States and Events in KandISTI
A Retrospective

Maurice H. ter Beek[1], Alessandro Fantechi[1,2], Stefania Gnesi[1], and Franco Mazzanti[1]

[1] ISTI–CNR, Pisa, Italy
{terbeek,gnesi,mazzanti}@isti.cnr.it
[2] University of Florence, Florence, Italy
alessandro.fantechi@unifi.it

Abstract. Early work on automated formal verification produced pioneering model-checking algorithms, in which system computations were modelled either as sequences of distinguished states in which the system evolves or as sequences of events or actions occurring during the system's state transitions. In both cases, automata-like structures generally known as transition systems were exploited to capture all possible computations, but still either state-based or event-based. Many years later, both views were combined in descriptions of computations as the evolution between distinguished states by means of transitions characterised by the occurrence of events, and verification tools were adapted to this more general setting. Meanwhile, the most important drive in improving verification tools concerned the complexity of models, which was attacked by algorithms capable of minimising the information needed for deciding the verification questions. One of the outcomes of this quest was local, on-the-fly model checking. Both of these lines of research, pioneered by Bernhard Steffen, are discussed in this paper in a general retrospective on state-based and event-based models of transition systems and temporal logics, followed by an overview of how this is exploited in the KandISTI model-checking environment.

1 Introduction

The development of expressive models of transitions systems that are capable of efficiently supporting formal verification by means of model-checking algorithms has been one of the concerns of Bernhard Steffen's career in research. The traditional model for the interpretation of modal and state-based logics, i.e. a Kripke structure [1], in which states are labelled by atomic propositions, was adopted by the early model-checking algorithms for CTL and LTL (cf. [2] and the references therein). On the other hand, Labelled Transition Systems (LTS), in which transitions instead are labelled with events, emerged as the most appropriate semantic model for process algebrae and process calculi [3,4]. In search for more expressivity and flexibility, the work by Bernhard Steffen and others has addressed models in which both states and transitions are labelled, such

as Doubly-Labelled Transition Systems [5], Kripke Transition Systems [6], and Labelled Kripke Structures [7].

It is well known that model checking is affected by the state-space explosion problem, for which realistic system models may require an exponential number of states (which may not fit the available computer memory). Or, as Cleaveland puts it in [8], "Consequently, while the best traditional model-checking algorithms [9,10,11,12] are linear in the number of states of a system, their applicability is severely restricted by the prohibitive number of states systems can have." Bernhard Steffen has made seminal contributions to the efficiency of model-checking algorithms [10,13]. To mitigate the state-space explosion problem, local, on-the-fly model-checking algorithms [14,15,16] can be of help. While these have the same worst-case complexity, they generally perform better in the many cases in which only a subset of the system states, generated 'on demand', needs to be analysed to determine whether a system model satisfies a formula. Local model checking moreover may provide results for infinite state spaces. Bernhard Steffen has made several important contributions also to this development (cf., e.g., [17,18]). In this paper, we list some models and logics that combine state and transition labelling and show how the KandISTI model-checking environment [19] and its rich logic, presented in this paper, exploit these features and thus relate to the aforementioned contributions of Bernhard Steffen.

KandISTI [3] is a family of model checkers developed at ISTI–CNR for over two decades now, which includes UMC [20], CMC [21], VMC [22], and FMC [23]. Each tool allows the efficient verification, by means of explicit-state on-the-fly model checking, of functional properties expressed in a state-based and event-based branching-time temporal logic, which builds upon the family of logics based on ACTL [24,25,26], i.e. action-based versions of CTL [27,9]. The KandISTI model checkers allow on-the-fly model checking with a complexity that is linear with respect to the size of the model and the size of the formula[4].

This paper is organised as follows. Sections 2 and 3 introduce transition system models and temporal logics, respectively, that explicitly combine state-based and event-based information. Section 4 discusses how KandISTI exploits states and events in a rich modelling and verification environment based on a comprehensive temporal logic, and highlights some of its more interesting features. Section 5 concludes the paper.

2 Modelling structures for reasoning on both state-based and event-based properties

In the literature, one can find several variants of graph structures that have information associated with both their nodes and their edges, used as models for state/event-based logical specifications.

[3] Available online at http://fmt.isti.cnr.it/kandisti

[4] When ignoring the fixed point operators and the parametric aspects of the logic.

One of the first structures that comes to mind is the one adopted for the propositional μ-calculus [28]. These models are constituted by a set of states, a set of *propositional constants* and a set of *program constants*. From a semantic point of view, the interpretation of a propositional constant is a set of states. Therefore each (control) state might have several state labels. The interpretation of a program constant, instead, is a transition relation (i.e. edges associated with exactly one label).

In the *Doubly-Labelled Transition Systems (L^2TS)* introduced by De Nicola and Vaandrager [5], the same concept was reshaped by explicitly assigning to each state a set of *atomic propositions*, and by describing the (now unique) transition relation as a set of triples of the form ⟨ source state, observable or silent *event*, target state ⟩. No constraints are explicitly imposed on the finiteness or absence of internal structure of atomic propositions and events.

Lawford, Ostroff and Wonham [29] introduced so-called *State-Event Labeled Transition Systems (SELTS)*, which are equivalent to the underlying model of the state/event systems of Graf and Loiseaux [30], in which a model is described by a countable set of states, a finite set of binary relations on the states, an initial state, and a mapping from the states to sets of atomic predicates (i.e. edges are still associated with precisely one label).

In 1999, together with Müller–Olm and Schmidt, Bernhard Steffen coined the term *Kripke Transition System (KTS)* [6]. In a KTS, states are labelled with sets of *atomic propositions* and transitions are labelled with sets of *events*. No constraint is imposed on the absence of internal structure of the labels, nor on the totality of the transition relation, and the presence of an explicit initial state is allowed (i.e. *rooted* structures). The authors point out that edge labellings can be encoded by node labellings and vice versa, such that theoretical analyses typically study one form of labelling. Nevertheless, we very much agree with their motivation for introducing KTS: "For modeling purposes, however, it is often natural to have *both* kinds of labeling available."

In 2004, Chaki et al. introduced *Labelled Kripke Structures (LKS)* [7], which are characterised by a finite set of states, an initial subset of states, a finite set of atomic state propositions, a finite set of events and a binary transition relation among states. The transition relation is no longer required to be total. A state-labelling function associates each state with a set of state propositions, and a transition-labelling function associates each pair of ⟨ source, target ⟩ states with a set of events (i.e. we cannot have two transitions between the same two states with different labellings).

In 2006, Pecheur and Raimondi use *Mixed Transition Systems* [31], not to be confused with Larsen's Modal Transition Systems [32,33,34], to denote a generalisation of both state-based models (Kripke structures) and action-based models (LTS) into a common super-structure very similar to L^2TS, which is characterised by a set of states (a subset of which can be qualified as initial states), a transition relation defined as a set of triples of the form ⟨ source state, event, target state ⟩ and two interpretation functions that associate each state and event with a set of *propositional atoms* over states and events, respectively.

3 Temporal logics for reasoning on both state-based and event-based properties

As already apparent form the previous section, state- and event-based models have been proposed often together with specific temporal logics having those models as interpretation structures.

We already mentioned the propositional μ-calculus [28], which is an extension of modal logic with propositions and fixed point operators [35]. Atomic propositions can be satisfied by single states. Modal operators are indexed by events that label the transitions. Fixed point operators are then introduced to extend the meaning of logic formulae over full, possibly infinite, computations.

Next to the Boolean constants *false* and *true*, the μ-calculus contains *atomic propositions*, logical connectives and the *diamond* and *box* operators $\langle\,\rangle$ and $[\,]$ of modal logic. The *least* and *greatest fixed point* operators μ and ν provide recursion used for 'finite' and 'infinite' looping, respectively.

Kindler and Vesper [36] introduced the *Event-and-State-based Temporal Logic (ESTL)* to reason over events and states of Petri nets, which are a typical example of a formal model for reasoning over states (places) and events (transitions). ESTL is a linear-time logic based on four basic temporal operators, namely *eventually* and *once* (eventually in the past), working on state properties, and *sometime* and *sometime in the past*, working on transition properties. From these operators, four dual operators called *always*, *so far*, *every-time* and *every-time in the past* can be derived. We refer to [36] for their precise meaning.

Also the logic interpreted over the LKS introduced in [7], called SE-LTL, is a linear-time logic. This logic is based on the X (*next*), G (*always*), F (*eventually*) and U (*until*) linear-time operators, which can be applied both to state and to transition properties.

The Mixed Transition Systems introduced in [31] serve as interpretation model for the *Action-Restricted CTL (ARCTL)* logic, which extends CTL but is less expressive than ACTL from [24]. In fact, ARCTL is instead a branching-time logic over mixed state/event models introduced as a generalisation of CTL. ARCTL has the same temporal operators as CTL, except that they can be restricted to paths whose actions satisfy a given action formula.

Among the various state- and event-based logics proposed in the literature, UCTL [20] was designed to include both the branching-time action-based logic ACTL [24,25] and the branching-time state-based logic CTL [27,37], with the aim to reason over UML state diagram specifications and L^2TS. The logic UCTL is *adequate* with respect to strong bisimulation equivalence on L^2TS [38]. Adequacy [39], as also investigated by Bernhard Steffen in [40], means that two L^2TS A_1 and A_2 are strongly bisimilar if and only if $F_1 = F_2$, where $F_i = \{\, \psi \in \mathit{UCTL} \mid A_i \models \psi \,\}$ for $i = 1, 2$. In other words, adequacy implies that if there is a formula that is not satisfied by one of the L^2TS but satisfied by the other L^2TS, then the two L^2TS are not bisimilar, and—on the other hand—if two L^2TS are not bisimilar, then there must exist a distinguishing formula.

The UCTL logic initially was supported by the UMC v3.3 [5] model checker, which later evolved into the KandISTI family of model checkers, as explained in the next section.

4 Exploiting states and events in KandISTI

In this section, we first introduce the KandISTI tool and we show how it exploits states and events in a rich modelling and verification environment, based on a comprehensive temporal logic interpreted over L^2TS, after which we discuss some of its more interesting features in more detail.

4.1 KandISTI

For over more than two decades, we are developing the KandISTI family of model checkers, each one based on a different specification language, but all sharing a common temporal logic and verification engine. The main objective of KandISTI is to provide formal support in the design phase of a software system, especially in its early stages, i.e. when a design is still likely to be incomplete and contain mistakes. The main features of KandISTI focus on the possibility to (i) explore the evolution of a system and generate a summary of its behaviour; (ii) investigate abstract system properties using a temporal logic supported by an on-the-fly model checker; and (iii) obtain a clear explanation of the model-checking results, in terms of possible evolutions of the specific specification model.

While the specification models supported by KandISTI are rather different, ranging from UML statecharts to various process algebrae, its verification engine is unique and based on a common temporal logic which encompasses the specific logics initially associated to the specific tools: ACTL for FMC, UCTL for UMC, SocL for CMC and v-ACTL for VMC. This is feasible by separating state-space generation (which depends on the underlying specification model) from L^2TS analysis, and by the introduction of an explicit abstraction mechanism that allows to specify the details of the model that should be observable as labels on the states and transitions of the L^2TS. Another essential characteristic of KandISTI is the on-the-fly structure of the model-checking algorithm: the L^2TS corresponding to the specification model is generated on demand, following the incremental needs of the verification engine. Given a state of an L^2TS, the validity of a logic formula on that state is evaluated by analysing the transitions allowed in that state, and by analysing the validity of the necessary sub-formulae possibly in some of the necessary next reachable states, and all this recursively.

Hence, each tool consists of two separate, interacting components: a tool-specific L^2TS generator engine and a common logical verification engine. The L^2TS generator engine is again structured in two components: a ground evolutions generator, strictly based on the operational semantics of the specification

[5] Still available online at http://fmt.isti.cnr.it/umc/legacy/V3.3

language, and an abstraction mechanism which allows to associate abstract observable events to system evolutions and abstract atomic propositions to the system states. The overall structure of KandISTI is depicted in Fig. 1.

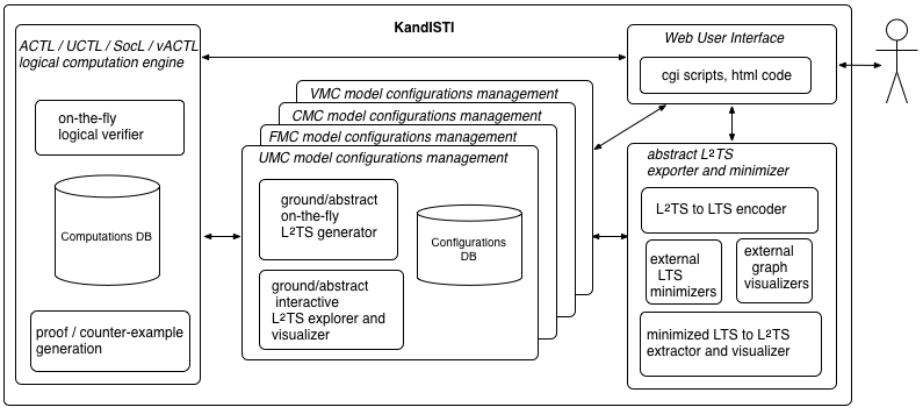


Fig. 1. The architecture of the KandISTI framework (from [41])

All KandISTI model checkers offer a downloadable command-line version of the tool as well as an online GUI through http://fmt.isti.cnr.it/kandisti. Detailed descriptions of the model-checking algorithms and architecture underlying KandISTI are beyond the scope of this paper, but they can be found in [42,41,20,21,43].

4.2 Modelling with KandISTI

The structure of the models underlying the KandISTI framework (still called L^2TS) is very similar to the KTS of Bernhard Steffen and colleagues and to the L^2TS of De Nicola and Vaandrager, in the sense that both states and transitions can be labelled with finite sets of predicates or events, and a unique initial state is explicitly required. None of the domains of states, predicates and events is required to be finite, and a matching function is required to evaluate whether an event expression or state predicate is satisfied by the set of labels associated to the states or transitions.

Very few model-checking tools provide support for sets of structured labels associated with the edges of a model's evolution graphs. KandISTI, for what we know, is the only publicly available framework that supports this. The tool of the KandISTI framework that better allows to exploit the doubly-labelling feature is UMC. In UMC, a model describes the possible evolutions of a set of UML-like state machines. The state labels of the abstract model contain the relevant state information that we want to observe (typically the values of a subset of the local variables of the state machines), while the transition labels contain the relevant information that we want to observe concerning the occurrence of events during system evolution.

The KandISTI framework allows an abstract view (in terms of an L^{2}TS) to be associated with the basic operational model of the specification language. So-called "abstraction rules" need to be defined by the user to associate a set of abstract observable (composite) state and event predicates with relevant states and transitions, hiding in the abstract view all other details. This abstract view of the system model is the one used during verification, while all the internal details of the traversed states and transitions remain available during the exploration of the model or the analysis of a counterexample. Figure 2 shows an example of an L^{2}TS associated with an UML model once the desired abstractions have been applied.

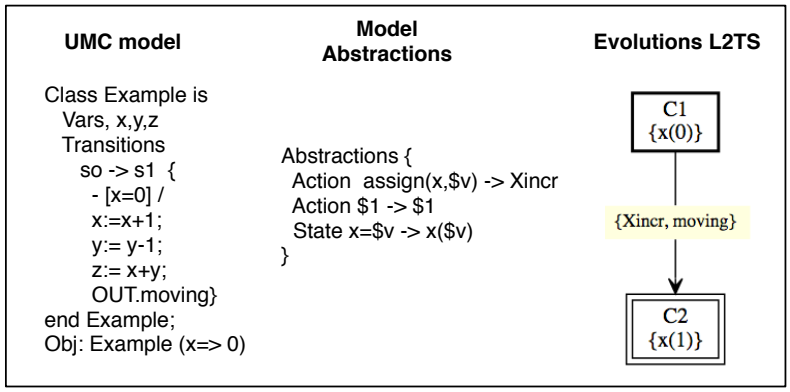


Fig. 2. From UMC model + abstractions to L^{2}TS

4.3 Verification with KandISTI

Figure 3 provides the syntax of the logic supported by the KandISTI framework. It encompasses the various logics of the individual model-checking tools, ranging from UCTL (cf. Section 3) to the most recent addition, v-ACTL, for the analysis of so-called Modal Transition Systems with variability constraints (MTSυ) [43]. The logic of KandISTI includes the following rich set of features:

- Parametric state predicates (represented by the state labels of the L^{2}TS), e.g. $pred1(arg1, arg2)$, $pred2$, and $pred3(*, arg3)$, where $*$ means 'don't care'.
- Parametric event formulae (represented by Boolean expressions over the transition labels (events) of the L^{2}TS), e.g. $(act1(arg1, arg2) \vee act2)$ and $\neg\, act3(arg3, *, *)$.
- Classical diamond and box modalities from Hennessy–Milner logic [44], e.g. $[act1]\ (\ pred1 \rightarrow \langle act2 \rangle\ true\ )$.
- Classical high-level CTL operators (e.g. *next*, *always*, *eventually*, *globally*, *until*, and *weak until*) in their state-based, action-based as well as mixed

KandISTI logic

State predicates

$$
\begin{aligned}
p &::= \ell \mid \ell(e,\ldots) \\
\ell &::= id \\
e &::= val \mid * \mid \%var
\end{aligned}
$$

Event formulae

$$
\begin{aligned}
\chi &::= true \mid false \mid \ell \mid \ell(e,\ldots) \mid \tau \mid \neg\chi \mid \chi \wedge \chi' \mid \chi \vee \chi' \\
\ell &::= id \mid * \mid \$var \mid \%var \\
e &::= val \mid * \mid \$var \mid \%var
\end{aligned}
$$

State fomulae

$$
\begin{aligned}
\phi &::= rel \mid true \mid false \mid P \mid (\phi) \mid \neg\phi \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \phi \rightarrow \phi' \mid \\
&\quad \langle\chi\rangle\phi \mid \langle\chi\rangle^{\Box}\phi \mid [\chi]\phi \mid [\chi]^{\Box}\phi \mid E\pi \mid A\pi \\
rel &::= \%var\ relop\ \%var \mid \%var\ relop\ val \\
relop &::= \leq \mid < \mid = \mid \neq \mid > \mid \geq
\end{aligned}
$$

Path fomulae

$$
\begin{aligned}
\pi ::= {} & X_{\chi}\,\phi \mid X^{\Box}_{\chi}\,\phi \mid [\phi\ {}_{\chi}U_{\chi'}\ \phi'] \mid [\phi\ {}_{\chi}U\ \phi'] \mid [\phi\ {}_{\chi}W_{\chi'}\ \phi'] \mid [\phi\ {}_{\chi}W\ \phi'] \mid \\
& F\phi \mid F_{\chi}\,\phi \mid F^{\Box}\phi \mid F^{\Box}_{\chi}\phi \mid G\phi \mid G_{\chi}\,\phi \mid G^{\Box}\phi \mid G^{\Box}_{\chi}\phi
\end{aligned}
$$

$\%var$ denotes a bound variable, whereas $\$var$ denotes a free variable, and it may only occur inside certain contexts (viz. next, diamond, box, eventually, and on the right side of the until operators) but not inside Boolean disjunctions or negations of event formulae.

Fig. 3. Full syntax of the KandISTI logic. Actually, the logic of the KandISTI framework supports also (not optimised) versions of the least and greatest fixed-point operators μ and ν from the μ-calculus (cf. Sect. 3), to be written as `min` and `max`, respectively.

modality flavours, e.g. $EX\,pred1$, $A\,[pred1(arg1)\ U\ pred2]$, $AG\,EF\,pred1$, and $E\,[pred1(arg1)\ W\ pred2]$.

- High-level ACTL-like operators (i.e. action-based variants of above CTL operators), e.g. $EX_{act1}\,true$, $A\,[pred1(arg1)\ {}_{act1}U_{act2}\ pred2]$, $AG\,EF_{act1}\,pred1$, and $E\,[\,pred1(arg1)\ {}_{act1}W\ pred2\,]$.
- Parametric formulae expressing data correlations among actions and subformulae, e.g. $[act1(\$1,\$2)]\ AF_{act2(\%1,\%2)}\,true$ and $EF_{\$1}\ EF_{\%1}\,true$.
- Deontic variants of some of the above operators (which allow to reason on classical Modal Transition Systems (MTS) [32,43,34], whose transitions are partitioned into mandatory and optional transitions), e.g. $\langle act1\rangle^{\Box}\,true$ and $EF^{\Box}_{act1}\,pred1$.

– Special-purpose predefined state predicates, e.g. $PRINT(msg, arg1, arg2)$ (prints the current state and the message *msg* each time it is evaluated), $DEPTH\_LT\_n$ (returns *TRUE* if when evaluated the current evaluation depth is less than n), and *FINAL* (shorthand for a final state).

The latter category of special-purpose predefined state predicates allows a better control and understanding of the ongoing evaluation process. Indeed, model checking is a technique that can be used for a variety of goals. On one side we have pure validation of a system design which is supposed to be correct with a high probability, as a final result of a development phase. In this case, the design of the verification tools is often focussed on techniques that contrast the state-space explosion problems (e.g. minimising memory requirements), often at the expense of a clear, easily understandable explanation when the validation fails.

On the opposite side we have the goal of an easy but exhaustive analysis/debugging of an initial (likely wrong) design. In this case, the focus of the tool can be more oriented to the collection and preservation of all the diagnostic information that might be useful to explain a negative result, even at the cost of an increased or less efficient usage of the resources.

Our KandISTI framework falls in this second class of verification environments. During the (on-the-fly) evaluation process all the local information of the generated states and transitions is preserved, to be eventually used when an explanation of the evaluation result is requested. The exploitation of this approach is made possible by the *lazy, left-to-right* evaluation approach for Boolean operators, and the *top-down* evaluation process with respect to the formula structure.

In the KandISTI framework, the logical verification engine shared by all the tools observes the underlying model as an abstract L^2TS. This L^2TS is independent from the operational semantics of the particular specification language adopted by the various tools, thanks to the intermediate set of abstraction rules associated to the specification itself. We do not provide the full semantics in this paper, but instead refer to its exhaustive (incremental) treatment in [20,21,43].

We note that not all KandISTI model checkers are able to fully exploit all features of the logic. For instance, VMC and FMC specifications do not support state labelling (and therefore neither state predicates), whereas variability-related aspects (e.g. the deontic 'boxed' operators) are fully supported only by VMC specifications (but partially supported by FMC and UMC specifications).

The actual usage of the logic in the KandISTI framework exploits a machine-friendly, ASCII-only, syntax. In particular, the silent event τ must be written as `tau`; the propositional connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$ must be written as `not` (or $\sim$), `and` (or `&` or `&&`), `or` (or `|` or `||`), and `implies`; the relational operators $\leq$, $\neq$, and $\geq$ must be written as `<=`, `!=` (or `\=`), and `>=`, respectively (and $=$ may also be written as `==`); the 'boxed' variants $\langle\chi\rangle^{\Box}$, $[\chi]^{\Box}$, $X^{\Box}$, $F^{\Box}$, and $G^{\Box}$ of the modal and temporal operators $\langle\chi\rangle$, $[\chi]$, X, F, and G, respectively, must be written by appending `#` to the operators (e.g. `<>#` and `F#`); finally, the event-based variants of the temporal operators U, W, X, F, and G must be written

by (prefixing and) suffixing the operators with the event formulae between curly brackets (e.g. `{e1} U {e2}` and `X {e}`).

In the following sections, we focus in detail on two particular features that have allowed KandISTI to cope with specialised formal verification tasks.

4.4 Variable binding

In certain cases, it is useful to express the fact that an event expression that appears in a formula can make use of variable names (e.g. $\$var$), which can either be free variables or variables bound to a value by a previous binding operator in the same formula. This data extraction feature from transition labels can be found also in other μ-calculus-based languages, like for example MCL [45].

The contexts in which a variable name is allowed to appear are only the next operator X, the diamond and box operators $\langle\,\rangle$ and $[\,]$, the *eventually* operator F, and on the right side of the *(weak) until* operators W and U. Moreover, in these contexts, the event expression can only have the form of a *basic event predicate*, or a *conjunction* of basic event predicates, and the variable name can only appear in the place of the *event name* or the place of an *event argument*. Here are some examples of legal occurrences of variable names:

$$[\,\$event\,]\ldots,\quad \langle aa(\$1,\$2)\rangle\ldots,\quad E[\ldots U_{\$event(\$var,123)}\ \ldots],$$
$$EF_{event(\$var)\wedge\neg event(11)}\ \ldots,\quad AF_{\$var}\ \ldots$$

When such an event expression is evaluated with respect to a set of transition labels, if the expression matches the labels, then a set of variable bindings occurs, and the obtained bound values can be referred inside the subsequent part of the formula by using the $\%var$ notation. Let us consider the L^2TS shown in Fig. 4a.

With the following formula we can express the property that along any path, any event may occur at most once (the formula is true in the L^2TS of Fig. 4a).

$$AG\,[\$event]\ \neg EF_{\%event}$$

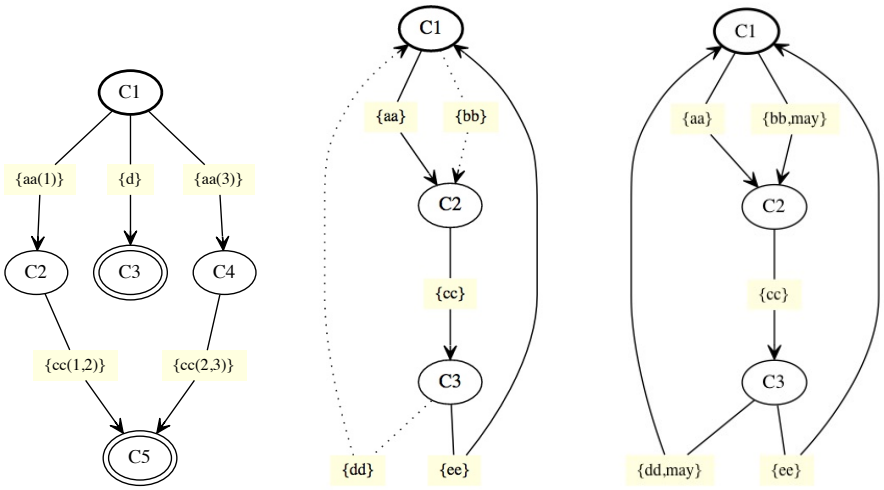
The next formula, instead, states that whenever an event of the form *cc*(*arg1*, *arg2*) occurs, its arguments differ (the formula is true in the L^2TS of Fig. 4a).

$$AG\,[cc(\$1,\$2)]\,(\%1 \neq \%2)$$

The following formula states the existence of a path in which an *aa* event with one argument is always eventually followed by a *cc* event with two arguments, where the second argument of *cc* is equal to the first argument of *aa* (again, the formula is true in the L^2TS of Fig. 4a).

$$EF_{aa(\$1)}\ AF_{cc(*,\%1)}$$

Finally, below formula, instead, expresses that for all the transitions that contain the event *aa* with an argument that is different from the value 3 lead to a state from which it is possible to perform a *cc* event with two arguments, of which the



(a) A sample L^2TS **(b)** A sample MTS and its L^2TS encoding

Fig. 4. Sample L^2TS and MTS

first one is equal to the argument of *aa* and the second one is greater than the first one (also this formula is true in the L^2TS of Fig. 4a).

$$[aa(\$1) \wedge \neg aa(3)]\ \langle cc(\%1, \$2)\rangle\ (\%1 < \%2)$$

Note that this formula might have been encoded in an equivalent way as follows.

$$[aa(\$1)]\ (\ (\%1 \neq 3) \rightarrow \langle cc(\%1, \$2)\rangle\ (\%1 < \%2)\ )$$

Note that the presence of the bound value notation $\%var$ introduces also the possibility of a new class of basic state predicates that have the form of a simple relation, where a bound value is compared with another bound value or literal.

4.5 MTS model checking

The VMC, UMC, and FMC tools of the KandISTI framework exploit another interesting use of the composite labelling of a model's transitions. In this case, the model is defined by a sequential algebraic process, and the first parameter of the events, if corresponding to the "may" literal, indicates the optionality of the corresponding evolution. This allows a direct encoding of an aforementioned MTS as an L^2TS, using the additional "may" label to denote the deonticity of the evolution. When displayed to the user (cf. Fig. 5), the corresponding graphical view of the L^2TS simply removes the optional "may" labels and shows this information via a dashed representation of the transition edge.

One of the purposes of MTS is to describe families of implementations, where edges may be associated with an 'optional' flavour that explicitly pinpoints the

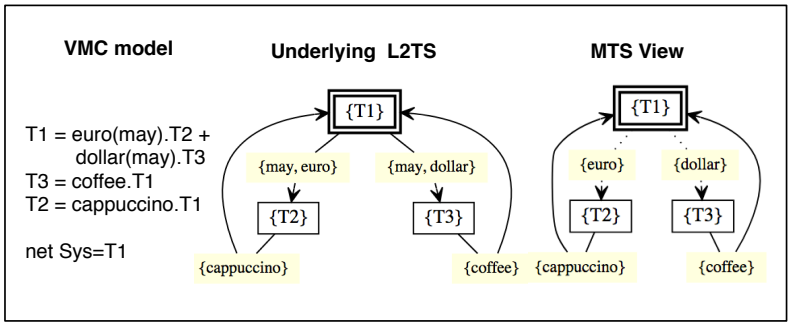


Fig. 5. From VMC model to L^2TS (MTS)

variability allowed among the possible implementation variants. In Fig. 4b, we show an example of an MTS and its L^2TS encoding, which will be used to show the way in which our KandISTI logical engine allows to reason on this kind of systems. Figure 6 depicts the four implementation variants that constitute the family represented by the MTS of Fig. 4b.

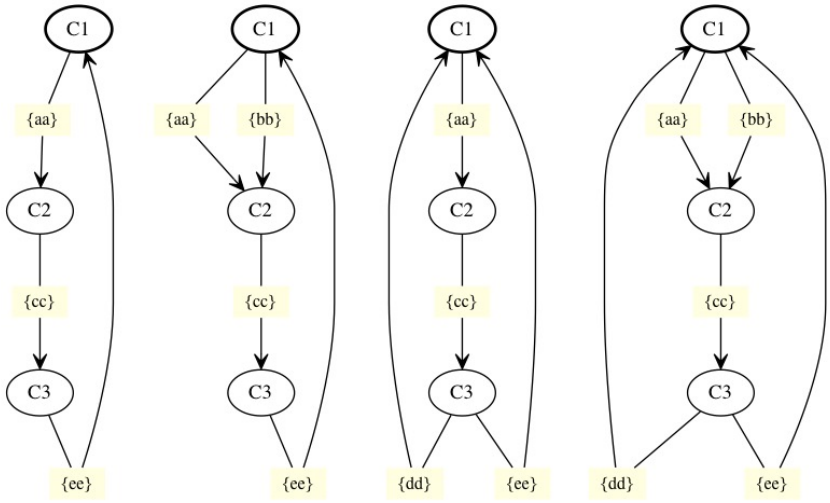


Fig. 6. All four implementation variants of the MTS of Fig. 4b

Now suppose we try to evaluate the formula $EX_{bb}\, true$ on the MTS/L^2TS of Fig. 4b. The formula will appear to be satisfied by the MTS because actually there is an initial transition that satisfies the event expression bb. However, it is also clear that it is not true that this formula holds for all the MTS variants. This means that a $TRUE$ result returned by the EX_{act} operator on an MTS, might in general not be preserved by all the implementation variants of the MTS.

Note, instead, that a negation of a next operator that returns a *FALSE* result is indeed preserved by all the allowed variants (i.e. $EX_{cc}\, true$ is does not hold on the MTS and neither on all its implementation variants). If we want to verify the existence of a next transition in all the variants, by checking a formula on the MTS, e.g. the existence of an initial *aa* transition, then we should verify the following formula.

$$EX_{aa \wedge \neg may}\, true$$

The KandISTI logic allows to simplify the writing of formulae like the above (making use of implicit $\ldots \wedge \neg may$ event expressions) by offering 'boxed' versions of most temporal operators. The above formula can hence be written as follows.

$$EX^{\Box}_{aa}\, true$$

The temporal operators for which such 'boxed' versions are supported in KandISTI are $\langle \chi \rangle^{\Box}$, $[\chi]^{\Box}$, $EX^{\Box}, EX^{\Box}_{\chi}$, $EF^{\Box}$, $EF^{\Box}_{\chi}$, $AF^{\Box}$, and $AF^{\Box}_{\chi}$ (cf. Fig. 3).

When a formula is satisfied by the MTS and its structure guarantees that the *TRUE* result is preserved by the MTS variants, then the model checker VMC notifies this fact to the user. For example, if we evaluate (on the MTS of Fig. 4b) the formula $AG\ EF^{\Box}_{cc}\, true$, the result will be as shown in Fig. 7.

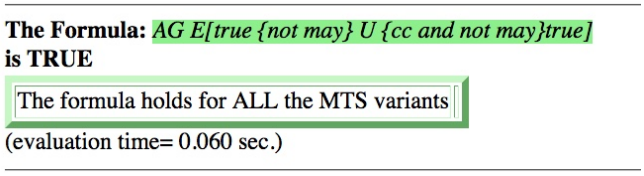


Fig. 7. Successful evaluation of $AG\ EF^{\Box}_{cc}\, true$

The following are some exemplary formulae that are satisfied by the MTS of Fig. 4b and preserved by all variants depicted in Fig. 6:

$EX^{\Box}_{aa}\, true$	*an initial mandatory aa transition exists*
$AG\ EF^{\Box}_{cc}\, true$	*from any state there is a mandatory path to cc*
$[bb]\, \langle cc \rangle^{\Box}\, true$	*an initial bb transition, if present, is followed by cc transition*
$\neg \langle cc \rangle\, true$	*no initial cc transition exists*
$AG\, \langle true \rangle^{\Box} true$	*in any state, at least one mandatory transition is possible*

The general rule, proved in [43], is that a *TRUE* result of any of the operators

$$\langle\chi\rangle^{\Box},\ [\chi],\ EX^{\Box},\ EX_{\chi}^{\Box},\ EF^{\Box},\ EF_{\chi}^{\Box},\ AF^{\Box},\ AF_{\chi}^{\Box},\ AG$$

is preserved by all the variants when appearing in a context without negations (or under an even number of negations), whereas a *FALSE* result of the operators

$$\langle\chi\rangle,\ [\chi]^{\Box},\ EX,\ EX_{\chi},\ EF,\ EF_{\chi},\ AF,\ AF_{\chi}$$

is preserved by all the variants when appearing in a context under an odd number of negations.

If we observe closely the MTS of Fig. 4b, we immediately see that it satisfies a particular property, namely that all its nodes are the source of at least one mandatory (i.e. not labelled with *may*) transition. A node that satisfies this property or which is final (i.e. without outgoing edges) is called *live* and an MTS is called *live* if all its nodes are. Under these circumstances, we have the additional property that also AF and AF_{χ} formulae, if *TRUE*, preserve their validity in all the implementation variants [43].

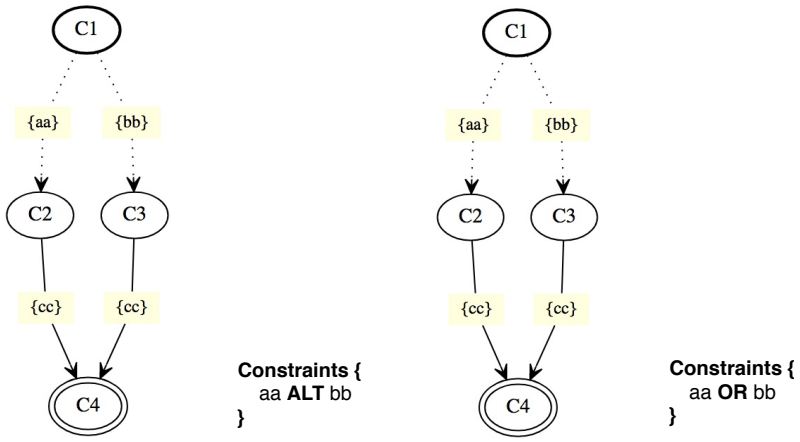
For example, we can verify that the MTS of Fig. 4b (and therefore all its variants depicted in Fig. 6) satisfies the property that any path from any state (in any variant) will eventually and necessarily reach a *cc* event. The property can be expressed by the following formula.

$$AG\ AF_{cc}\ true$$

One of the tools of the KandISTI framework, namely the variability model checker VMC [22,46], is explicitly tailored for the verification of behavioural models of so-called (software) product families in the form of MTS with variability constraints (MTSυ) [43]. One of the particular features supported by VMC is the possibility to express variability constraints that allow to fine-tune the set of valid implementation variants, and in particular allow to extend further the notion of *live* nodes.

Let us consider the MTSυ shown in Fig. 8a. The constraint *aa* ALT *bb* allows to specify that we consider as valid variants (products) of the MTSυ only those variants that either have the *aa* event or the *bb* event, but not both of them, nor none of them (i.e. equivalent to a logical xor). Therefore there exist precisely two valid implementations, for both of which the formula $AF_{cc}\ true$ holds. This property can be checked directly on the MTSυ, because the specified variability constraint has the effect of transforming the C1 node into a *live* node.

The second constraint *aa* OR *bb*, instead, allows to specify that we consider as valid variants (products) of the MTSυ only those variants that have either the *aa* event or the *bb* event, and possibly both of them, but not none of them (i.e. equivalent to a logical or). In this case, we end up with three valid LTS variants, and the formula $AF_{cc}\ true$ continues to hold. Also in this case the effect of the variability constraint is to change the status of the node C1 into a *live* node, thus allowing the verification of the above formula directly on the MTSυ with the guarantee the *TRUE* result is preserved by all the valid variants.



(a) A sample MTSυ **(b)** A sample MTSυ

Fig. 8. Sample MTSυ with different variability constraints

5 Conclusion

The KandISTI family of model checkers fully exploits the expressive power of the underlying L^{2}TS models. The framework plays the role of an experimental workbench, targeted mainly at teaching and research activity, without having in mind verification efficiency as its major aim.

The capability to navigate the state space both at the concrete and at an abstract level, together with useful debugging-oriented tools allow easy but exhaustive analysis/debugging of an initial (likely wrong) system design: in such cases, the focus of the tool is oriented to the collection and preservation of all the information that might be useful to explain a negative result, even at the cost of an increased or less efficient usage of the resources. Indeed, during the (on-the-fly) evaluation process all the local information of the generated states and transitions is preserved, to be possibly used once an explanation of the evaluation result is requested. Moreover, a small set of basic state predicates is defined, which allows to better control and understand the ongoing evaluation. The exploitation of this approach is made possible by the *lazy, left-to-right* evaluation approach for Boolean operators and the *top-down* (with respect to the formula structure and initial root state) evaluation process.

The characteristics of the KandISTI framework outlined in this paper have favoured its use in numerous exploratory studies, such as those in [47,48] (intelligent domotic environments), [49,50,51] (deadlock avoidance in train scheduling), [52] (distributed railway interlocking concept) and [53] (web-based communication interworking). The versatility of its underlying L^{2}TS models moreover allowed to map rich logics developed in the context of trust and reputation systems, like the so-called *trust temporal logic* originally defined over *trust LTS*, onto UCTL [54,55]. Finally, KandISTI is much appreciated as an effective teaching tool by students at the University of Florence.

References

1. Kripke, S.A.: Semantical Considerations on Modal Logic. Acta Phil. Fennica **16**(1963), 83–94 (1963)
2. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
3. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
4. Baeten, J.C.M., Weijland, W.P.: Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press (1990)
5. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995)
6. Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model-Checking: A Tutorial Introduction. In: Proceedings 6th International Symposium on Static Analysis (SAS'99). LNCS, vol. 1694, pp. 330–354. Springer (1999). https://doi.org/10.1007/3-540-48294-6_22
7. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Proceedings 4th International Conference on Integrated Formal Methods (IFM'04). LNCS, vol. 2999, pp. 128–147. Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_8
8. Cleaveland, R.: Pragmatics of model checking: an STTT special section. Int. J. Softw. Tools Technol. Transf. **2**(3), 208–218 (1999). https://doi.org/10.1007/s100090050030
9. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986). https://doi.org/10.1145/5397.5399
10. Cleaveland, R., Steffen, B.: A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. Form. Method. Sys. Design **2**(2), 121–147 (1993). https://doi.org/10.1007/BF01383878
11. Queille, J., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: Proceedings 5th International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer (1982). https://doi.org/10.1007/3-540-11494-7
12. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Proceedings Symposium on Logic in Computer Science (LICS'86). pp. 332–344. IEEE (1986)
13. Cleaveland, R., Klein, M., Steffen, B.: Faster Model Checking for the Modal Mu-Calculus. In: Proceedings 4th International Workshop on Computer Aided Verification (CAV'92). LNCS, vol. 663, pp. 410–422. Springer (1993). https://doi.org/10.1007/3-540-56496-9_32
14. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient On-the-Fly Model Checking for CTL*. In: Proceedings 10th Symposium on Logic in Computer Science (LICS'95). pp. 388–397. IEEE (1995). https://doi.org/10.1109/LICS.1995.523273
15. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Program. **46**(3), 255–281 (2003). https://doi.org/10.1016/S0167-6423(02)00094-1
16. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
17. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Proceedings 3rd International Conference on Concurrency Theory (CONCUR'92). LNCS, vol. 630, pp. 123–137. Springer (1992). https://doi.org/10.1007/BFb0084787

18. Hungar, H., Steffen, B.: Local model checking for context-free processes. In: Proceedings 20th International Colloquium on Automata, Languages and Programming (ICALP'93). LNCS, vol. 700, pp. 593–605. Springer (1993). https://doi.org/10.1007/3-540-56939-1_105
19. ter Beek, M.H., Gnesi, S., Mazzanti, F.: From EU Projects to a Family of Model Checkers: From Kandinsky to KandISTI. In: Software, Services and Systems, LNCS, vol. 8950, pp. 312–328. Springer (2015). https://doi.org/10.1007/978-3-319-15545-6_20
20. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Sci. Comput. Program. **76**(2), 119–135 (2011). https://doi.org/10.1016/j.scico.2010.07.002
21. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A logical verification methodology for service-oriented computing. ACM Trans. Softw. Eng. Methodol. **21**(3), 16:1–16:46 (2012). https://doi.org/10.1145/2211616.2211619
22. ter Beek, M.H., Mazzanti, F., Sulova, A.: VMC: A Tool for Product Variability Analysis. In: Proceedings 18th International Symposium on Formal Methods (FM'12). LNCS, vol. 7436, pp. 450–454. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_36
23. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Using FMC for Family-Based Analysis of Software Product Lines. In: Proceedings 19th International Software Product Line Conference (SPLC'15). pp. 432–439. ACM (2015). https://doi.org/10.1145/2791060.2791118
24. De Nicola, R., Vaandrager, F.W.: Action versus State based Logics for Transition Systems. In: Semantics of Systems of Concurrent Processes. LNCS, vol. 469, pp. 407–419. Springer (1990). https://doi.org/10.1007/3-540-53479-2_17
25. De Nicola, R., Fantechi, A., Gnesi, S., Ristori, G.: An Action Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. In: Proceedings 3rd International Workshop on Computer Aided Verification (CAV'91). LNCS, vol. 575, pp. 37–47. Springer (1991). https://doi.org/10.1007/3-540-55179-4_5
26. Fantechi, A., Gnesi, S., Mazzanti, F., Pugliese, R., Tronci, E.: A Symbolic Model Checker for ACTL. In: Proceedings International Workshop on Current Trends in Applied Formal Methods (FM-Trends'98). LNCS, vol. 1641, pp. 228–242. Springer (1998). https://doi.org/10.1007/3-540-48257-1_14
27. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proceedings Workshop Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer (1982). https://doi.org/10.1007/BFb0025774
28. Kozen, D.: Results on the Propositional mu-Calculus. Theoret. Comput. Sci. **27**, 333–354 (1983). https://doi.org/10.1016/0304-3975(82)90125-6
29. Lawford, M., Ostroff, J.S., Wonham, W.M.: Model Reduction of Modules for State-event Temporal Logics. In: Proceedings IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI (FORTE/PSTV'96). IFIP Conference Proceedings, vol. 69, pp. 263–278. Chapman & Hall, Ltd. (1996)
30. Graf, S., Loiseaux, C.: Property preserving abstractions under parallel composition. In: Proceedings 5th International Joint Conference on Theory and Practice of Software Development CAAP/FASE (TAPSOFT'93). LNCS, vol. 668, pp. 644–657. Springer (1993). https://doi.org/10.1007/3-540-56610-4_95

31. Pecheur, C., Raimondi, F.: Symbolic Model Checking of Logics with Actions. In: Revised Selected and Invited Papers 4th Workshop on Model Checking and Artificial Intelligence (MoChArt'06). LNCS, vol. 4428, pp. 113–128. Springer (2006). https://doi.org/10.1007/978-3-540-74128-2_8
32. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: Proceedings 3rd Symposium on Logic in Computer Science (LICS'88). pp. 203–210. IEEE (1988). https://doi.org/10.1109/LICS.1988.5119
33. Antonik, A., Huth, M., Larsen, K.G., Nyman, U., Wąsowski, A.: 20 Years of Modal and Mixed Specifications. Bulletin of the EATCS **95**, 94–129 (2008)
34. Křetínský, J.: 30 Years of Modal Transition Systems: Survey of Extensions and Analysis. In: Models, Algorithms, Logics and Tools, LNCS, vol. 10460, pp. 36–74. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_3
35. Bradfield, J.C., Stirling, C.: Modal Logics and μ-Calculi: An Introduction. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 293–330. Elsevier (2001). https://doi.org/10.1016/B978-044482830-9/50022-9
36. Kindler, E., Vesper, T.: ESTL: A Temporal Logic for Events and States. In: Proceedings 19th International Conference on the Application and Theory of Petri Nets (ICATPN'98). LNCS, vol. 1420, pp. 365–384. Springer (1998). https://doi.org/10.1007/3-540-69108-1_20
37. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
38. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In: Revised Selected Papers 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'07). LNCS, vol. 4916, pp. 133–148. Springer (2007). https://doi.org/10.1007/978-3-540-79707-4_11
39. Pnueli, A.: Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In: Proceedings 12th International Colloquium on Automata, Languages and Programming (ICALP'85). LNCS, vol. 194, pp. 15–32. Springer (1985). https://doi.org/10.1007/BFb0015727
40. Steffen, B., Ingólfsdóttir, A.: Characteristic Formulae for Processes with Divergence. Inf. Comput. **110**(1), 149–163 (1994). https://doi.org/10.1006/inco.1994.1028
41. Gnesi, S., Mazzanti, F.: An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems. In: Rigorous Software Engineering for Service-Oriented Systems, LNCS, vol. 6582, pp. 390–407. Springer (2011). https://doi.org/10.1007/978-3-642-20401-2_18
42. ter Beek, M.H., Mazzanti, F., Gnesi, S.: CMC–UMC: A Framework for the Verification of Abstract Service-Oriented Properties. In: Proceedings 24th Symposium on Applied Computing (SAC'09). pp. 2111–2117. ACM (2009). https://doi.org/10.1145/1529282.1529751
43. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. J. Log. Algebr. Meth. Program. **85**(2), 287–315 (2016). https://doi.org/10.1016/j.jlamp.2015.11.006
44. Hennessy, M., Milner, R.: Algebraic Laws for Nondeterminism and Concurrency. J. ACM **32**(1), 137–161 (1985). https://doi.org/10.1145/2455.2460
45. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Proceedings 15th International Symposium on Formal Methods (FM'08). LNCS, vol. 5014, pp. 148–164. Springer (2008). https://doi.org/10.1007/978-3-540-68237-0_12

46. ter Beek, M.H., Mazzanti, F.: VMC: Recent Advances and Challenges Ahead. In: Proceedings 18th International Software Product Line Conference (SPLC'14). vol. 2, pp. 70–77. ACM (2014). https://doi.org/10.1145/2647908.2655969
47. Corno, F., Sanaullah, M.: Design Time Methodology for the Formal Verification of Intelligent Domotic Environments. In: Proceedings 2nd International Symposium on Ambient Intelligence (ISAmI'11). AINSC, vol. 92, pp. 9–16. Springer (2011). https://doi.org/10.1007/978-3-642-19937-0_2
48. Corno, F., Sanaullah, M.: Formal Verification of Device State Chart Models. In: Proceedings 7th International Conference on Intelligent Environments (IE'11). pp. 66–73. IEEE (2011). https://doi.org/10.1109/IE.2011.36
49. Mazzanti, F., Spagnolo, G.O., Longa, S.D., Ferrari, A.: Deadlock Avoidance in Train Scheduling: A Model Checking Approach. In: Proceedings 19th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'14). LNCS, vol. 8718, pp. 109–123. Springer (2014). https://doi.org/10.1007/978-3-319-10702-8_8
50. Mazzanti, F., Spagnolo, G.O., Ferrari, A.: Designing a Deadlock-Free Train Scheduler: A Model Checking Approach. In: Proceedings 6th International Symposium NASA Formal Methods (NFM'14). LNCS, vol. 8430, pp. 264–269. Springer (2014). https://doi.org/10.1007/978-3-319-06200-6_22
51. Mazzanti, F., Ferrari, A., Spagnolo, G.O.: Towards formal methods diversity in railways: an experience report with seven frameworks. Int. J. Softw. Tools Technol. Transf. **20**(3), 263–288 (2018). https://doi.org/10.1007/s10009-018-0488-3
52. Fantechi, A., Haxthausen, A.E., Nielsen, M.B.R.: Model Checking Geographically Distributed Interlocking Systems using UMC. In: Proceedings 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'17). pp. 278–286. IEEE (2017). https://doi.org/10.1109/PDP.2017.66
53. Paganelli, F., Ambra, T., Fantechi, A., Giuli, D.: Formalizing REST APIs for web-based communication and SIP interworking. Telecommun. Syst. **66**(1), 75–93 (2017). https://doi.org/10.1007/s11235-016-0271-2
54. Aldini, A.: Modeling and verification of trust and reputation systems. Security Comm. Networks **8**(16), 2933–2946 (2015). https://doi.org/10.1002/sec.1220
55. Aldini, A.: Design and Verification of Trusted Collective Adaptive Systems. ACM Trans. Model. Comput. Simul. **28**(2), 9:1–9:27 (2018). https://doi.org/10.1145/3155337