⟩ '`causes`' ('`!`')? ⟨*clock*⟩
('`unless`' ⟨*unless-statement*⟩)?
('`if`' ⟨*if-statement*⟩)?
('`satisfies`' ⟨*satisfies-statement*⟩)?

⟨*before-expression*⟩ ::= ⟨*clock-sequence*⟩ '`before`' ⟨*before-expression*⟩
| ⟨*clock-sequence*⟩

⟨*clock-sequence*⟩ ::= ⟨*clock*⟩ '`<>`' ⟨*clock-sequence*⟩
| ⟨*clock*⟩

⟨*unless-statement*⟩ ::= ⟨*inside-statement*⟩ '`and`' ⟨*unless-statement*⟩
| ⟨*inside-statement*⟩

⟨*inside-statement*⟩ ::= ⟨*clock*⟩ '`inside`' ⟨*clock*⟩ '`to`' ⟨*clock*⟩

⟨*if-statement*⟩ ::= ⟨*clock-count-statement*⟩ '`and`' ⟨*if-statement*⟩
| ⟨*clock-count-statement*⟩

⟨*clock-count-statement*⟩ ::= ('`global`')? '`count`' ⟨*clock*⟩ ('`+`' | '`-`') ⟨*clock*⟩ ⟨*comparator*⟩ ⟨*constant*⟩

⟨*comparator*⟩ ::= '`<`'
| '`<=`'
| '`==`'
| '`!=`'
| '`>=`'
| '`>`'

⟨*satisfies-statement*⟩ ::= ⟨*satisfies-statement*⟩ '`and`' ⟨*satisfies-statement*⟩
| ⟨*clock-delay-statement*⟩
| ⟨*clock-count-statement*⟩
| ⟨*clock-absent-statement*⟩

⟨*clock-delay-statement*⟩ ::= '`delay`' ⟨*clock-timestamp-expression*⟩
'`-`' ⟨*clock-timestamp-expression*⟩ ⟨*comparator*⟩ ⟨*constant*⟩

⟨*clock-timestamp-expression*⟩ ::= ⟨*clock*⟩ '`.`' ('`first`' | '`last`')

⟨*clock-absent-statement*⟩ ::= ⟨*clock*⟩ '`absent`' '`from`' ⟨*clock*⟩ '`to`' ⟨*clock*⟩

⟨*each-causes-rule*⟩ ::= ⟨*before-expression*⟩ '`each`' '`causes`' ⟨*clock*⟩
⟨*unless-statement*⟩ ⟨*satisfies-statement*⟩

⟨*absent-rule*⟩ ::= ⟨*clock-absent-statement*⟩

⟨*jitter-rule*⟩ ::= '`jitter`' '`(`'
⟨*clock*⟩ '`,`' ⟨*constant*⟩ '`,`' ⟨*constant*⟩ '`,`' ⟨*constant*⟩
'`)`'

⟨*throughput-rule*⟩ ::= '`throughput`' '`(`'
⟨*clock*⟩ '`,`' ⟨*constant*⟩ '`,`' ⟨*constant*⟩
'`)`'

⟨*burstiness-rule*⟩ ::= '`burstiness`' '`(`'
⟨*clock*⟩ '`,`' ⟨*constant*⟩ '`,`' ⟨*constant*⟩ '`,`' ⟨*constant*⟩
'`)`'

⟨*period-rule*⟩ ::= '`period (`' ⟨*clock*⟩ '`,`' ⟨*constant*⟩ '`,`' ⟨*constant*⟩ '`)`'

Some other concepts are explained below:

- `<clock-specification>` is the path of clock specification file.
- `<target-directory>` is the directory where TPSL will put the generated C++ code for verifier.
- `<constant-value>` is an integer value.
- `<clock>` is a valid clock defined inside TPSL file, or inside clock specification imported by it.
- `<scope>` is a valid scope defined in the clock specification imported by TPSL.
- `<variable>` is a valid variable name consisting of characters and underscores.

In TPSL, relationships between two or more clocks can be mainly divided into three categories: clock *expressions*, clock *relations* and clock *properties*.

- A clock expression takes two or more clocks as arguments and defines a new virtual clock from given parameters. In the grammar, virtual clocks are introduced by `<clock-declaration>` and can be used everywhere `<clock>` is needed. They tick every time the expression is satisfied, with the timestamp of last input clock who triggered it.
- A clock relation takes two or more clocks as input and defines the constraint of given clock(s). Clock relations are defined by `<rule>` and they will be verified by the verifier.
- A clock property takes exact one clock and expresses the property of it. Similar to clock relations, they are defined by `<rule>` and verified by the verifier.

Since TPSL was built as a layer of abstraction upon CCSL, the user may be unaware of the theory behind TPSL. TPSL are designed to be flexible and extendable to provide more possibilities in the future, with consistent behavior among different composition of use cases. In next subsections, clock expressions, relations and properties are explained in detail.

11.3 Clock Expression

Current three kinds of expressions are supported to introduce manually-defined virtual clocks:

1. Subclocking expression specified by `<clock-sub>`. It takes a clock A and a constant N as input, and generates a new clock which ticks every N ticks of the input clock A. It is designed to express sub-clocks relying on others.

2. Union expression introduced by `<clock-union>`. It takes a sequence of clocks concatenated with keyword `union`. The output clock ticks when any one of input clocks ticks.

3. Intersection expression specified by `<clock-intersect>`. Similar to `<clock-union>`, it takes a sequence of clocks as input concatenated with keyword `intersect`, generating an output clock that ticks when all the input clocks tick simultaneously.

A special case is `<before-expression>`. Unlike other clock ones, `<before-expression>` can only be used inside `<causes-rule>` or `<each-causes-rule>`. It can not be used to define a virtual clock, however it introduces an implicit virtual tick used internally by TRAP.

Clock expressions provides more power of abstraction and makes it easier to express complicated properties while keeping the syntax neat and clean. TRAP handles nested expressions in a recursive manner, i.e. the expression with highest priority is handled first and acts like one virtual tick to the outside nested expressions. So each expression and relations only cares about its own input clock and ignores the original raw clocks.

11.4 Rule

11.4.1 Overview

In TPSL, each rule defines a system property, thus the verification ability of TRAP is mainly decided by the power of TPSL. A rule consists of two parts: scope declaration and rule body. In TRAP tool, each type of rule is mapped into a specific template of automaton that will be explained in section 14. Every time a rule gets its first input clock inside the corresponding scope, the automaton is activated. During its state transitions, input ticks are fed into the automaton and stored inside its local history. If any error occurs, this automaton fails and is deactivated. If nothing wrong happens this rule is passed for its input and the automaton is reset to the initial state. Scopes and rule body definition will be explained in the following sub-sections.

11.4.2 Rule Scope

As mentioned above, LCSL declares both logical clocks and scopes. In TPSL, each rule has its own scope defined by `<scope-declaration>`. It indicates the range in which the rule should be verified. For any specific rule, all the clock ticks outside its scope are simply ignored by it. The scope are changed by keyword `enter` and `leave` in STML, so the `current` scopes is changed respectively during the execution. A rule is called *active* inside its scope. In TPSL the scope of a rule can be defined by composition of three keywords:

1. Keyword `always` indicates a global scope, also understood as “no specific scope”. Rules inside global scope is active all the time.

2. Keyword `inside` indicates a scope that the rule is active inside it.

3. Keyword `outside` indicates a scope that the rule is active outside it.

If more than one scopes are given, the intersection of them will be taken into account. In this case, `always` actually has no effect. If union of scopes is needed, i.e. "either inside A or inside B", the user has to define another scope logically containing both and use it instead of A or B.

After the scope goes the body of the rule indicating the property of a single clock, or clock relation among several clocks. They will be introduced one by one in the following sub-sections.

11.4.3 Alternates Relation

Keyword `alternates` specifies the relation between exactly two clocks that they tick one after another repeatedly. Neither of the two can tick twice continuously before another one ticks. Also the counts of two clocks must be equal.

11.4.4 Absent Relation

Absence is an important and commonly-used property in embedded HW/SW development. It indicates that some event can never happen between two other events. In TPSL it is defined using keyword `absent from` with form "*target* `absent from` *start* `to` *end*", with *target*, *start* and *end* being valid clock names. By definition, after clock *start* ticks, if clock *target* ticks before another clock *end*, there is a violation.

11.4.5 Causes Relation

Causality is expressed by keyword `causes` and is associated with two clocks: *trigger* and *target*. It indicates the causality between the two. Note that *trigger* here can be either a clock or a `<before-expression>`, given that before expression introduces an implicit virtual clock as mentioned above.

`<before-expression>` takes one or more `<clock-sequence>` as input and works in a nested manner. For each "*A* `before` *B*" part, in which A and B are either clock or `<clock-sequence>`, any tick of B will trigger the virtual tick if and only if$A$ ticks at least once before B. If keyword `before` is replaced with `each before`, each pair of A followed by B will trigger a virtual tick no matter how far the B is after A. When B here is a `<clock-sequence>`, it emits a virtual clock tick when all the input clocks have already ticked, without considering the order of ticks.

Causes clause works in two modes depending on the existence of question mark after `causes` keyword:

- If there is no "!", `<causes-rule>` represents a sufficient condition. The basic behavior is that if *trigger* ticks at some time, there must be a tick of *target* before the end of simulation, otherwise an error is emitted. Note that it is valid to have several *trigger* ticks before a single *target*, or *target* may tick individually in spite of *target*.
- If "!" is provided, `<causes-rule>` becomes both sufficient and necessary. In this case, apart from the normal behavior, for any tick of *target*, there must be at least one *trigger* tick before it.

In addition to the basic grammar, three extra control clauses is added to provide more control over the default behavior. They are optional, but if not omitted, they must appear in order. Listed below:

- `<unless-statement>` starts with keyword `unless`. It is a chain of `<inside-statement>` concatenated by *and*. It specifies the situation that if a clock ticks within some range, ticks in current history is thrown away and automaton goes to initial state, waiting for *trigger* to restart. `<unless-statement>` has highest priority over next control statement.

- Keyword `if` specifies `<if-statement>`. It defines that once the condition has been satisfied, the next input tick must be *target*, otherwise an error is emitted. The condition is an expression related with the count of clocks. It has the form "`if count` *ClockA - ClockB* $<$*CONSTANT*", where *A* and *B* stands for clocks and *CONSTANT* stands for a constant variable or value. Notion *count* here means the count of ticks in the local history of automaton. If keyword `global` is given before `count`, the global count of *A* and *B* are calculated instead of the local count.
- Keyword `satisfies` leads a `<satisfies-statement>`. It defines the extra constraint of causes clause that will be checked when the automaton reaches the end point. `Satisfies` statement can check the count, delay or absence of clocks:
  - Count checking has the same form as used in `<if-statement>` to compare the tick count difference of two different clocks.
  - Delay checking takes the first or last timestamp of two clocks and compare their difference with given constant.
  - Absence checking works similarly with `<unless-statement>` and checks the target clock does not tick between given other two.

With the extension, `causes` relation in TPSL is more flexible and powerful to handle complicated cases.

11.4.6 Each Causes Relation

Grammar of `<each-causes-rule>` is similar to `<causes-rule>` and the basic ehavior of `<each-causes-rule>` is similar to `<before-expression>` with "`each`" keyword. It takes a `<clock-sequence>` or a single `<clock>` and makes sure that each time input clock ticks, there must be one target clock tick in the future. It supports `<unless-statement>` and `<satisfies-statement>` but not `<if-statement>`. Also the question mark in `<causes-rule>` cannot appear here.

11.4.7 Jitter Property

It is specified by `<jitter-rule>` and has the form "`jitter`(*A, S, P, N*)", where *A* is the clock to be checked, *S*, *P* and *N* are timestamp constants. Jitter is used to make sure clock *A* ticks regularly with a valid margin. Suppose there are pivot ticks standing in the time line, with exact same period *P* between any two of them. *A* is supposed to tick like the pivots, however it may tick sooner or later each time, with a valid margin *M* within which the property is not violated. Constant *S* is used to specify the start timestamp offset after the scope begins.

11.4.8 Period Property

Period property is specified by `<period-rule>` and has the form "`period`(*A, T, M*)" where *A* is a valid clock and *T*, *M* are constants standing for timestamp difference. It is similar to jitter rule but it does not have any pivot. Instead, each clock tick is compared with its previous tick. This rule means that the distance between any two ticks of clock *A* must be *T* time units with a valid margin of *M*.

11.4.9 Throughput Property

Throughput property is specified by `<throughput-rule>` with the form "`throughput`(*A, T, N*)". For clock *A*, throughput is defined as the count of ticks during time *T*. Throughput rule checks

during any time interval T, the count of ticks of clock A must be at least N. Note that the time interval here is left-close-right-open.

11.4.10 Burstiness Property

Burstiness property is specified by `<burstiness-rule>` and has the form "`burstiness`(A, T, N)". Burstiness of clock A is defined as that the count of ticks of clock A has a peak during some time T. This rule emits an error if in any time interval T the count of ticks is larger than given constant N.

11.5 Example

This section also takes the train controlling system as introduced in above sections. Suppose all the clocks needed are already defined by LCSL.

```
// Door cannot open twice, or close twice.
always door_open alternates door_close;

// Door can never open during the train is moving.
always door_open absent from train_start to train_stop;

// If the light blinks rapidly, the trains is going to brake
// within 1 second.
always light_blink_rapidly causes brake
  satisfies delay brake - light_blink_rapidly <= 1

// Each brake action causes slow_down.
always brake each causes slow_down;

// 30 seconds after the train started, the driver must send signal to
// control center stating he is not sleeping. The driver must do this
// every 15 seconds with margin of maximum 3 seconds.
inside train_running jitter(send_signal, 30, 15, 3);
```

12 Verifier

12.1 Overview

The verifier takes ticks as input a stream of clock and analyze it. As mentioned above, the verifier may work in either offline mode or online mode, meaning that the input stream can either be a generated trace file, or any other stream such as Unix named pipe. Whichever the mode, the verifier logically takes a tick stream and analyze it line by line. If any error is found during the verification process, the verifier emits an error about the corresponding rule and continues with other properties. When the trace stream reaches the end, the verifier will give a final summary about the verification.

The pipe mode works in a blocking manner due to the nature of named pipe in Linux. When simulator writes to and verifier reads from the same pipe, the simulation is possible to write only when the old contents in the pipe have been read. Otherwise, the simulator will be blocked and wait for the reading of verifier.

Since an error in property level might be caused by a long chain of causality, sometimes it is not so easy to find the bug directly from the error message. In order to make it easier for the user to identify the error place inside the simulation application, a GDB-like debugger is designed to provide more control over the verification, thus brings convenience to debugging the clock tick stream. Detailed information will be given in next sub-section.

12.2 User Interface

Verifier tool provides a command-line user interface. The basic command is "verifier [option]". Each option has long form and short form. Valid options are listed below:

- `-i, -ignore-invalid-clock`. Make verifier ignore invalid clock ticks during verification. If this option is not given, the verifier will emits an error every time an invalid tick occurs.
- `-d, -debug`. Specify debug mode to invoke the debugger. This mode provides more control and information during the verification.
- `-f, -file <file_name>`. Specify name of input trace file and make verifier work in offline mode.
- `-p, -pipe <file_name>`. Specify a opened name of pipe to communicate with the simulator. This option can not work with file mode or debug mode.
- `-D, -print-debug`. The verifier may print to standard output some information. This options allows it to provide more information.
- `-V, -print-debug`. This option makes verifier provide most verbose information.
- `-h, -help`. This option prints the help message.
- `-v, -version`. This option prints the version information of verifier.

12.3 Debugger

The debugger is invoked by providing `-debug` argument to the verifier program. In this case, the traces must be dumped into a trace file thus the verifier must work in offline model. The debugger contains a command line user interface that takes input from the user and act according to the command. The user may input just part of the commands, such as only the first character, to save some typing. Valid commands are listed below:

- `help`. This command gives the help information about other commands. Typing `help <command>` prints the help information for specific command.
- `run`. This command reset the internal state of verifier and re-analyze the trace file from the beginning, until a breakpoint set by the user or any error happens.
- `continue`. This command is similar to `run`, however it does not reset the verifier. Instead, it starts from the current point of verifier until a breakpoint or an error.
- `breakpoint`. This command manages the breakpoints. The debugger will pause if any breakpoint is reached. Each breakpoint can be a specific clock type, the line number of trace file, or the timestamp of clock tick. The breakpoint can be set with sub-command `clock`, `line` and `timestamp` respectively, with an second argument specifying the value. If this command is invoked without any arguments, a list of current breakpoints with indexes

will be printed. Also, sub-command `delete` with the index will remove the corresponding breakpoint from the list.

- `next`. For a debugger, it is mandatory to support step-by-step execution. Here the command `next` is introduced for analyze next several ticks. It accepts a second argument specifying the next N ticks to be processed. If no argument is given, it will process the next one tick.
- `info`. This command prints the global information of the verifier. Sub-command `clocks` prints a list of valid clocks that can then be used to set the breakpoint. `status` prints the current state of verifier, and `rules` print the rule text in original TPSL specification. Sub-command `history` with an rule id prints the ticks took by this rule before the rule finishes.

The verifier must be compiled every time the property specification is changed with newly generated code. However, since some of the complexity is transferred to compilation time, the verifier runs fast. With an trace file of 2 megabytes and 8 rules, the verifier finishes verification within 1 second.

13 Design Summary

We have introduced here the system architecture and usage of whole tool chain of TRAP. It consists of four specific languages implemented upon Xtext framework: 1) TISL abstracts the simulation detail into trace items; 2) LCSL provides basic block for system events; 3) STML provides the mapping between trace items and logical clocks; 4) TPSL expresses system properties to satisfy. For each part TRAP generates C++ code and they are loosely coupled with each other. At last, a verifier and debugger is provided to analyze the trace and find out the problem. These four parts construct TRAP tool chain and provides power, convenience and flexibility. The next section detail the implementations

The TRAP platform is implemented in C++ and Java programming language and runs on Linux operating system. As shown in figure 3, code generated by TISL are compiled together with simulator to generate a new simulator. The simulator will dynamically load the mapping library and call function `push_item()` to emit trace items. The mapping library converts trace items into logical clocks according to STML. Finally the generated trace stream are fed into corresponding verifier, which contains code generated by TPSL, to get analysis result.

From the implementation's perspective, the tool chain may be briefly divided into three parts: trace generation, property definition and verification process. These topics are detailed in the following sections.

14 Trace Generation

14.1 Architecture

UML representation for architecture of trace generation is shown in figure 4. It contains briefly two parts: simulation and mapping library. The gray rectangle represents the original simulator, i.e. the simulation back-end. It imports trace mapping classes provided by TRAP and trace items classes generated by TISL. This part of code will be compiled into a new simulator, which includes the trace emission facility provided by TRAP.

Inside the mapping facility, class `TraceMapper` provides function `push_item` that is the key function to emit trace. `TraceMapper` takes a list of `TraceWriter` and length of period as input. It

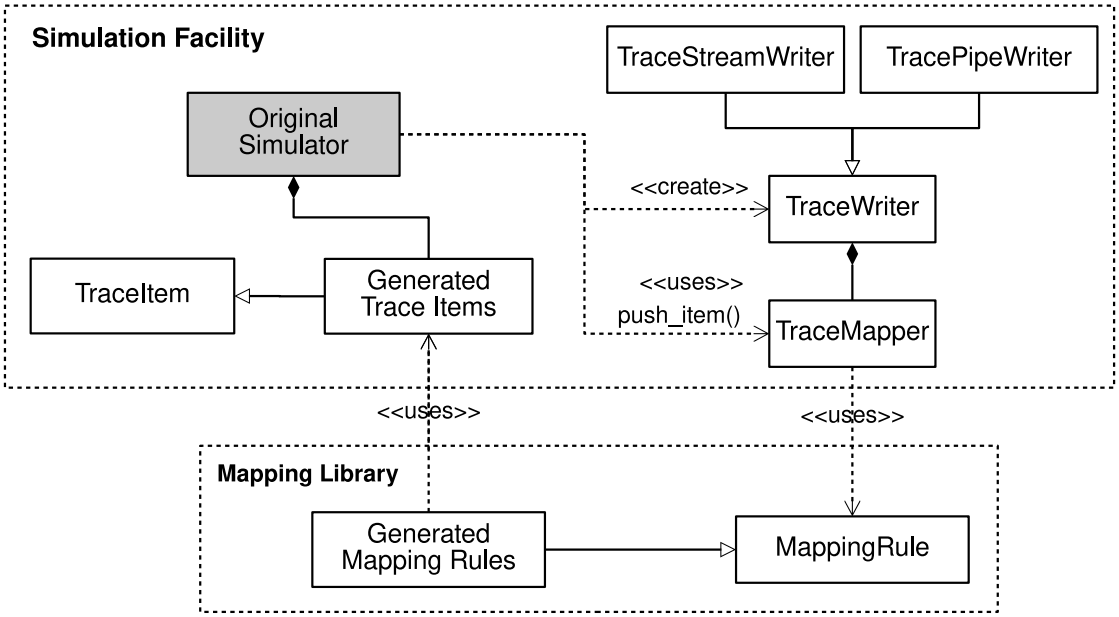

Figure 4: Trace generation architecture

processes input trace items and process them according to the algorithms introduced in following sections. Here notion of *period* is introduced in section 10.1 meaning a interval of timestamp in which trace items are treated as simultaneous.

`TraceMapper` keeps a list of `TraceWriter` so that it can write transformed clock tick stream to several places. `TraceWriter` is a virtual parent class for handling low-level output. Here it is inherited by `TraceStreamWriter` and `TracePipeWriter` that are responsible for dealing with detailed implementation to write into `std::ostream` or named pipe respectively. This design makes it possible for simulator to serialize trace stream to named pipe and dump it into trace files at the same time.

14.2 Trace Mapping

While generating the traces, the simulator should call interface function `push_item(TraceItem *)` defined inside class `TraceMapper`. This function takes a pointer of `TraceItem` type as input and handle the transformation till the end of simulation. For each input item, this function handles it according to the algorithm described in algorithm 1.

For each incoming trace item, TRAP checks if the timestamp is within the same period with the oldest tick (i.e. the first tick) in the item queue. If not, it will process the queue by three steps:

1. For each item inside the queue, try to find a rule with that item as the start. Since for each final trace item there must be a corresponding mapping rule, the `rule` will definitely be found.

2. For the rule, call FIND_MATCHING_ITEMS function that is a pattern-matching method to find the pattern for it. The algorithm will be shown in algorithmalgo:pattern-matching.

Algorithm 1 Trace mapping algorithm

```
1: item_queue ← list of items defined in TraceMapper
2: period ← user-defined period value
3: oldest_timestamp ← 0
4: procedure PUSH_ITEM(input_item)
5:     if e.timestamp − oldest_timestamp > period then
6:         while item_queue is not empty do
7:             rule ← find rule starts with item_queue.front
8:             matching_items ← MATCH_ITEMS
9:             call APPLY on matching_items
10:            erase items relevant to rule from item_queue
11:        end while
12:    else
13:        push input_item to the end of item_queue
14:        old_timestamp ← e.timestamp
15:    end if
16: end procedure
```

3. Call APPLY function with matching items. This function returns a string as generated logical clocks. Then, the matched items are removed from *item_queue* and the trace string is output to the right place.

The processing method runs until the first item in the queue is in another period. At the end of it, all trace items which are inside the same old period will be processed and removed.

14.3 Pattern Matching

For corresponding rule of each trace item, TRAP will try to find the pattern which is matched to the item and apply the pattern with matched item sequence. A pattern is implemented by internal class `Pattern` in a recursive manner. `Pattern` is a tree-like structure that holds a trace item as root and a list of `Pattern`. The patter matching and apply procedure is described in algorithm 2.

Algorithm 2 Pattern matching algorithm

```
1: matched_items ← empty queue
2: procedure MATCH_ITEMS(item_queue, pattern_list)
3:     is_match ← false
4:     for item in item_queue do
5:         for pattern in pattern_list do
6:             if item and pattern.item() has the same type then
7:                 append item to matched_items
8:                 MATCH_ITEMS(subsequence(item_queue, item),
9:                     pattern.next_pattern())
10:            end if
11:        end for
12:    end for
13: end procedure
```

In the algorithm, functoin *MATCH_ITEMS* finds matching items in a recursive manner. For each *item* in the *item_queue*, function *MATCH_ITEM* tries to find the pattern started with the same item type. If any matched item is found, it will first append the matched item to the queue *matched_items*, then call itself with rest items and patterns. After all, *matched_items* will store the items that match one pattern of the rule.

14.4 Summary

During the execution of simulator, each call to `push_item` pushes the given item into the queue and process it according to the comparison result between its timestamp and the oldest timestamp in the queue. Every time `apply` is called, all the logical clocks transformed from items within the same period will be written to file or pipe with the help of `TraceWriter`. After all the trace stream are generated continuously during the simulation. However, as mentioned above, TRAP does not handle infiniteness because the limit of memory usage of real computers. If the period is too large all the items will be stored in the queue that may cause the memory usage exceeding its limit.

15 Property Specification

15.1 Overview

In this section the relationship between TPSL and corresponding clock expression or relation in CCSL theory is introduced. Also the underlying automaton of each rule is drawn. TPSL was created with flexibility and availability in mind, by taking a subset of CCSL with extension of functionality and English-like grammar. Rules of jitter, distance, throughput and burstiness are special CPS properties so they cannot be expressed by CCSL.

Referring to automata, classic automata theory is built on the assumption that the alphabet is finite. However, in trace verification transition input may be unlimited, according to the external condition. Thus, Symbolic Finite Automata (STA) theory [16] is built to make automata more expressive, especially easier to express conditional transitions.

Upon SFA, Symbolic Transducer (ST) is built as an extension to it. In a ST transition, after reading an input symbol, an output is computed, and expressed as a function of the input. Many variants of ST has been proposed, and TRAP chooses Symbolic Finite Transducer [17] as the underlying automata inside verifier.

Since TPSL compiler generates both automata initializer and clock dispatcher, each automaton only accepts clock ticks it concerns about. Thus any unrelated clocks for an automaton will be ignored. The guard of each transition in automata is either related to input clock (represented by x), or a boolean value. Output of each transition is a function call updating corresponding values. Node of automata labeled by S is the initial state. Concentric circles is used to represent the ending state in SFT. Because the verification is a streaming process so that it transits to the initial state on every success, here the ending state represents error state. The automaton will signal an error and exit.

In the following sub-sections, representation of each clock expression or relation in CCSL and SFT automata will be given, to show the internal implementation of verifier.

15.2 Clock Expressions

As mentioned above, every time a clock expression *passed*, a corresponding virtual clock tick will be emitted. So in the clock expression automata, virtual ticks will be emitted when automata

transits back to the initial state. In the figures below, function `emit()` is used to represent the emission of virtual ticks.

15.2.1 Subclocking

Subclocking expression has the form "$B = A$ `by` N", in which A and B are clock names and N is a positive integer. In CCSL, subclocking denoted by symbol "$\subset$" is a relation between two clocks, meaning that each tick of one clock must be consistent with a tick of another clock. In TPSL it works as a clock expression and is used to introduce a new clock that has the relation "$B \subset A$", with exact count N. Corresponding automaton is shown in figure 5.

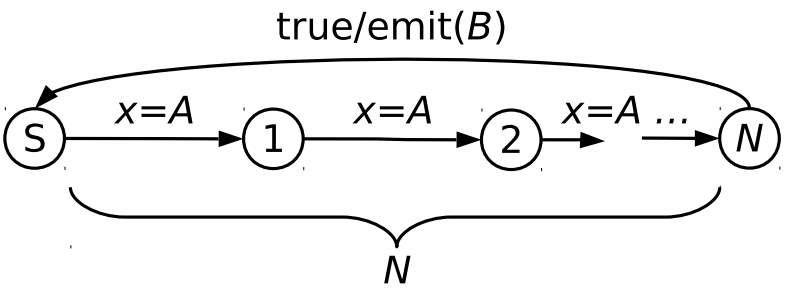

Figure 5: Subclock automaton

When in initial state, an tick of A will activate the automaton. After N ticks of A, a tick of B is emitted.

15.2.2 Union

In CCSL, union clock expression "$A + B$" ticks whenever A or B ticks. In TPSL, form "$C = A$ `union` B" is used to introduce clock C as union of A and B. In the syntax, `union` can be chained to union several clocks, but they are handled as if they are specified separately by introducing internal virtual clocks. Thus for each union expression, it can be treated as a chain of automata each dealing with two clocks.

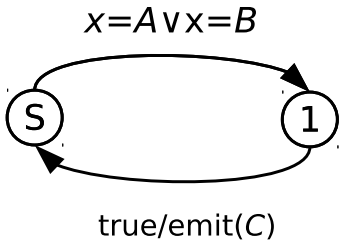

Figure 6: Union automaton

As shown in figure 6, automaton keep accepting either A or B. If either if provided, it transits to state 1, then immediately transits back to initial state, and emits a tick of clock C.

15.2.3 Intersection

In CCSL, expression "$A * B$" ticks whenever both A and B ticks. In TPSL, form "$C = A$ `intersect` B" is used to introduce clock C as the intersection of A and B. Similar to union expression, intersection expression can be chained, and is handled in the same way.

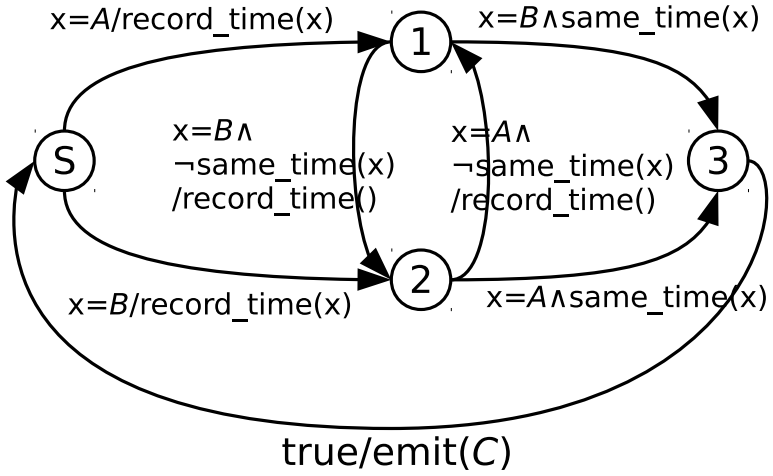

Figure 7: Intersection automaton

As shown in figure 7, if clock A is accepted in initial state, automaton transits to state 1 and update the timestamp. If clock B ticks when automaton is in state 1, it transits based on condition:

- If input x has the same timestamp as recorded, automaton transits to state 3 and emits clock C then transits back to initial state immediately.
- If input x has different timestamp from recorded, automaton transits to state 2 and update the recorded timestamp. Then automaton acts in the reverse way of state 1.

In initial state, if B ticks, automaton acts similarly to tick of clock A.

15.2.4 Before

Strictly speaking, before expression is not trivial because it can be used only in causes clauses. But it is an expression because it returns a new clock, even though internally used. In CCSL it is expressed by sampling expression $A \setminus B$. The expression ticks in coincidence with the tick of A immediately following a tick of B. In TPSL, a clock tick is emitted whenever B ticks after at least one tick of A.

Figure 8 shows the automaton for expression "A `before` B", which emits a internal virtual clock C. In initial state, the automaton is triggered by a tick of A and transits to state 1. Then, it continuously accepts A until B ticks. Afterwards, it transits to state 2, and then emits a tick of C with the same timestamp with B and transits back to initial state.

15.2.5 Each Before

Similar to before expression, each before expression is used only in causes clauses. In the semantic, it is only a "`each`" keyword in front of "`before`", but their automata are different. As an extension

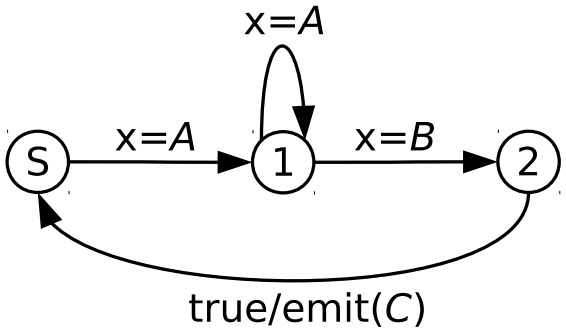

Figure 8: Before automaton

to before expression, there is no corresponding expression in CCSL. Form "A `each before` B" introduces an internal clock, named C here for convenience, can be described by automaton in figure 9.

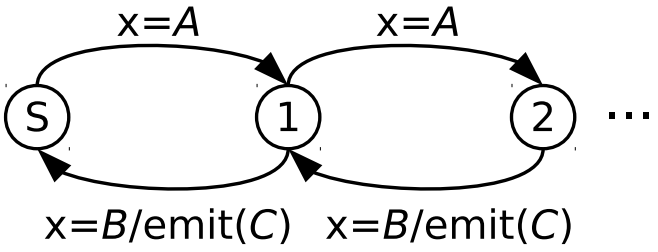

Figure 9: Each before automaton

As shown in figure 9, in any state including initial state, if A ticks, the automaton transits to next state; if B ticks, automaton transits one state back and emits a tick of C at the same time.

15.2.6 Clock Sequence

Clock sequence denoted by "<>" is only used together with before expression and can be understood as "unsorted before". Compared with clock intersection, clock sequence does not require two clocks to tick at the same time. Semantically, "A <> B" means "A `before` B or B `before` A", which can be written as "$A \setminus B + B \setminus A$".

Similar to before automaton, clock sequence is treated as composition of several automata which each deals with relationship between two clocks (i.e. A and B) and emits a virtual clock tick (named C), as shown in figure 10.

The automaton waits for tick of clock A or B in initial state. If either is given, automaton makes transition and waits for the other clock tick. Finally it transits to state 3, emits C and then transits back to initial state.

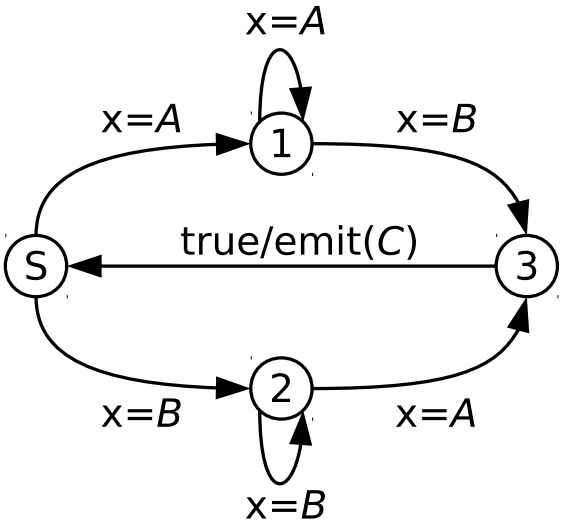

Figure 10: Clock sequence automaton

15.3 Clock Relations

15.3.1 Alternates

In CCSL, alternates can be expressed by form "$A \prec B \wedge B \prec A\hat{}1$". Symbol "$\prec$" means "strictly precedence", so that each tick of A appears is strictly before B. Symbol "$\hat{}$" means the next Nth tick, so that for each ith tick of B, $B[i]$ ticks strictly before $A[i{+}1]$. By composition of two expression, A and B must tick alternately.

Automaton of TPSL expression "A `alternates` B" is drawn in figure 11. For convenience, internal clock emitted by automaton is named C.

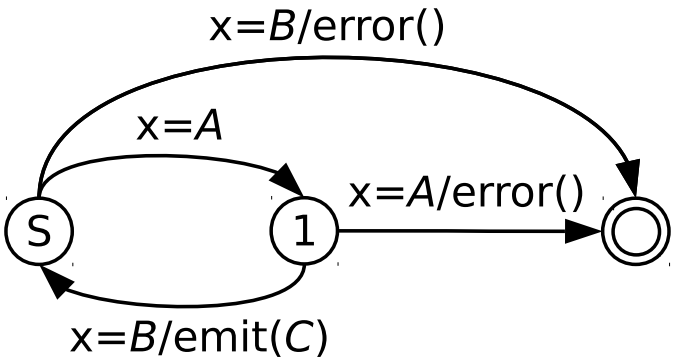

Figure 11: Alternates automaton

As shown in figure 11, automaton waits for clock A in initial state. If B presents, it signals an error. After an A, presence of B is desired, then the automaton transits back to initial state and emits a tick of C; otherwise, an error occurs and automaton goes to an end.

15.3.2 Absent

It is not that straightforward to express absence in CCSL, but the automaton representation is easy to understand. Automaton of "C `absent from` A `to` B" is shown in figure 12.

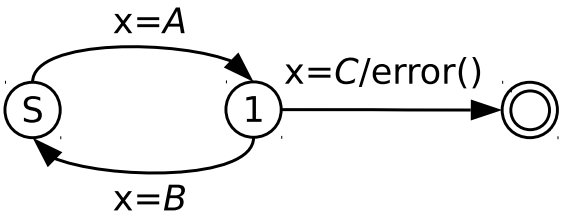

Figure 12: Absent automaton

In the initial state, automaton is activated and transits to state 1 when a A tick is accepted. Then, if any tick of C is presented, an error occurs. Otherwise if B clock is accepted, the automaton transits back to initial state and waits for another *Start*.

15.3.3 Causes

Causes clause by itself, works similar as before expression. But in CCSL, sampling is a clock expression while in TPSL causes represents a clock relation. Apart from that, some extra functions are added to describe how the automaton should behave. Here the automaton of "A `causes!` B `unless` ... `if` ... `satisfies` ..." is described in figure 13. This is the most complicated form of causes statement. Other simplified forms is just a simplification of this automaton.

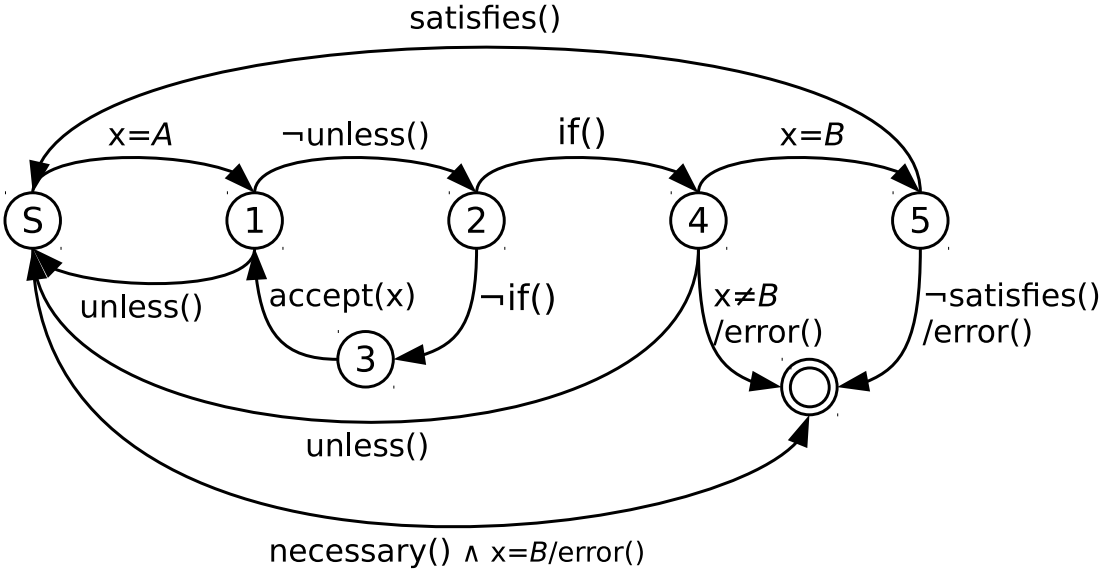

Figure 13: Causes automaton

Automaton starts with expectation for clock A. If A ticks, then it transits to state 1 and transits to state 2 or back to initial state according to the result of `unless` clause. In state 2,

it checks whether `if` clause is satisfied and transits to state 4 or to state 3 and waits for other clock ticks. State 4 means that the `if` statement is triggered, thus the *next* relevant tick must be of type clock B, otherwise there is a fatal error. If B shows up next, automaton goes to state 5 and checks `satisfies` clause. Additionally, if the clause is declared to have necessity (by symbol `!`) and clock B ticks in initial state, automaton transits directly to the final state and signal an error.

15.3.4 Each Causes

Each causes rule is expressed as a special form as causes rule, but its automaton significantly distinguishes from causes rule. Even though TRAP does not deal with infiniteness due to the limit of memory resources etc as mentioned above, automaton of each causes behaves logically like infinite.

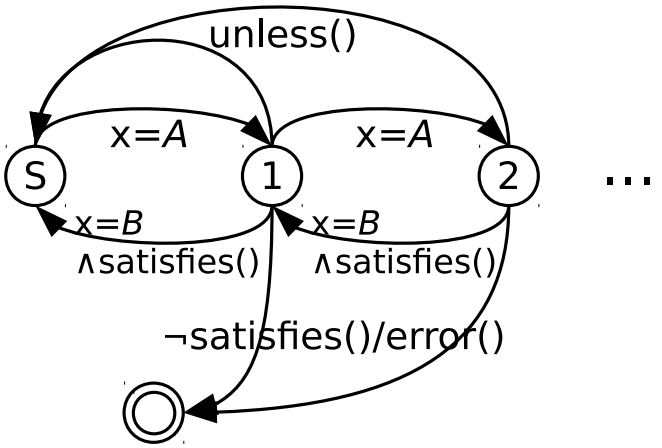

Figure 14: Each causes automaton

As shown in figure 14, each node of automaton behaves similarly. For each node, if clock A ticks, it transits to next one; if clock B ticks, it transits to state 3 and checks the result of satisfies clause. If it is satisfied, automaton transits to its previous state, otherwise it signals an error and dies. In any state other than the initial one, if unless clause is satisfied, the automaton is reset to the initial state.

15.4 Summary

Other rules, which are some common CPS properties, cannot be mapped to trivial automata. They provide same interface as automata, but works internally differently. Each automaton is implemented as a template class inherited from `Automaton`. During compilation, these automata is initiated by generated code. The code is then integrated into verifier to do the verification.

16 Verification Process

16.1 Overview

Verification is the last yet important process. It takes a stream of logical clock ticks and gives the verification result, together with a offline debugger. The verifier is implemented in C++,

consists of two parts: 1) the verification core, which is manually written, containing automaton templates for each rule described in section 15; 2) code generated by TPSL compiler to instantiate corresponding template and clock dispatcher.

When invoked, verifier main function setups environment based on command line options given by user, then invokes core functions. If it works in online mode, a named pipe is opened and clock ticks are taken from it; otherwise, a trace file is opened to provide clock ticks.

In this section, the overall architecture and explanation of each component is introduced in detail. Implementation of verifier will be given to show how TRAP deals with clock tick stream and gives the analysis result.

16.2 Architecture

The class architecture of verifier is shown in figure 15. In the figure, classes are represented as rectangle with solid border and white filling color. UML-like relations are used between classes.

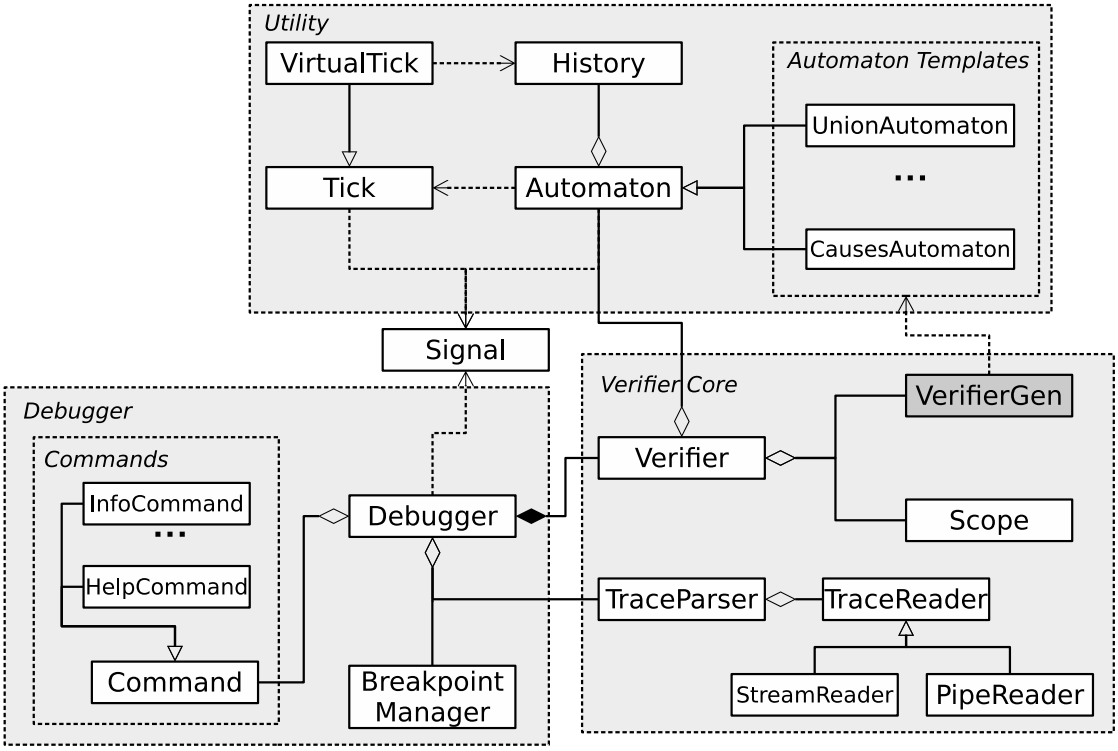

Figure 15: Verifier architecture

The digram can be divided mainly into three parts:

- Verifier utilities consisting of *automaton templates* and some basic classes such as `Tick`, `History` etc.
- Verifier core including class `Verifier`, parser and reader of traces and code generated by TPSL compiler.
- Debugger including class `Debugger` and a manager for breakpoints and some classes of commands.

In order to pass information among different components, a signal system is designed to achieve this goal. Class `Signal` is used by every other classes to emit a signal, implemented similar to call-back functions.

In the following sub-sections, each part will be introduced one by one and some concepts beneath the surface will be explained in detail.

16.3 Automata Utilities

Automaton together with some utility classes consists the most basic parts of TRAP verifier. Class `Automaton` is the base class and is inherited by every automaton template. It defines some pure virtual functions that every template must implement to provide polymorphic interface for verifier, listed below:

- Function `accept()`, takes a pointer of class `Tick` as input and make transitions. It is called every time a tick is dispatched to corresponding automata.
- Function `check_final_state()`. Automaton has no sense of scope and each end-of-scope is treated as end-of-simulation. This function is called when end of simulation is reached to check the current state of automaton.
- Function `system_reset()`, takes no argument and is called every time the whole verifier system is reset. For example, when user input command *run* in the debugger, this function of each automaton will be called.

Each automaton stores a local history of pointers to accepted ticks by class `History`. The local history is then retrieved at any time when needed, and cleared every time the automaton goes to initial state or final state. Apart from it, a global history is kept to store the statistics information of each clock, such as global count or so.

Instance of class `VirtualTick` is *emitted* when a clock expression automaton produces a virtual tick. When emitted, source automaton will attach its history to `VirtualTick` instance and the history is then merged into the history of target automaton which accepts the tick.

16.4 Verifier Core

This part provides core functionality for verification. It consists of several classes:

- `TraceReader` provides interface to read clock ticks from trace source. It is currently implemented as `PipeReader` and `StreamReader`, which reads ticks from named pipe or C++ `istream` respectively.
- `TraceParser` contains an object of `TraceReader` and provides interface to parse stream of ticks and serve the ticks one by one.
- `Scope` records the scope information, including inside or outside a defined scope.
- `Verifier` is the core class for verification. It keeps lists of automata and scopes. Automata are initiated and initialized by generated code (`VerifierGen` in figure 15). As mentioned above, because automata have no sense of scope, it is recorded and maintained by `Verifier`. Also, `Verifier` provides interface function to handle ticks one by one. It takes one tick and process it, if any property is violated, `Verifier` prints an error message and signals an error.

16.5 Debugger

This part provides user interface and everything related to interaction. The core class `Debugger` is responsible for:

- Calling functions provided by `TraceParser` to parse the trace and get ticks one by one;
- Calling functions provided by `Verifier` and pass each tick gain from `TraceParser` to it. If any error happens, error signal emitted by `Verifier` is caught by `Debugger` and appropriate action will be performed, such as pausing the debugger.
- Accepting user input and passing arguments to corresponding `Command`, which will then give information or change some state according to user command.
- Managing the breakpoints by `BreakpointManager`. As introduced in section 12.3, user can set breakpoints so that debugger will pause and let user input command again. Class `BreakpointManager` is responsible for managing and detecting breakpoints.

17 Summary

This section introduces the implementation of TRAP system. It can be divided into three parts, trace generation, property specification and verification: 1) trace mapping algorithm is described to show how TRAP maps pattern of trace items into logical clocks; 2) the relationship between TPSL and CCSL is given, together with the symbolic transducer representation for each TPSL rule; 3) the system architecture of verifier is given to show the overall process of verification and debugging.

TRAP is currently implemented in C++, thus all the code generated by every compiler is in C++. However, it is possible for compilers to generate other type of code, even though it is not implemented yet. So verifier and other parts are loosely coupled and thus can be replaced.

18 Conclusion

This paper introduces a model driven tool chain, named TRAP, to automatically analyze simulation traces and verify system properties at runtime. It can be integrated with any simulation back-end as long as the target simulator imports required module from TRAP that enables functionality for simulator to generate logical clocks.

Thanks to Eclipse and Xtext framework, four DSLs are defined to model the whole process of trace analysis: 1) TISL is used to structurize raw traces and provide informational trace items; 2) LCSL is used to define logical clocks as "event" used to express system properties; 3) STML is used to provide mapping rules to transform trace items into logical clocks; 4) TPSL is used to define system properties that are constraints to be satisfied referring to corresponding traces. Besides, a verifier containing a debugger is defined to verify whether system constraints are violated and gives analysis result.

The goal of TRAP is to provide a practical tool chain that can do runtime trace verification. The property language is inspired by CCSL and compared to formal languages such as temporal logic, TPSL has a more user-friendly grammar and its semantic covers most user cases. Even though TPSL is not offering more expressive power than any variant of temporal logic, it is more convenient for engineers to learn and use.

TRAP can massively reduce the amount of traces generated. Compared with other tools that can reduce the amount of traces, another advantage of this technique is that it does not require

the recompilation of simulator when mapping methods or system properties have changed. This provides a mechanism to separate simulator developer and simulator user, making it easy for engineers to explorer failure cases.

In the current implementation, trace file and pipe mode is developed to support both offline and online verification, and a very simple trace format is used to store trace information. In the future, more trace format supports will be added and more input source such as socket will be taken into consideration.

19 Acknowledgements

I would like to thank all those who helped me along the way, in any form. First thanks to Dr.Vania Joloboff my advisor, who guided me through the way of exploring not only this project but also the whole computer science world. May the good code be with you! Also thanks to Pr. Frédéric Mallet who helped me with the CCSL knowledge.

Secondly I want to thank those people who helped me getting here, including Pr. Wang Changbo and Pr. Zhu Huibiao, professors at ECNU, and Pr. Patrice Quinton from ENS Rennes.

Thirdly some special thanks to some local friends who helped me settled in Rennes, such as Marie Charrier, Deborah France Piquet and Annie Morin. Also I sincerely appreciate supports from my family, my girlfriend, and my good friend John.

References

[1] C. Helmstetter and V. Joloboff, "Simsoc: A systemc tlm integrated iss for full system simulation," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pp. 1759–1762, IEEE, 2008.

[2] 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.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[4] Y. S. Ramakrishna, P. M. Melliar-Smith, L. E. Moser, L. K. Dillon, and G. Kutty, "Interval logics and their decision procedures: part i: an interval logic," *Theoretical Computer Science*, vol. 166, no. 1, pp. 1–47, 1996.

[5] F. Balarin, J. Burch, L. Lavagno, Y. Watanabe, R. Passerone, and A. Sangiovanni-Vincentelli, "Constraints specification at higher levels of abstraction," in *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, pp. 129–133, IEEE, 2001.

[6] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Automatic trace analysis for logic of constraints," in *Design Automation Conference, 2003. Proceedings*, pp. 460–465, IEEE, 2003.

[7] 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, IEEE, 2012.

[8] 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.

[9] 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.

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

[11] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pp. 114–122, IEEE, 1999.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pp. 411–420, IEEE, 1999.

[13] S. Konrad and B. H. Cheng, "Real-time specification patterns," in *Proceedings of the 27th international conference on Software engineering*, pp. 372–381, ACM, 2005.

[14] 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.

[15] F. Mallet, J. DeAntoni, C. André, and R. De Simone, "The clock constraint specification language for building timed causality models," *Innovations in Systems and Software Engineering*, vol. 6, no. 1-2, pp. 99–106, 2010.

[16] M. Veanes, "Applications of symbolic finite automata," in *International Conference on Implementation and Application of Automata*, pp. 16–23, Springer, 2013.

[17] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic finite state transducers: Algorithms and applications," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 137–150, 2012.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399