LLFSMs to *TLA+*: A Model-to-Text Transformation of Executable Models Enabling Specification and Verification of Multi-Threaded and Concurrent Systems

Vladimir Estivill-Castro[1][a], Miguel Carrillo[2][b] and David A. Rosenblueth[2][c]

[1]*Department of Engineering, Pompeu Fabra University, Roc Boronat 138, Barcelona 08018, Spain*

[2]*Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, Universidad Nacional Autónoma de México, Apdo. 20-126, Ciudad de Mexico 01000, Mexico*

Keywords: Reasoning About Models, Model Transformation, Executable Models, Formal Verification.

Abstract: As complexity of software systems increases, ensuring reliability becomes ever more crucial. Despite advances, behaviour-modelling techniques still face challenges due to semantic gaps. This work focuses on translating Logic-Labelled Finite-State Machines (LLFSMs) to the Temporal Logic of Actions (TLA), bridging the gap between a time-triggered formalism and common temporal logic for model checking. The translation is innovative as multi-threaded and distributed systems can now be designed using LLFSMs. We illustrate the translation with Fischer's protocol (for multi-threaded systems), and release tools with examples for distributed systems. The approach addresses semantic gaps from three sources: differing finite-state machine semantics, variations in translating to executable models versus models for checking, and discrepancies between abstract and executable model translations.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Achieving dependable and trustworthy behaviour in software for autonomous, cyber-physical and real-time systems, poses significant challenges. While progress has been made in *Model-Driven Software Development* (*MDSD*), there are indications that tools and techniques still need to incorporate executable and verifiable modelling fully. We introduce a model-to-text translation of arrangements of Logic-Labelled Finite State Machines (LLFSMs) to the Temporal Logic of Actions (*TLA*).

The industry's current adoption of model transformation techniques seems marginal because of semantic issues (Bucchiarone et al., 2020). Our translation facilitates the development of software models for multi-threaded and distributed systems based on the ubiquitous and well-known concept of finite-state machines. It enables the simulation and execution of the models under a transparent and clear semantics. Without becoming familiar with the target language of model checkers, developers can verify their models without semantic gaps.

Our translation enables, for the first time, the use of LLFSMs to model multi-threaded and distributed systems. Our prototype implementation is based on the Eclipse Modelling Framework (EMF) but, as opposed to our previous translation of LLFSMs to `SMV` (Carrillo et al., 2020), we do not use ATL. Thus, our implementation prototype executes outside Eclipse and includes demonstrations of two prominent examples in the literature of distributed systems: the classical *two phase commit* and the well-studied elevator example of the IEC 61499 standard. With this paper, we release Docker containers that enable the full reproduction of all examples[1]. Moreover, we also implement and include a parallel translation to `SMV`. Each example includes verification of safety properties, liveness properties, and real-time properties without semantic gaps. Moreover, our models are also translated to three executable formats without semantic gaps (see Fig. 1).

Sec. 2 argues why model translation to *TLA+* is relevant and is followed by a discussion on seman-

[a] https://orcid.org/0000-0001-7775-0780
[b] https://orcid.org/0000-0003-2105-3075
[c] https://orcid.org/0000-0001-8933-8267

[1] A "How_To" document (Estivill-Castro et al., 2024) assists using the Docker containers hub.docker.com/r/vladestivillcastro/llfsms-examples.

Estivill-Castro, V., Carrillo, M. and Rosenblueth, D. A.
LLFSMs to TLA+: A Model-to-Text Transformation of Executable Models Enabling Specification and Verification of Multi-Threaded and Concurrent Systems.
DOI: 10.5220/0013094700003896
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 15-26
ISBN: 978-989-758-729-0; ISSN: 2184-4348
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

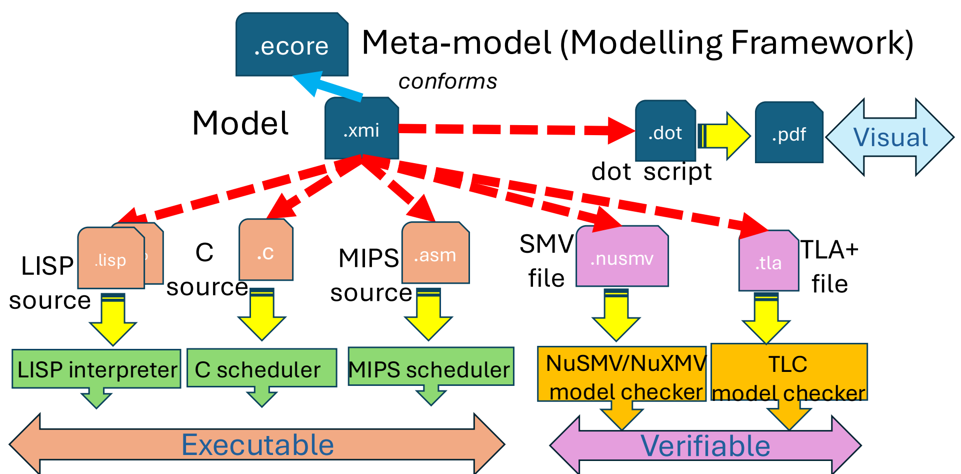


Figure 1: Our translator (in `java`) takes models (`.xmi`) and produces executable and verifiable models.

tic gaps. We present the advantages of LLFSMs as behaviour models and we summarise related work. Sec. 4 presents the translation using Fischer's Protocol as a running illustration. This protocol is central to multi-threaded systems, particularly to *TLA+*. Fischer's Protocol is correct for mutual exclusion but not necessarily for avoiding starvation. We discuss models of schedulers in Sec. 5. This is also a first for LLFSMs: on one hand, the scheduler for the arrangement is an LLFSM itself, and on the other, formal verification of the scheduler is enabled by our model-to-text translation. Sec. 6 illustrates further advantages before we conclude in Sec. 7.

2 JUSTIFICATION

TLA has been central to the specification and formal verification of distributed systems (Merz, 2019). Formal verification with *TLA+* of multi-threaded systems and concurrent systems is regularly used by suppliers of cloud services such as Amazon (Newcombe et al., 2015) and Microsoft (Kuppe, 2023). The authors of the *IronFleet* methodology highlight *TLA+* as one of the central tools for formal verification (Hawblitzel et al., 2015). Others agree: "*For our work,* TLA *perfectly meets our requirements*" (Niyogi and Nath, 2024).

Building a specification with *TLA+* is equivalent to defining *"the system's* spec*"* where a *spec* is *"a succinct description of every allowable behaviour of the system"* (Hawblitzel et al., 2015) or the Kripke structure (a large non-deterministic state machine). A Kripke state v is a valuation of all system variables. There is a Kripke transition from one Kripke state v_1 to another v_2 if the system can reach v_2 from v_1 in a single execution step. In *TLA+* and `SMV`, the Kripke structures is defined implicitly, usually composing modules defined with specific notation from mathematics and logic (Konnov et al., 2022).

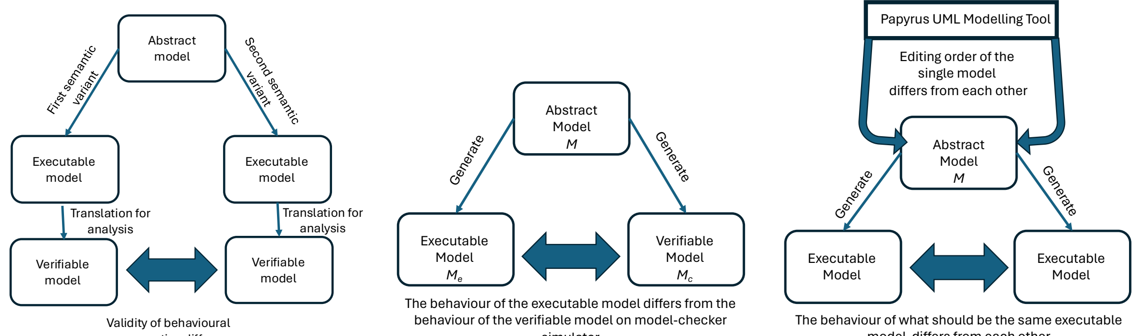
We enable the construction of *specs* using models of behaviour. We aim to address requests such as *"Although* TLA+ *specifications are purely declarative, they are meant to be used to describe algorithmic behaviour, and that should be simulatable, executable and testable*" (Moreira et al., 2022).

Requests for tools that generate executable code from formal specifications often lead to discussions about technical differences between formalisms, with little support for bridging the gap between executable and verifiable models (Carvalho, 2019). We postulate that specifications can be created using LLFSMs, which can be simulated, translated into high-level programming languages, and used as input for model checkers. For developers, working with executable models and simulating the specification is a more intuitive approach (Martínez et al., 2012). Our translation facilitates the development of *specs* as executable models.

Developers prefer using executable models over mathematical notation from specification languages such as Z and *TLA*, as evidenced by *"programmers do not enjoy reading white papers full of mathematical notation"* (Bellotti, 2019).

Translating these specifications into executable systems is still largely a manual process: "*The adoption of formal methods in industry is challenged by the cost and complexity involved in the formal specification of the system*" (Nicolás and Toval, 2009). The essence of this paper is that LLFSMs are just as formal (as they can be automatically translated into `SMV` or *TLA+*) and just as executable (as they can be translated into MIPS and C).

IronFleet stresses the importance of eradicating



(a) What is valid of the executable models?
(b) What is valid of the executable model M_e?
(c) What is valid, if the same model can have different behaviours?

Figure 2: Known semantic gaps.

semantic gaps (Hawblitzel et al., 2015). (Kurshan, 2018) attributes the lesser penetration in the industry of formal verification to wide semantic gaps.

We are concerned with three types of semantic gaps (Fig. 2). First, Fig. 2a shows executable models generated from *UML* models for analysis with a model checker. Despite numerous attempts to address this translation, however, this process may yield inconsistent formal verification outcomes due to multiple semantic variants within *UML*. Under one semantic variant, the properties may be valid but invalid under another (Besnard et al., 2018). The IEC 61499 also suffers from semantic variants (Cengic and Akesson, 2010). Moreover, *UML* has several extensions (fUML, krtUML, SysML, UML-RT) which not only have their extended semantics but offer semantic variants among them. (Posse and Dingel, 2016) report on six attempts for formalising semantics and their Table 1 contrasts at least four.

A second gap involves scenarios where a model M is constructed and subsequently translated into both an executable model M_e and a model M_c for the model checker (Fig. 2b). Discrepancies may arise between the observed behaviour of the executable model M_e and the simulation of M_c on the model checker. For instance, consider a scenario where a Papyrus-RT model M comprises multiple communicating finite-state machines. Papyrus-RT commonly produces an executable model M_e designed for Linux that employs multitasking services for communication purposes. These intricacies rarely appear in the idealised model M_c. Consequently, during simulation by the model checker, M_e may exhibit traces not present in M_c (Sahu et al., 2020).

A final gap arises when the semantics of a model depends on the sequence in which components of the model are edited within an *MDSD* tool (Pham et al., 2017) (Fig. 2c). This is a significant risk because visually identical models ought to possess identical semantics; (Guermazi et al., 2015) report extensive evaluation of executions, but even with the reference implementation, identical models can be created that behave differently (Estivill-Castro, 2021).

LLFSMs are designed with precise semantics for their execution and with familiar visual notation. LLFSMs were first used for robotic systems (Brooks, 1990) but have been applied in embedded systems (McColl et al., 2022; Carrillo et al., 2020), micro-controllers, and FPGAs.

In LLFSMs, predicates (as opposed to events) label the transitions between states 1) facilitating simple semantics, e.g., using polling instead of interruptions, and 2) enabling scheduling the machines as time-triggered systems, which are easier to verify than event-triggered machines (Furrer, 2019). By using LLFSMs, we eliminate the second type of semantic gap (Fig. 2b) to the point that, as suggested by (Besnard et al., 2018), although inefficient, the model checker could act as the interpreter for execution.

3 RELATED WORK

There have been efforts to translate *UML* statecharts into PROMELA for the SPIN model checker (Latella et al., 1999). There have also been efforts to translate to timed automata or some form of Petri nets (André et al., 2023). Many others translate or generate `SMV`, the input language of `NuSMV`, started by (Kwon, 2000). Other formalisms have been the target of the translation, such as process algebras, i.e., Hoare's Communicating Sequential Processes (CSP), or the Language Of Temporal Ordering Specification (LOTOS) as well as PVS, KIV, B and Z (André et al., 2023). None of these translate to *TLA+*.

Three types of research are closest to our work. First is the translation using ATL to `SMV` by (Carrillo

```
Bool sensor:  T1_Non_C_Sec_Delay, T1_C_Sec_Delay, T2_Non_C_Sec_Delay, T2_C_
Sec_Delay
Int blackboard:  Shared:0..2
```

dInitSTARTER(0)

1:[true](Tid:00)

START(1)

write "Initialise shared variable"
Shared ::= Not_A_Thread

Figure 3: Initialisation for Fischer's protocol.

et al., 2020) since the source is also an LLFSM model. Second is a translation to `SMV` from models built with Function Blocks of the IEC 61499 (Patil et al., 2015) since, the machines inside function blocks have no events, only guards (thus, these are LLFSM), and events are handled outside the block on the interface and by priority schemes (using timers). Third, some degree of parallelism is achieved by (McColl et al., 2022) by scheduling LLFSMs in groups of independent, not communicating LLFSMs and inside groups using a predefined round-robin schedule.

4 TRANSLATION

4.1 Example

Fischer's protocol is a software-based approach requiring no hardware support to ensure mutual exclusion among multiple threads (Lamport, 1987; Lamport, 2005a; Lamport, 2005b; Lamport, 2024). Its correctness has been validated multiple times. The variable `Shared` identifies the thread in the critical section and initially holds the predefined constant `NotAThread` (see the STARTER machine in Fig. 3, with index 0 in the arrangement).

Each thread (Fig. 4) can spend an arbitrarily long time both in its non-critical section and in its critical section.

The protocol coordinates several copies of the LLFSM in Fig. 4. For illustration, we use one LLFSM modelling the Fischer's protocol thread (Fig. 4) with index 1 in the arrangement but with two copies (however, our tools allow to set the number k of threads). In the translation, the threads are named `ThreadID` with an `ID` $\in \{1,2\}$ [2]. Fischer's protocol is a suitable example of multi-threaded system because each thread gets a turn by a non-deterministic scheduler (at any time any thread can be the next thread).

[2]The transitions of LLFSM are identified with names of the form Tid<*machine index*><*transition index*>. Thus, the `Tid00` in the name of the only transition in Fig. 3, but `Tid10`, `Tid11`, `Tid12`, `Tid13`, `Tid14`, `Tid15`, and `Tid16`, are the names of the transitions in Fig. 4.

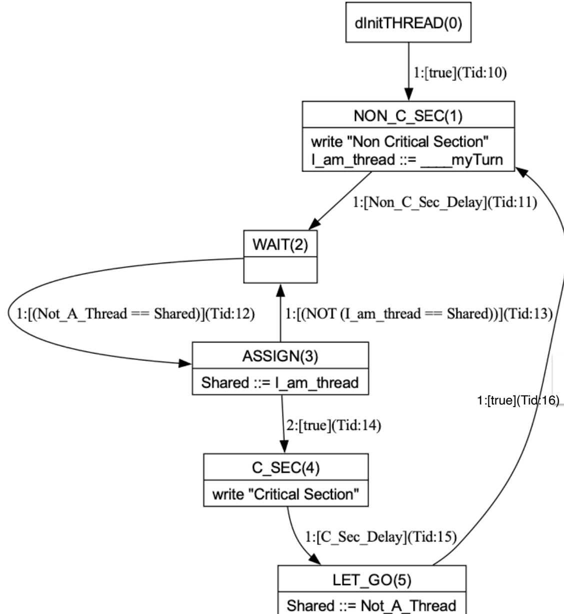


Figure 4: The LLFSM corresponding to a thread.

4.2 Brief Introduction to LLFSMs

A system is comprised of an *arrangement* (i.e., a sequence of instances) of LLFSMs. Each LLFSM consists of *states* and *transitions*. Time progresses as the system executes code within a state; a transition moves the execution from one state to another.

LLFSMs are extended machines, where executable code can also be associated with a state involving Boolean and integer variables. In line with the principles of *MDSD*, we abstract the action language for this executable code in a meta-model. The LLFSMs presented here will not include sections (or equivalently, they only have an `OnEntry` section and its code is the code of the state). LLFSMs featuring states with sections (`OnEntry`, `OnExit`, and `Internal`) can be model-to-model translated into LLFSMs without sections and yet maintain equivalent semantics (Carrillo et al., 2020).

A critical aspect of LLFSMs' semantics is that the execution of code statements within a state is atomic: once execution reaches a state, the entire code of the state is executed without interruption. This atomicity is crucial for bridging the semantic gap between model checkers and executable implementations. However, it is possible to build non-atomic versions if the modeller desires to do so.

A state can serve as the source for multiple transitions, ordered in sequence $\langle l_0, \ldots, l_t \rangle$. A transition $l_i = (B_i, t_i)$ (where B_i is the Boolean label and t_i is the target state) updates the machine current state to t_i if B_i evaluates to true and all labels in preceding tran-

sitions $l_0, \ldots, l_{i-1}$ evaluate to false. Therefore, at most one transition can be activated at a time. A machine updates its current state only when such LLFSM has its turn and a transition has a Boolean expression that evaluates to true. When it has its turn, the machine M executes a *ringlet* atomically, that is:

1. M copies the current valuation v (the assignment of all variables to their values),
2. it evaluates the Boolean expression of all transitions leaving the current state in that fixed valuation v,
3. it executes the code of the target state of the transition that fired and updates its current state to the target state (or does nothing if no transition fires).

4.3 The Translation and Its Illustration

We present a model-to-text transformation enabling the creation of a formal specification for *TLA+* from executable models of sequential, distributed, or multi-threaded systems. (Carrillo et al., 2020) provided a formal sequential semantics for an arrangement A of LLFSMs.

Each machine must have a single initial state. The case where a model requires a machine M with multiple initial states I_i chosen non-deterministically can be modelled with a simple transformation.

All machines begin in a pseudo-state (prefixed with `dInit`) as their current state. The initial pseudo-state contains only one transition, labelled "`true`". For each machine M, the first turn of M updates M's current state (from the pseudo-state `dInit`M) to M's initial state. In our running example, STARTER will execute first, initialising variable `Shared` that initially has no value.

A *TLA+* specification is the conjunction of three formulas: *Init*, *Next*, and *Liveness* (Lamport, 2002).

$$\mathit{Init} \land \Box[\mathit{Next}]_{vars} \land \mathit{Liveness}.$$

Each formula serves the following purpose.

1. **Specification of Initial States (*Init*):** This formula defines the initial Kripke states. Since Kripke states are valuations, only those valuations that make *Init* true are initial Kripke states.
2. **Safety Specifications (*Next*):** This second formula defines which actions the system can perform and is a disjunction ($\lor$) of formulas in the form $\psi_i \implies \phi_i$. Each formula $\psi_i \implies \phi_i$ indicates there is a possible transition from any Kripke state satisfying ψ_i to any target Kripke state satisfying ϕ_i. Thus, $\psi_i \implies \phi_i$ describes possible edges within the Kripke structure.
3. **Liveness Specifications (*Liveness*):** A third formula specifies when a Kripke transition must be taken.

We now describe the model-to-text transformation responsible for constructing these three components.

4.3.1 Constructing the Formula *Init*

A *TLA+* specification starts with a declaration of the variables. As each LLFSM maintains its current state, we declare variables in *TLA+* that serve as program counters. For each machine `M_NAME`, we declare a variable `M_NAMEState`.

Using the running example from Fig. 3 and Fig. 4, we show the declaration of variables generated by our tools. The *TLA+* specifications and mathematical notation we show next are the output of the coded translation (and text in grey are comments). For the Fischer's protocol arrangement, that has three machines (a STARTER and two threads) we have three variables that keep track of the current state of each machine.

```
\* The variables that represent the current state of a machine
VARIABLES STARTERState, THREAD_1State, THREAD_2State
```

LLFSMs have two primary types of variables. Local variables (which are only visible to one machine) and shared variables (with scope across the arrangement). Shared variables are of four types: those named blackboard variables (which can be both read and written), those named sensor variables (that are read only, but the environment can modify in an open model), and effector variables (that are write only, and the environment observes).

We show the definition of the variables of our running example. In *TLA+*, variable names cannot be duplicated, and we declare them so all variables are part of the subscript of the weak fairness condition for the *Liveness* formula (see Sec. 4.3.3). The user variables are followed by the declaration of the variable `turn`.

```
\* Boolean sensor variables
, T1_Non_C_Sec_Delay, T1_C_Sec_Delay
, T2_Non_C_Sec_Delay, T2_C_Sec_Delay
\* Integer whiteboard variables
, Shared
\* The integer local variables of all machines
  \* The integer local variables of machine THREAD_1
, THREAD_1_I_am_thread_T1
  \* The integer local variables of machine THREAD_2
, THREAD_2_I_am_thread_T2
\* Which machine takes a turn
, turn
```

In *TLA+*, it is necessary to explicitly specify the types of all variables as an invariant, represented by

formulas in the form of $G\varphi$, where G is the LTL operator globally/always. This is achieved for the variable that keeps the current state of the LLFSM by defining that the potential values of the current state consist of the states of the LLFSM, for each LLFSM.

For the example, we show only a few invariants, but the translation includes one for each variable.

The type-correctness invariants

$\texttt{TypesTHREAD_1_I_am_thread_T1OK} \triangleq$
$\texttt{THREAD_1_I_am_thread_T1} \in \{n \in Int : n \geq 0 \wedge n \leq 2\}$
$\texttt{TypesT1_Non_C_Sec_DelayOK} \triangleq$
$\texttt{T1_Non_C_Sec_Delay} \in \text{BOOLEAN}$
$\texttt{TypesSharedOK} \triangleq \texttt{Shared} \in \{n \in Int : n \geq 0 \wedge n \leq 2\}$

Among the invariants is the constraint that the scheduler keeps the value of turn to be the index of one of the LLFSMs in the arrangement.

We now provide the predicate that defines the initial states of the Kripke structure. The formula is named Arrangement_NameInit and is a conjunction of what is possible at the start of the system. Each machine Nachine_Name will have its counter initialised to the pseudo-state:

Nachine_NameState = "dinitNachine_Name".

We express each variable as initially undefined but holds some value in its domain. Consequently, there exist numerous initial states in the Kripke structure, encoding that execution can start with any combination of valuations for the variables. In our example, before STARTER runs, the variable Shared can initially hold any value in its domain. The sensor variables with postfix Delay are controlled by the environment, enabling each thread to stay in its critical or non-critical sections for an undefined long time.

The variable turn is set to the first machine (with index 0) in our example, but in general it could be another value. Also, if we intended to implement an arrangement of LLFSM where the arrangement could start from any LLFSM, we would simply alter the turn=0 statement to turn $\in \{0, 1, \ldots, n\}$. This adjustment shows the first step towards furnishing the multi-threaded or distributed semantics.

The initial predicate. Each machine starts in its initial state.

$\texttt{FischerInit} \triangleq \wedge\ \texttt{turn} = 0$
$\wedge\,\texttt{STARTERState} = \text{"dInitSTARTER"} \wedge \texttt{THREAD_1State} = \text{"dInitTHREAD_1"}$
$\wedge\,\texttt{THREAD_2State} = \text{"dInitTHREAD_2"}$
$\wedge\,\texttt{T1_Non_C_Sec_Delay} \in \text{BOOLEAN}$
$\wedge\,\texttt{T1_C_Sec_Delay} \in \text{BOOLEAN}$
$\wedge\,\texttt{T2_Non_C_Sec_Delay} \in \text{BOOLEAN}$
$\wedge\,\texttt{T2_C_Sec_Delay} \in \text{BOOLEAN}$
$\wedge\,\texttt{Shared} \in \{0, 1, 2\} \wedge \texttt{THREAD_1_I_am_thread_T1} \in \{0, 1, 2\}$
$\wedge\,\texttt{THREAD_2_I_am_thread_T2} \in \{0, 1, 2\}$

4.3.2 Constructing the Formula *Next*

In general, for every state s of every machine M in the arrangement, we proceed to define the transitions out of s. We write these transitions as a disjunction because a machine M with a current state s and a sequence of transitions $\langle (e_1, s_{t1}), (e_2, s_{t2}), \ldots, (e_t, s_{t_t}) \rangle$ (where s represents the current and source state, $s_{t1}, s_{t2}, \ldots, s_{t_t}$ are the target states, and $e_1, e_2, \ldots, e_t$ are Boolean expressions) moves to a new Kripke state if and only if $\{(e_1, s_{t1}) \text{ fires}\} \oplus \{(e_2, s_{t2}) \text{ fires}\} \oplus \cdots \oplus \{(e_t, s_{t_t}) \text{ fires}\} \oplus \{\text{no transition fires}\}$, where $\oplus$ stands for exclusive or. Hence, a Kripke transition occurs if and only if an LLFSM M has its turn, and either: (a) the current state of M has a transition that fires and no preceding transition with the current state as the source state fires or (b) no transition fires (but not both). If no transition fires, the system will still progress as another LLFSM in the arrangement has its turn.

Therefore, for every instance of a transition (e_j, s_{t_j}) out of the current state s_i for the machine with turn t, we write a formula incorporating the following components.

$$T_{t,j} \triangleq (\texttt{turn} = t) \wedge (\texttt{current_State_Variable} = s_i) \wedge (\forall v < j, \neg e_v) \wedge e_j.$$

We illustrate this part of the translation with the transition out of the state T1_WAIT to the state T1_ASSIGN:

$$T12 \triangleq \texttt{turn} = 1 \wedge \texttt{THREAD_1State} = \text{"T1_WAIT"} \wedge (\texttt{Not_A_Thread} = \texttt{Shared})$$

This formula is the conjunction of three conditions: the turn matches the machine number, the current state is T1_WAIT, and the transition is labelled by a test for equality (a Boolean formula) involving the shared variable Shared.

A formula for the effect of a transition must be specified for each transition in the system. In *TLA+* we must continue the description of a step in the system by specifying the new Kripke state (the new valuation). The first effect of a firing transition from s_i to s_{jt} is the update of the state. The program counter for the state must be updated to the target state. Thus, the definition of the transition continues with a conjunction with the following equality

$$\wedge\ \texttt{Machine_NAMEState}' = s_{jt}.$$

In *TLA+*, the value of a variable in the next Kripke state is primed.

For the example of transition T_{12} of the second thread in Fischer's protocol, this means

$$\wedge\,\texttt{THREAD_1State}' = \text{"T1_ASSIGN"}$$

The second effect is the atomic execution of the OnEntry in the target state. This depends on the assignment statements in the target state. For the first thread, arrival to the state T1_ASSIGN implies evaluation of only one assignment.

$$\wedge\ \texttt{Shared}' = \texttt{THREAD_1_I_am_thread_T1}$$

There is a third part for each transition. Formalisms like SMV and *TLA+* must specify what happens with everything else. Crucially, everything else remains unchanged, except for the sensor variables, since the environment could modify such variables. In *TLA+*, we specify this aspect of sensor variables, indicating that their new value is some value in their domain.

For the running example of THREAD_1, transition T_{12} continues as follows.

$\wedge \text{UNCHANGED } \langle \texttt{STARTERState}, \texttt{THREAD_2State} \rangle$
$\wedge \text{UNCHANGED } \langle \texttt{THREAD_1_I_am_thread_T1} \rangle$
$\wedge \text{UNCHANGED } \langle \texttt{THREAD_2_I_am_thread_T2} \rangle$
$\wedge \texttt{T1_Non_C_Sec_Delay}' \in \text{BOOLEAN}$
$\wedge \texttt{T1_C_Sec_Delay}' \in \text{BOOLEAN}$
$\wedge \texttt{T2_Non_C_Sec_Delay}' \in \text{BOOLEAN}$
$\wedge \texttt{T2_C_Sec_Delay}' \in \text{BOOLEAN}$

LLFSMs can model open and closed models. An open model can characterise the environment through sensor variables. In this case, the potential successor Kripke states are as many as the possible combinations of assignments to the sensor variables. Closed models, by contrast, are simpler because the environment can be modelled as an additional LLFSM and sensor variables as blackboard (shared) variables. The additional LLFSM provides values to those blackboard (shared) variables while other LLFSMs in the arrangement read them.

For each transition, the next value of the variable `turn` must be defined. However, how `turn` is updated determines the difference between sequential execution, or multi-threaded or distributed system. Since the update appears in all definitions of all transitions, it can be factored out in the formula defining *Next*.

There is a possibility that no transition fires, in which case every action-language variable remains unchanged. This also must be specified for each machine M. We define a default formula indicating everything remains the same guarded by a conjunction that all other transitions must evaluate to false.

We exemplify the default transition for the machine that corresponds to the first thread in Fischer's protocol.

$\texttt{T1condDefault} \triangleq \wedge \texttt{turn} = 1$
$\wedge (\neg \texttt{THREAD_1State} = \text{“dInitTHREAD_1”}$
$\wedge \neg(\texttt{THREAD_1State} = \text{“T1_NON_C_SEC”}$
$\wedge \texttt{T1_Non_C_Sec_Delay})$
$\wedge \neg(\texttt{THREAD_1State} = \text{“T1_WAIT”} \wedge (\texttt{Not_A_Thread} = \texttt{Shared}))$
$\wedge \neg(\texttt{THREAD_1State} = \text{“T1_ASSIGN”} \wedge$
$(\neg(\texttt{THREAD_1_I_am_thread_T1} = \texttt{Shared})))$
$\wedge \ \neg(\texttt{THREAD_1State} = \text{“T1_ASSIGN”} \wedge \text{TRUE})$
$\wedge \neg(\texttt{THREAD_1State} = \text{“T1_C_SEC”}$
$\wedge \texttt{T1_C_Sec_Delay})$
$\wedge \neg(\texttt{THREAD_1State} = \text{“T1_LET_GO”} \wedge \text{TRUE})$
$)$
$\wedge \text{UNCHANGED } \langle \texttt{THREAD_1State} \rangle$
$\wedge \text{UNCHANGED } \langle \texttt{STARTERState}, \texttt{THREAD_2State} \rangle$
$\wedge \text{UNCHANGED } \langle \texttt{THREAD_1_I_am_thread_T1} \rangle$
$\wedge \text{UNCHANGED } \langle \texttt{THREAD_2_I_am_thread_T2} \rangle$
$\wedge \text{UNCHANGED } \langle \texttt{Shared} \rangle$
$\wedge \texttt{T1_Non_C_Sec_Delay}' \in \text{BOOLEAN}$
$\wedge \texttt{T1_C_Sec_Delay}' \in \text{BOOLEAN}$
$\wedge \texttt{T2_Non_C_Sec_Delay}' \in \text{BOOLEAN}$
$\wedge \texttt{T2_C_Sec_Delay}' \in \text{BOOLEAN}$

Thus, the formula for the *Next* predicate is the disjunction because one of the machines is awarded the `turn`, and that machine executes a transition, or if no other transition fires, it fires its default transition. The previous transitions must be false for each transition to fire, and this pattern is also for the default transition, placed last. As the turn update happens in every disjunct, as we anticipated, we can factor it out.

In the sequential execution of an arrangement with n LLFSMs, the variable `turn` (not accessible to the modeller) is assigned values in a round-robin fashion: $\texttt{turn} \leftarrow (\texttt{turn} + 1) \bmod n$, typically starting from $\texttt{turn} \leftarrow 0$. However, after factoring out the update of the `turn`, it becomes clear that we can achieve a multi-threaded (or distributed system) by adjusting how the `turn` is updated. When modelling a multi-threaded system, where the CPU can allocate the next ringlet to any of the LLFSMs, the update of the `turn` is non-deterministic:

$\texttt{turn}' \in \{0, 1, 2\}$.

Therefore the generated *Next* formula for Fischer's example is as follows.

Move to a successor state in the *Kripe* Structure

$FischerNext \triangleq$

A non-deterministic scheduler advances some transition

$\texttt{turn}' \in \{1, 2\} \wedge$

The transitions of machine STARTER

$\vee \ \ T00 \ \ \vee \ \ T0condDefault$

The transitions of machine $THREAD_1$

$\vee\, T10 \vee T11 \vee T12 \vee T13 \vee T14 \vee T15 \vee T16 \vee T1condDefault$

The transitions of machine $THREAD_2$

$\vee\, T20 \vee T21 \vee T22 \vee T23 \vee T24 \vee T25 \vee T26 \vee T2condDefault$

4.3.3 Constructing the *Liveness* Formula

The formula for *Liveness* should be a conjunction of weak and/or strong fairness formulas for subactions of *Next* since this guarantees that the specification is *machine closed* (Lamport, 2002). We will not define subactions or machine closedness. Arrangements of LLFSMs always progress, even if the machine holding the turn lacks a transition that fires, the execution proceeds to the next machine. This implies the absence of stuttering behaviours, where there are no behaviours that continuously enable a step but do not execute the step. Thus, we eliminate all potential stuttering behaviours using *TLA*'s construct for weak fair-

Table 1: Fischer's mutual exclusion verification (sec).

k	TLC		NuSMV	
5	3	±0.5	1	±0.05
6	27	±1.1	5	±0.11
7	166	±1.4	25	±0.32
8	1,085	±5.1	95	±0.43

ness WF. Adhering to the convention that *vars* represents the tuple of all variables,

$$\mathit{Liveness} \triangleq \mathrm{WF}_{vars}(\mathit{Next}),$$

where the formula $\mathrm{WF}_{vars}(A)$ in *TLA+* is defined as (Lamport, 2002)

$$\Box(\Box \mathit{ENABLED}\langle A\rangle_{vars} \Rightarrow < A >_{vars})$$

and it states that, if A ever becomes forever enabled, then an A step must eventually occur.

4.3.4 Using the Translation

Our translation enables the verification of the central property of mutual exclusion expressed in *TLA+* exactly as discussed by Lamport (Lamport, 2005a) by choosing TLC as our target model checker.

```
[](~((THREAD_1State="T1_C_SEC")
    /\(THREAD_2State="T2_C_SEC")))
```

Using our parallel translation, we can also formally verify the mutual exclusion of any two threads translating to SMV.

```
LTLSPEC G( !(THREAD_1.At_T1_C_SEC
           & THREAD_2.At_T2_C_SEC))
```

Other properties checked on Fischer's algorithm are deadlock freedom and a progress property (Lamport, 2005b; Lamport, 2005a) that we can verify with both model checkers. For instance, the following *Progress* (Lamport, 2005a) property.

$$\begin{aligned} \mathit{Progress} \quad \triangleq \quad & (\exists t \in \mathit{Thread} : pc[t] \in \{\mathit{WAIT}, \mathit{ASSIGN}\}) \\ & \implies (\exists t \in \mathit{Thread} : pc = C_SEC). \end{aligned}$$

This condition expresses that, "*if some thread is waiting to enter its critical section, then some thread (not necessarily the same one) will eventually enter*" (Lamport, 2005a).

Different model checkers use different computational resources for verifying a property. At least for Fischer's algorithm, NuSMV is computationally faster than TLC. We compared TLC with NuSMV repeating the verification five times for each model checker on the same computer. Table 1 shows average times (with 95% confidence intervals) verifying mutual exclusion for five to eight threads with a unrestricted scheduler. These results show that the coded translations are efficient.

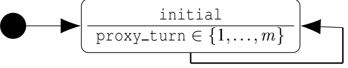


Figure 5: A nondeterministic scheduler encoded as an LLFSM.

4.3.5 The Translation's Structure

There are two ways by which a sequential arrangement of LLFSMs simulates the non-deterministic execution of a multi-threaded arrangement. The first version adds a scheduler to the arrangement.

Definition 1. *The* LLFSM-scheduler *version to simulate multi-threaded execution of an arrangement $A = \langle M_1, M_2, \ldots, M_m\rangle$ (where each M_i is an LLFSM) is a new sequential arrangement $A' = \langle S, M_1, M_2, \ldots, M_m\rangle$ with a new variable* turn_proxy*, where A' is forced to start with its first LLFSM S, the original variable* turn *in A is replaced by* turn_proxy*, and S is a scheduler which non-deterministically assigns the* turn_proxy *of a machine in A.*

The second version adds a Boolean sensor variable as conjunction with the existing Boolean expression of every transition.

Definition 2. *The* LLFSM-variable *version to simulate multi-threaded execution of an arrangement $A = \langle M_1, M_2, \ldots, M_m\rangle$ (where each M_j is an LLFSM) is a new sequential arrangement $A' = \langle M'_1, M'_2, \ldots, M'_m\rangle$ with a new Boolean sensor variable* grant_CPU*, all machines M'_j are copies of the corresponding machine M_j except that all transitions $T_i = (e_i, s_{t_i})$ of the machines M_j have been replaced by $T'_i = (e_i \wedge$ grant_CPU$, s_{t_i})$ for machine M'_j, and*

1. *the environment cannot set* grant_CPU *always to false from any point forwards, (that is, in* LTL*: G F* grant_CPU*), and*
2. *if the environment sets* grant_CPU *to true on machine M_j's turn, then it must sustain* grant_CPU *to true while M_j evaluates all transitions out of its current state.*

The following proposition establishes that the semantics of multi-threaded LLFSMs, can be used to obtain faithful translations in two equivalent ways.

Proposition 1. *The LLFSM-scheduler multi-threaded execution and the LLFSM-variable multi-threaded execution are equivalent.*

Proof. First, we show that any sequence $N = \langle t_1, t_2, \ldots\rangle$ (whether finite or infinite) of turns with $t_i \in \{1, \ldots, m\}$ for the multi-threaded execution of A is simulated by the round-robin execution of the *LLFSM-scheduler* A'.

Let $N = \langle t_1, t_2, \ldots \rangle$ be an arbitrary sequence of non-deterministic turns of a multi-threaded execution of A. The scheduler S is depicted in Fig. 5. It runs once for every round-robin cycle, and thus, it can happen that when the scheduler is granted a turn for the i-th time, it sets `turn_proxy` to t_i. Although the variable `turn` is updated in a round-robin fashion by the sequential execution, a machine M_i in the arrangement only reacts if `turn_proxy` equals i. Otherwise, M_i does nothing. Thus, for each scan across the arrangement, only one of the machines M_j executes its ringlet, precisely M_{t_i}, when `turn_proxy` equals t_i. Thus, the sequential execution of A' grants turns to machines M_i in the arrangement in exactly the sequence N of the multi-threaded execution of A.

We now proceed to the converse. We show that any round-robin execution R of the simulator arrangement A' corresponds to a multi-threaded execution of A. But let $N = \langle t_1, t_2, \ldots \rangle$ the sequence of values granted by the schedulers S to the variable `turn_proxy` on sequential execution R of A'. The sequential execution only enacts machine M_{t_i} on each round-robin scan over arrangement A. Thus, the multi-threaded execution with sequence of turns $N = \langle t_1, t_2, \ldots \rangle$ enacts the execution R.

Now we show that any sequence $N = \langle t_1, t_2, \ldots \rangle$ (whether finite or infinite) of turns with $t_i \in \{1, \ldots, m\}$ for the multi-threaded execution of A is simulated by the round-robin execution of the *LLFSM-variable* A'.

The idea is simple. In the round-robin execution of A', when `turn` equals t_i (at the i-th sequential scan), the environment sets `grant_CPU` to false for all machines in the scanning of the arrangement but `grant_CPU` is set to true for machine M_{t_i}. Thus, all transitions are guaranteed not to fire for all other machines. Machine M_{t_i} evaluates its transitions with `grant_CPU` set to true, which is logically equivalent as evaluating $(\text{true} \wedge a)e = e$ for every Boolean expression e labelling its transitions. Thus, the only actions that happen in the round-robin execution of A' are those in the multi-threaded execution of A'.

For the converse, since the environment cannot keep `grant_CPU` set to false forever, some machine m_i will get a turn. Since the environment must keep `grant_CPU` true for all outgoing transitions of the current state of M_i this is equivalent to granting a turn in a non-deterministic schedule to M_i. □

We have two immediate corollaries, one for *TLA+* and the following which is the `NuSMV` version.

Corollary 1. *Let P be a property. Using the* `NuSMV` *translation of the LLFSM-scheduler version of an arrangement A (modelling a multi-threaded system) to verify P is equivalent to using the* `NuSMV` *translation of the LLFSM-variable version of an arrangement A (modelling a multi-threaded system) to verify P.*

Since the LLFSM-variable version requires a delicate interaction of when `grant_CPU` is true, we prefer the translation using the LLFSM-scheduler version.

Fischer's protocol shows the case when there are several copies of the same LLFSM. There are two possible ways to perform the model-to-text translation of an arrangement when an LLFSM has several instances. The first option uses facilities in specifications for `NuSMV` or for `TLC` to create several instances of a module.

Definition 3. *The* LLFSM-module *version to translate a machine M with c copies in an arrangement of LLFSMs consists in using the* module *facility of the language L_C to create specifications of a model checker* C*, and to translate M to a module M_L and indicate that the L_C-specification has c instances of M_L.*

The second option is to use that `SMV`-specifications and *TLA*+-specifications allow arrays.

Definition 4. *The* LLFSM-array *version to translate a machine M with c copies in an arrangement of LLFSMs consists in using the* array *facility of the language L_C to create specifications of a model checker* C*, and to translate M to a module M_L, but for each element* `a` *of M, create an array* `a:[1..c]` *in M_L.*

Proposition 2. *At least for* `SMV` *and* TLA+*, the LLFSM-module translation of an arrangement and the LLFSM-array translation can be made equivalent.*

We elected to use the LLFSM-module translation for our implementations of the translation.

5 SCHEDULERS

In Fischer's protocol, the unrestricted scheduler of Fig. 5 may cause starvation. LLFSMs can be used to define, and analyse properties of alternative schedulers. We can verify the scheduler on its own or verify an arrangement with a specific scheduler.

The scheduler of Fig. 5 is automatically translated to `SMV`, and we can verify

Property 1 *at any time, any thread in the future can have a turn*, and

Property 2 *With two or more threads, at any point, for any thread t, there is a path that starves t forever.*

Because the scheduler of Fig. 5 allows a thread to monopolise the CPU, we formulate a new scheduler in Fig. 6 controlling t LLFSMs. The arrangement can include one starter LLFSM for set up. Here, for each

thread i, we have a variable $\texttt{waiting}_i$ that counts how long i has been waiting for the CPU. For this new model, we also apply our model-to-text transformations. We confirm additional properties:

```
LTLSPEC -- If thread i has starved to the maximum, it will
eventually  have a turn (no starvation)
G(Scheduler.waiting_i=g_c.MaxConsecutive
  -> F Scheduler.waiting_i=0)
CTLSPEC -- Thread i can starve for the maximum
EF (Scheduler.waiting=g_c.MaxConsecutive )
```

Modelling with LLFSM is agnostic to the model checker, and designers of behaviours can apply the translation to a model checker with which they are most comfortable enunciating verification properties. LLFSMs provide abstraction from the programming language for execution of the model. At first sight, the mathematical notation of *TLA+* appears to be more expressive than `SMV` since it supports sets and functions. However, `NuSMV` supports interactive simulation (*TLA+* only supports command-line trace generation). Also, *TLA+* does not have all the operators of *LTL*, notably, lacking `X` (*Next*) (Kröger and Merz, 2008).

There are two fundamental types of properties about real-time systems: upper bound and lower bound (Lamport, 2005b). With $t = 3$ and `MAX`=4, we show an example of each.

```
LTLSPEC G (!(w_b.the_Turn=i) -> F[0,11] w_b.the_Turn=i).
```

No thread, ever, waits for a turn more than 11 Kripke state transitions. Notice that this is a stronger time-domain property as the operator `F` (eventually) is replaced by a precise upper bound.

```
LTLSPEC G ((!(w_b.the_Turn=i)& Xw_b.the_Turn=i)
      -> H[0,1]!(w_b.the_Turn=i))
```

Since `X` is missing, we have not found how to express these properties in *TLA+*. But the next section illustrates an aspect feasible with `SMV` that seems more comprehensive with *TLA+*.

6 DISCUSSION

Every time we translate to `NuSMV`, we run this model checker with the option `-ctt` validating that the transition relation in the Kripke structure is total. Every translation to *TLA+* validates the *invariants* of variables within domains.

We provide further automation, offering the optional generation of a series of sanity checks validated by both model checkers. That is, for all states s in one of the LLFSMs, the *LTL*

```
LTLSPEC (F s)
```

is automatically generated. Verifying such reachability properties for all states s seems intuitively sound.

For closed models, such intuition is correct. However, the property may be false for open models because the environment never provides an input driving the behaviour to s. Nevertheless, this outcome is informative. First, the designer can confirm that, indeed, the interactions that avoid s are always what is expected. Secondly, the model can be closed with an additional LLFSM that plays the role of the environment, resulting in a closed model.

For all states s that are the source of a transition, the following sanity check

```
LTLSPEC G (s -> F (!s))
```

can optionally be generated. Intuitively, if a state s is not terminal, then eventually, the behaviour must leave s. Although this sanity check may be false in open models, it is informative because it explicitly reveals which paths, by interacting with the environment, freeze the behaviour of the LLFSM at s.

Transitions are the dual of states. Thus if $T : s_s \to s_t$ is a transition in one of the LLFSMs of the arrangement, intuitively, T must eventually fire.

```
LTLSPEC F  (s & X t)
```

In *TLA+*, these properties must be expressed as *action properties* using the *globally* operator and not the *eventually* operator[3].

$$\neg\neg\Diamond(state = \text{“}s\text{”} \wedge state' = \text{“}t\text{”}) \quad \equiv \\ \neg\Box(state \neq \text{“}s\text{”} \vee state' \neq \text{“}t\text{”})$$

Then, we must use a *box-action formula* (Lamport, 2002).

$$\Box[state \neq \text{“}s\text{”} \vee state' \neq \text{“}t\text{”}]_state$$

Note that we verify the negation, and we expect the model checker to declare this formula invalid.

This offers another angle between `NuSMV` and `TLC`. The sanity checks with properties with an *eventually* form could be vacuously true. Thus, they should be complemented by ensuring that the negation is false and the model checker provides the trace. Although `TLC` does this, the output of false properties is lengthy and `TLC` (in command-line mode) halts when a property is found to be false.

A significant added value is that users can request the generation of sanity checks about the scheduling. For the round-robin deterministic scheduler, with n LLFSMs in the arrangement

[3]Note that *LTL*'s `G` (*globally/always*) is $\Box$ in *TLA+*, while *LTL*'s `F` (*eventually*) is $\Diamond$, `&` is $\wedge$, and `!` is $\neg$ in *TLA+*.

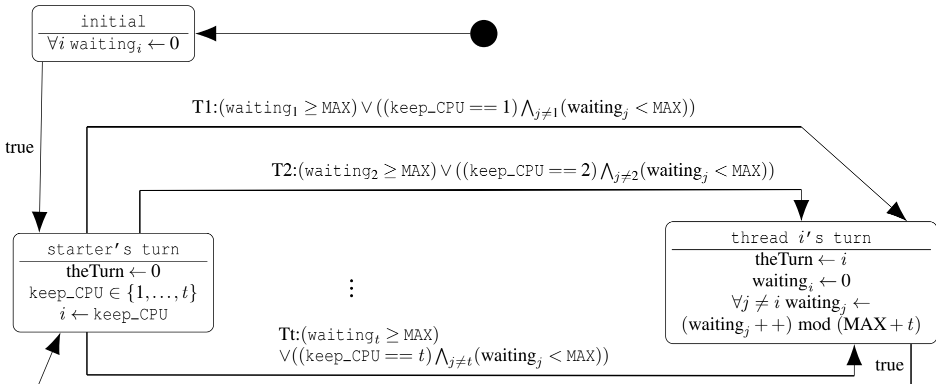


Figure 6: A fair scheduler encoded as an LLFSM.

```
LTLSPEC
G ( (turn=0 & X turn=1)
     ...
   | turn=t-2 & X turn=t-1)
   | turn=t-1 & X turn=0)
)
```

is the sanity-check.

7 CONCLUSION

Time-triggered architectures require careful tuning of time step frequencies. In event-driven software, such as web or GUI applications with simple user inputs, assumptions about time gaps between events typically hold. However, in cyber-physical systems, the large number of connected sensors drastically increases event frequency. *UML* statecharts have several features that reduce their understandability for developers, while LLFSMs present different but comparable issues (Estivill-Castro and Hexel, 2019).

By analysing schedulers for arrangements of LLFSMs we can now handle multi-threaded systems and distributed systems. We eliminated semantic gaps by ensuring that verification traces are consistent with executions in C or MIPS. There is a possibility that the translation implementation is faulty. However, regular comparison of the executable models (Lisp, MIPS, C) and using two model checkers reduces this risk. The next step is to formally verify the translation tool, which we consider feasible because each metamodel element follows a rule, similar to the recursive rules used in ATL transformations that can be verified by induction.

We keep a single representation and provide translations of such a representation to both programming and model-checker languages, thus, experts in one language need not become experts in others. Recall that programming languages differ significantly from the languages used by model checkers. "*As* TLA+ *is math-based, it comes with a difficult learning curve and might appear intimidating to software engineers*" (Caballar, 2023). Thus, a top priority is to "*translate a high-level* TLA+ *design directly into code*" (Caballar, 2023). We believe our approach contributes to the simultaneous development of executable code and simultaneous verification.

More translations beyond C, LISP, MIPS, *TLA+*, and SMV would be desirable, but such efforts can be performed with current translations as translations of reference. Such validation against all previous translations aids significantly in eradicating semantic gaps and errors from implementing new translations.

REFERENCES

André, E., Liu, S., Liu, Y., Choppy, C., Sun, J., and Dong, J. S. (2023). Formalizing UML state machines for automated verification – a survey. *ACM Comput. Surv.*, 55(13s).

Bellotti, M. (2019). Introduction to TLA+ model checking in the command line. Software Safety medium.com/software-safety. [Online; accessed 29-05-24].

Besnard, V., Brun, M., Jouault, F., Teodorov, C., and Dhaussy, P. (2018). Unified LTL verification and embedded execution of UML models. *21th ACM/IEEE*, MODELS, p. 112–122, NY, USA. ACM.

Brooks, R. (1990). The behavior language; user's guide. TR AIM-1227, MIT, Dep. Elec. CS.

Bucchiarone, A., Cabot, J., Paige, R. F., and Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13.

Caballar, R. D. (2023). TLA+ helps programmers squash bugs before coding. *IEEE Spectrum*.

Carrillo, M., Estivill-Castro, V., and Rosenblueth, D. A. (2020b). Verification and simulation of time-domain properties for models of behaviour. *Revised papers Int. Conf., MODELSWARD*, vol. 1361 *CCIS*, p. 225–249. Springer.

Carvalho, A. (2019). TLA+, Event B comparison. [Online; accessed 1-12-23].

Cengic, G. and Akesson, K. (2010). On formal analysis of IEC 61499 applications, part A: Modeling. *IEEE Trans. Industrial Informatics*, 6(2):136–144.

Estivill-Castro, V. (2021). Tutorial activity diagrams with Moka and unsafe race conditions. https://www.youtube.com/watch?v=P1KX2dBjmO8.

Estivill-Castro, V., Carrillo Barajas, M., and Rosenblueth, D. A. (2024). How to use `javallfsmstransformation` and its associated tools. https://mipal.net.au/Downloads/HowToUseM2T.pdf.

Estivill-Castro, V. and Hexel, R. (2019). The understandability of models for behaviour. *Revised papers 7th Int. Conf., MODELSWARD*, vol. 1161 *CCIS*, p. 50–75. Springer.

Furrer, F. (2019). *Future-Proof Software-Systems: A Sustainable Evolution Strategy*. Springer, Berlin.

Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., and Gérard, S. (2015). Executable modeling with fUML and Alf in Papyrus: Tooling and experiments. *1st Int. Workshop on Executable Modeling co-located with ACM/IEEE 18th MODELS*, vol. 1560 *CEUR*, p. 3–8.

Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. (2015). IronFleet: proving practical distributed systems correct. *25th Symp. on Operating Systems Principles*, SOSP, p. 1–17, NY, USA. ACM.

Konnov, I., Kuppe, M., and Merz, S. (2022). Specification and verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. *Leveraging Applications of Formal Methods, Verification and Validation*, p. 88–105, Springer.

Kröger, F. and Merz, S. (2008). Temporal logic and state systems. *Texts in Theoretical Computer Science. An EATCS Series*. Springer.

Kuppe, M. A. (2023). Teaching TLA+ to engineers at Microsoft. *Formal Methods Teaching Workshop*, LNCS, vol 13962, p. 66–81, Springer.

Kurshan, R. P. (2018). Transfer of model checking to industrial practice. *Handbook of Model Checking*, p. 763–793, Springer.

Kwon, G. (2000). Rewrite rules and operational semantics for model checking UML statecharts. *UML 2000 — The Unified Modeling Language*, LNCS, vol 1939, p. 528–540, Springer.

Lamport, L. (1987). A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11.

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, USA.

Lamport, L. (2005a). Real time is really simple. MSR-TR-2005-30, Microsoft Research, Dept. of Electronics and Computer Science.

Lamport, L. (2005b). Real-time model checking is really simple. *Correct Hardware Design and Verification Methods, 13th IFIP Advanced Research Working Conf., CHARME*, vol. 3725 *LNCS*, p. 162–175. Springer.

Lamport, L. (2024). A science of concurrent programs. version 7th June. https://lamport.azurewebsites.net/tla/science.pdf.

Latella, D., Majzik, I., and Massink, M. (1999). Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Form. Asp. Comput.*, 11(6):637–664.

Martínez, Y., Cachero, C., and Meliá, S. (2012). Evaluating the impact of a model-driven web engineering approach on the productivity and the satisfaction of software development teams. *Web Engineering, ICWE*, LNCS, vol 7387, p. 223–237, Springer.

McColl, C., Estivill-Castro, V., McColl, M., and Hexel, R. (2022a). Decomposable and executable models for verification of real-time systems. *Revised papers 9th Int. Conf., MODELSWARD*, vol. 1708 of *CCIS*, p. 135–156. Springer.

Merz, S. (2019). *Formal specification and verification*, p. 103–129. ACM, NY, USA.

Moreira, G., Vasconcellos, C., and Kniess, J. (2022). Fully-tested code generation from TLA+ specifications. *7th Brazilian Symp. Systematic and Automated Software Testing*, SAST ’22, p. 19–28, NY, USA. ACM.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73.

Nicolás, J. and Toval, A. (2009). On the generation of requirements specifications from software engineering models: A systematic literature review. *Information and Soft. Technology*, 51(9):1291–1307.

Niyogi, R. and Nath, A. (2024). Formal specification and verification of a team formation protocol using TLA+. *Soft.: Practice and Experience*, 54(6):961–984.

Patil, S., Dubinin, V., and Vyatkin, V. (2015b). Formal verification of IEC61499 function blocks with abstract state machines and SMV – modelling. *IEEE Trustcom/BigDataSE/ISPA*, vol. 3, p. 313–320.

Pham, V. C., Radermacher, A., Gérard, S., and Li, S. (2017). Complete code generation from UML state machine. *5th Int. Conf. Model-Driven Engineering and Software Development, MODELSWARD*, p. 208–219. SciTePress.

Posse, E. and Dingel, J. (2016). An executable formal semantics for UML-RT. *Softw. Syst. Model.*, 15(1):179–217.

Sahu, S., Schorr, R., Medina-Bulo, I., and Wagner, M. F. (2020). Model translation from Papyrus-RT into the nuXmv model checker. *Software Engineering and Formal Methods. SEFM*, vol. 12524 *LNCS*, p. 3–20. Springer.