Refining Real-Time System Specifications through Bounded Model- and Satisfiability-Checking

Matteo Pradella
IEIIT
Consiglio Nazionale delle Ricerche
via Ponzio 34/5
20133 Milano, Italy
Email: pradella@elet.polimi.it

Angelo Morzenti
DEI
Politecnico di Milano
via Ponzio 34/5
20133 Milano, Italy
Email: morzenti@elet.polimi.it

Pierluigi San Pietro
DEI
Politecnico di Milano
via Ponzio 34/5
20133 Milano, Italy
Email: sanpietro@elet.polimi.it

***Abstract*—In Bounded Model Checking (BMC) a system is modeled with a finite automaton and various desired properties with temporal logic formulae. Property verification is achieved by translation into boolean logic and the application of SAT-solvers. Bounded Satisfiability Checking (BSC) adopts a similar approach, but both the system and the properties are modeled with temporal logic formulae, without an underlying operational model. Hence, BSC supports a higher-level, descriptive approach to system specification and analysis. We compare the performance of BMC and BSC over a set of case studies, using the Zot tool to translate automata and temporal logic formulae into boolean logic. We also propose a method to check whether an operational model is a correct implementation (refinement) of a temporal logic model, and assess its effectiveness on the same set of case studies. Our experimental results show the feasibility of BSC and refinement checking, with modest performance loss w.r.t. BMC.**

***Index Terms*—Model checking, Satisfiability checking, Temporal logic, SAT-solver.**

I. Introduction

Bounded Model Checking is a well established technique for analyzing timed reactive systems [1]. The system under analysis is modeled as a finite-state transition system and the property to be checked is expressed as a formula in temporal logic. Infinite, ultimately periodic temporal structures that assign a value to every element of the model alphabet are encoded through a finite set of boolean variables, and the cyclic structure of the time domain is encoded into a set of *loop selector variables* that mark the start and end points of the period. The model and the property are also suitably translated into boolean logic formulae, so that the model checking problem is expressed as an instance of a SAT problem, that can be solved efficiently thanks to the significant improvements that occurred in recent years in the technology of the SAT-solver tools [2], [3]. As it usually occurs in a model checking framework, a (bounded) model-checker tool can either prove a property or disprove it by exhibiting a counter example, thus providing means to support simulation, test case generation, etc. Among the various SAT-based verification tools we cite *NuSMV* [4], a symbolic model checker which supports bounded model checking; and *Alloy* [5], oriented towards the analysis of descriptive models, but without full support to the verification of temporal properties.

In our past work [6] we have introduced a variant of bounded model checking where the underlying, ultimately periodic timing structure was not bounded to be infinite only in the future, but may extend indefinitely also towards the past, thus allowing for a simple and intuitive modeling of continuously functioning systems like monitoring and control devices. Most important, however, in our approach both the system under analysis and the property to be checked are expressed in a single uniform notation as formulae of temporal logic. In this novel setting, which we called bounded *satisfiability* checking (**BSC**), the system under analysis is modeled through the set of all its fundamental properties as a formula ϕ (that in all non-trivial cases would be of significant size) and the additional property to be checked (e.g. a further desired requirement) is expressed as another (usually much smaller) formula ψ. A bounded model checker in this case is used to prove that any implementation of the system under analysis possessing the assumed fundamental properties ϕ would also ensure the additional property ψ; in other terms, the model checker would prove that the formula $\phi \rightarrow \psi$ is valid, or equivalently that its negation is not satisfiable (hence the term satisfiability checking).

Satisfiability verification is very useful, in its simplest form, as a means for performing a sort of testing [7] or *sanity check* of the specification [8], [9] and, more generally, it allows the designer to perform System Requirement Analysis [10], i.e., to investigate which system properties and behaviors are implied by (or are compatible with) the assumed requirements considered as a high level specification. This kind of activity, being centered on the requirements, is naturally performed at the initial stages of the development cycle. This has the advantages of allowing the designer to reason at a high level of abstraction, using simple and readable artifacts like the requirements specification, and without imposing any premature constraint on the implementation. On the other hand, an analysis performed at an early stage of the development process does not provide any support to the design phase and *may* potentially be less efficient that an analysis performed, as in traditional model checking, with reference to an operational model consisting, as it is customary, of a state transition system.

In the present work, we take a significant further step in investigating the feasibility and usefulness of BSC by exploiting its generality and flexibility to provide *two* kinds of models for the system under analysis:

- the *descriptive model*, which consists essentially of a compact, high level, readable requirements specification expressed as a set of formulae in timed temporal logic with past operators, and
- the *operational model*, which is written in a simple but rather general language to characterize a state-transition system.

Through a running example we introduce a method to build the two models in such a way as to facilitate their analysis and comparison. In particular we focus on the interfaces through which they can be related and we investigate the notions of equivalence and implementation among a descriptive and an operational model. These ideas are also validated by applying them to a set of significant examples consisting of benchmark case studies.

The results here reported can be the basis for a unified, encompassing development framework for reactive, embedded, (time) critical systems that supports a seamless transition from requirements elicitation and analysis (carried out by means of satisfiability checking on the first, descriptive model of the system), to a refinement-based, and hence provably correct, high-level design supported by proof of correct implementation and by verification through model checking.

The paper is structured as follows. Section II presents background material on temporal logic and bounded model- and satisfiability-checking. Section III introduces, by means of a running example, the notions of descriptive and operational models. Section IV illustrates, still on the running example, the operations of refinement, proof of correct implementation and equivalence among models. In Section V we report and comment on the experimental results on applying our method and tool to the benchmark case studies. In the concluding section we summarize the obtained results and outline possible future developments.

II. Preliminaries

We first recall here Linear Temporal Logic with past operators (PLTL), in the version introduced by Kamp [11].

Syntax of PLTL The alphabet of PLTL includes: a finite set Ap of propositional letters; two propositional connectives $\neg, \vee$ (from which other traditional connectives such as $\top, \bot, \neg, \vee, \wedge, \rightarrow, \dots$ may be defined); four temporal operators (from which other temporal operators can be derived): the "until" operator $\mathcal{U}$, the "next-time" operator $\circ$, the "since" operator $\mathcal{S}$ and the "past-time" (or Yesterday) operator, $\bullet$. Formulae are defined in the usual inductive way: a propositional letter $p \in Ap$ is a formula; $\neg\phi, \phi\vee\psi, \phi\mathcal{U}\psi, \circ\phi, \phi\mathcal{S}\psi, \bullet\phi$, where ϕ, ψ are formulae, are formulae; nothing else is a formula.

The traditional eventually and globally operators may be defined as: $\Diamond\phi$ is $\top\mathcal{U}\phi$, $\Box\phi$ is $\neg\Diamond\neg\phi$. Their past counterparts are: $\blacklozenge\phi$ is $\top\mathcal{S}\phi$, $\blacksquare\phi$ is $\neg\blacklozenge\neg\phi$. Another useful operator for PLTL is the Always operator $\mathcal{A}lw$, which can be defined by $\mathcal{A}lw\ \phi := \Box\phi \wedge \blacksquare\phi$. The intended meaning of $\mathcal{A}lw\ \phi$ is that ϕ must hold in every instant in the future and in the past. Its dual is the Sometimes operator $\mathcal{S}om\ \phi$ defined as $\neg\mathcal{A}lw\neg\phi$.

Semantics of PLTL A bi-infinite word w over alphabet 2^{Ap} (also called a $\mathbb{Z}$-word) is a function $w : \mathbb{Z} \longrightarrow 2^{Ap}$. Hence, $w(j) \in 2^{Ap}$ for every j. Word w is also denoted as $\dots w(-1)w(0)w(1)\dots$ and each $w(j)$ also as w_j. The set of all bi-infinite words over 2^{Ap} is denoted by $(2^{Ap})^{\mathbb{Z}}$. An ω-word over 2^{Ap} is a function from $\mathbb{N} \rightarrow 2^{Ap}$, i.e., it has the form $w(0)w(1)\dots$.

The semantics of PLTL may be defined on $\mathbb{Z}$-words (i.e., bi-infinite time) or on ω-words (i.e., mono-infinite time). We present here only the former case, even if the latter is much more common in BMC, since bi-infinite semantics is actually simpler and includes the mono-infinite one as a special case. Also, our tool Zot supports both mono-infinite and bi-infinite cases, and some of the experiments of Section V use bi-infinite time.

For all PLTL formulae ϕ, for all $w \in (2^{Ap})^{\mathbb{Z}}$, for all integer numbers i, the satisfaction relation $w, i \models \phi$ is defined as follows.

$$
\begin{array}{l}
w,i \models p, \iff p \in w(i), \text{for } p \in Ap \\
w,i \models \neg\phi \iff w,i \not\models \phi \\
w,i \models \phi \vee \psi \iff w,i \models \phi \text{ or } w,i \models \psi \\
w,i \models \circ\phi \iff w,i+1 \models \phi \\
w,i \models \phi\mathcal{U}\psi \iff \exists k \geq 0 \mid w,i+k \models \psi, \text{ and} \\
\quad w,i+j \models \phi\ \forall 0 \leq j < k \\
w,i \models \bullet\phi \iff w,i-1 \models \phi \\
w,i \models \phi\mathcal{S}\psi \iff \exists k \geq 0 \mid w,i-k \models \psi, \text{ and} \\
\quad w,i-j \models \phi\ \forall 0 \leq j < k
\end{array}
$$

Metric PLTL PLTL can also be extended by adding metric operators, on discrete time. Metric operators are very convenient for modeling hard real time systems, with quantitative time constraints. The resulting logic, called *Metric PLTL*, does not actually extend the expressive power of PLTL: it is a syntactically-sugared, but considerably more succinct and convenient, version of PLTL.

Metric PLTL extends the alphabet of PLTL with a *bounded until* operator $\mathcal{U}_{\sim c}$ and a *bounded since* operator $\mathcal{S}_{\sim c}$, where $\sim$ represents any relational operator (i.e., $\sim \in \{\leq, =, \geq\}$), and c is a natural number. Also, we allow n-ary predicate letters (with $n \geq 1$) and the $\forall, \exists$ quantifiers as long as their domains are finite. Hence, one can write, e.g., formulae of the form: $\exists p\ \mathrm{gr}(p)$, with p ranging over $\{1,2,3\}$ as a shorthand for $\bigvee_{p\in\{1,2,3\}} \mathrm{gr}_p$.

The bounded globally and bounded eventually operators are defined as follows: $\Diamond_{\sim c}\phi$ is $\top\mathcal{U}_{\sim c}\phi$, $\Box_{\sim c}\phi$. The past versions of the bounded eventually and globally operators may be defined symmetrically to their future counterparts: $\blacklozenge_{\sim c}\phi$ is $\top\mathcal{S}_{\sim c}\phi$, $\blacksquare_{\sim c}\phi$ is $\neg\blacklozenge_{\sim c}\neg\phi$.

Versions of the bounded operators with a strict bound may be introduced as a shorthand. For instance, $\phi\mathcal{U}_{>0}\psi$ stands for $\circ(\phi\mathcal{U}_{\geq 0}\psi)$.

The semantics of Metric PLTL may be defined by a straightforward translation τ of its operators into PLTL:

$$\tau(\phi_1 \mathcal{U}_{\leq 0} \phi_2) := \phi_2$$
$$\tau(\phi_1 \mathcal{U}_{\leq t} \phi_2) := \phi_2 \vee \phi_1 \wedge \circ\tau(\phi_1 \mathcal{U}_{\leq t-1} \phi_2), \text{with } t > 0$$
$$\tau(\phi_1 \mathcal{U}_{\geq 0} \phi_2) := \phi_1 \mathcal{U} \phi_2$$
$$\tau(\phi_1 \mathcal{U}_{\geq t} \phi_2) := \phi_1 \wedge \circ\tau(\phi_1 \mathcal{U}_{\geq t-1} \phi_2), \text{with } t > 0$$
$$\tau(\phi_1 \mathcal{U}_{=0} \phi_2) := \phi_2$$
$$\tau(\phi_1 \mathcal{U}_{=t} \phi_2) := \phi_1 \wedge \circ\tau(\phi_1 \mathcal{U}_{=t-1} \phi_2), \text{with } t > 0$$

and symmetrically for the operators in the past.

A. The Zot toolkit

Zot is an agile and easily extendible bounded model checker, which can be downloaded at http://home.dei.polimi.-it/pradella/, together with the case studies and results described in Section V.

The tool supports different logic languages through a multi-layered approach: its core uses PLTL, and on top of it a decidable predicative fragment of TRIO [12] is defined (essentially, equivalent to Metric PLTL). An interesting feature of Zot is its ability to support different encodings of temporal logic as SAT problems by means of plugins. This approach encourages experimentation, as plugins are expected to be quite simple, compact (usually around 500 lines of code), easily modifiable, and extendible. At the moment, a few variants of some of the encodings presented in [13] are supported, a dense-time variant of MTL [14], and the bi-infinite encoding presented in [6].

Zot offers three basic usage modalities:

1) *Bounded satisfiability checking (BSC)*: given as input a specification formula, the tool returns a (possibly empty) history (i.e., an execution trace of the specified system) which satisfies the specification. An empty history means that it is impossible to satisfy the specification.
2) *Bounded model checking (BMC)*: given as input an operational model of the system, the tool returns a (possibly empty) history (i.e., an execution trace of the specified system) which satisfies it.
3) *History checking and completion (HCC)*: The input file can also contain a partial (or complete) history H. In this case, if H complies with the specification, then a completed version of H is returned as output, otherwise the output is empty.

The provided output histories have temporal length $\leq k$, the bound given by the user, but may represent infinite behaviors thanks to the loop selector variables, marking the start of the periodic sections of the history. The BSC/BMC modalities can be used to check if a property *prop* of the given specification *spec* holds over every periodic behavior with period $\leq k$. In this case, the input file contains $\text{spec} \wedge \neg\text{prop}$, and, if prop indeed holds, then the output history is empty. If this is not the case, the output history is a counterexample, explaining why prop does not hold.

The tool and its plugins were validated on mono-infinite examples, such as the Mutex examples included in the distribution of NuSMV. The results were exactly the same as those obtained by using NuSMV [4] with the same encoding. On one hand, Zot is in general slower than NuSMV, but being quite small and written in *Common Lisp* is quite flexible, and promotes experimentation with different encodings and logic languages. On the other hand, in practice its performances are usually acceptable, because for non-trivial verifications the bottleneck typically resides in the SAT solver rather than in the translator.

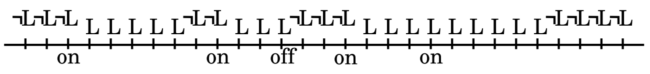


Fig. 1. A history for the example of the timed lamp

Zot supports the model checkers MiniSat [3], zChaff, [2], and the recent multi-threaded MiraXT solver [15].

III. Descriptive vs. Operational models

We now remind of two different, complementary ways to define a model of a system.

The first one, called descriptive model, is based on the idea of characterizing the modeled system through its *fundamental properties*, described by means of LTL formulae on an alphabet of items that correspond to the interface of the system with the external world, without considering any possible further internal components that might be necessary for its functioning. Such LTL formulae are *not* constrained in any way in their form: they may refer to any time instant, possibly relating actions and events occurring at any arbitrary distance in time, or they may constrain values and behaviors for arbitrarily long time intervals.

On the other hand the second way of modeling, the *operational model* consists of a set of clauses that constrain the transition of the system from a state valid in one given instant, the *current state*, to the *next state*, reached by the modeled system in the successive time instants. The Zot toolkit provides a simple language to describe both descriptive and operational models, and to mix them freely. This is possible since both models are finally to be translated into boolean logic, to be fed to a SAT solver.

A. An example: a lamp with a timer

As a simplest example on which to discuss the introduced concepts we consider a so-called *timer-reset-lamp* (TRL), i.e., a lamp with two buttons, called *ON* and *OFF*. When the *ON* button is pressed the lamp is lighted and it remains so for Δ time units (t.u.) and then it goes off, unless the *OFF* button is pushed before the Δ time-out expires (in which case the light goes off immediately after the push of the *OFF* button , even is this occurs before the end of the time-out period), or unless the *ON* button is pressed again, before the time-out, in which case the lamp will remain lighted for more Δ t.u. (unless the *OFF* button is pressed before the time-out expires, etc.). To ensure that the pressure of a button is always meaningful, it is assumed that *ON* and *OFF* cannot be pressed simultaneously.

An example of a trace of execution of the TRL system (a so-called *history*) is represented in Figure 1, for the case $\Delta = 5$. The history shows typical behaviors of the modeled system: the lamp being off is turned on by pushing button *ON* and then it turns off "spontaneously" after Δ t.u.; then the lamp is lighted again and then turned off within Δ t.u. by pressing button *OFF*; the lamp is kept on by pushing again button *ON* before the Δ time-out, and then it finally goes off spontaneously.

The descriptive model of the TRL is based on following three propositional letters, with the indicated meaning:

L the light is on,
ON the button to turn it on is pressed,
OFF the button to turn it off is pressed.

To distinguish the present, descriptive model from the operational model that will be presented next, we add a subscript $_{de}$ to the names of the propositional letters, which thus become L_{de}, ON_{de}, OFF_{de}. The first sub formula of the descriptive model is:

$$\text{(D1)} \quad L_{de} \leftrightarrow \exists x \begin{pmatrix} 0 < x \leq \Delta \wedge \\ \blacklozenge_{=x} ON_{de} \wedge \\ \neg \blacklozenge_{<x} OFF_{de} \end{pmatrix}$$

Which states that the lamp is on (at the current time) if and only if the *ON* button was pressed not more than Δ time units ago and since then the *OFF* button was never pressed. The second sub formula expresses the mutual exclusion between the pressing of the *ON* and *OFF* buttons

$$\text{(D2)} \quad \neg(ON_{de} \wedge OFF_{de})$$

The descriptive model of the formula simply consists of the conjunction of these two formulae, enclosed in a universal temporal quantification (an $\mathcal{A}lw$ operator) asserting that they hold for all instants of the temporal domain.

$$\text{(DM)} \quad \mathcal{A}lw(D1 \wedge D2)$$

The descriptive model, despite its simplicity and succinctness, characterizes completely the TRL system: starting from it the history depicted in Figure 1 can be generated using the Zot tool, or one can prove that the following (conjectured) property

$$\text{(DP1)} \quad \mathcal{A}lw(\neg \Box_{\leq \Delta+1} L_{de})$$

(i.e., the lamp will never remain on for more than Δ time units) does *not* hold, by generating, through the Zot tool, a counter-example consisting of a history similar to the one shown in Figure 1, including two push actions of the *ON* button at distance less than Δ; the Zot tool can instead prove, from the descriptive model, the following property

$$\text{(DP2)} \quad \begin{array}{c} \mathcal{S}om(\Box_{\leq \Delta+1} L_{de}) \rightarrow \\ \mathcal{S}om(ON_{de} \wedge \Diamond_{\leq \Delta} ON_{de}) \end{array}$$

(i.e., the lamp remains lighted for more than Δ time units only in case of two consecutive press actions of the *ON* button at a distance of less than Δ t.u.). The latter property is proved by the Zot tool in 1.45 seconds with a time structure of $k = 15$ time points.

We now show how an *operational model* for the TRL system can be provided. As mentioned above, the idea is to define, for each instant, the next system state based on the current state and, possibly, of the stimuli coming, still at the current time, from the environment. A brief reflection shows however that the current state of the TRL system is *not* completely characterized by the value of predicate letter L; e.g., if at a given time we know that the lamp is on (predicate letter L_{op} holds, notice subscript op on the predicate letter standing for *operational*) and that no button is pressed, this does *not* allow us to conclude that the lamp will still be on at the next time instant, since this obviously depends on the time that the lamp has been on (more precisely, it depends on the time that has elapsed from the last press action on the *ON* button). To model explicitly this component of the state it is therefore necessary to introduce a further element in the alphabet of the model: a counter variable ranging in the interval $[0 \ldots \Delta]$ to store exactly this information. With this addition the definition of the operational model becomes an easy exercise. The model (not reported here for the sake of brevity) consists of a set of propositional formulae that relate the present state with the next state, and it corresponds to a classical finite state automaton depicted, with customary graphical conventions, in Figure 2.

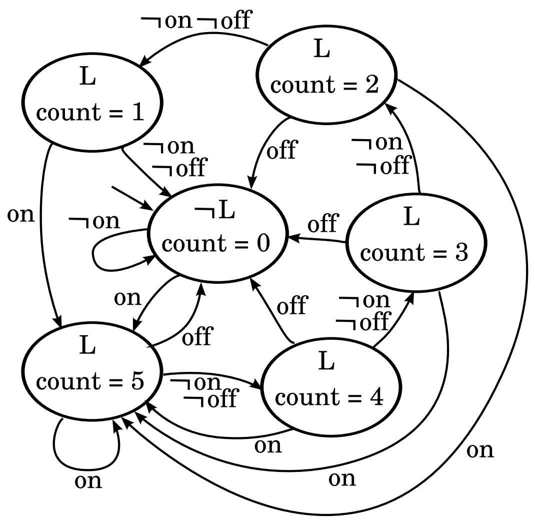


Fig. 2. An automaton for the example of the timed lamp

Clearly, the operational model (OM) provides a complete and unambiguous characterization of the TRL system, as well as the descriptive model (DM). For instance, the following properties, at all similar to DP1 and DP2,

$$\text{(OP1)} \quad \mathcal{A}lw(\neg \Box_{\leq \Delta+1} L_{op})$$

$$\text{(OP2)} \quad \begin{array}{c} \mathcal{S}om(\Box_{\leq \Delta+1} L_{op}) \rightarrow \\ \mathcal{S}om(ON_{op} \wedge \Diamond_{\leq \Delta} ON_{op}) \end{array}$$

can easily be (dis)proved by the Zot tool with the same results as in the descriptive model. It is also interesting to note that property (OP2) is proved by the tool in 0.86 seconds (as opposed to the 1.45 seconds for the descriptive model).

IV. Model development, analysis and verification, refinement, and equivalence

Let us now step back and reconsider the conceptual path that we have followed so far, and add a few methodological remarks.

We started from an informal description of the TRL system. Then we characterized it by means of a set of LTL formulae with a minimal alphabet of specification items and without imposing any particular constraint of the structure of the formulae: we called the result of this formalization the descriptive model. Next we provided a further, alternative characterization of the TRL system in an operational style (through a set of boolean formulae relating current-state and next-state) maintaining the same alphabet of specification items for the external, visible part of the model; not surprisingly, going from a descriptive model to an operational one we were led to add elements of the model alphabet to represent the system internal state variables (here, in a typical way, a counter variable). The internal state variables are necessary in the operational model to "carry the information" on the current system configuration from one instant to the next, because of the constrained form of the clauses composing the operational model.

The descriptive model is typically more compact and concise than the operational one, being composed of compact formulae, often with a high level of temporal nesting, that express more abstractly its characteristic properties without any reference to its internal state.

In fact, the descriptive model constitutes both the formalization of the system requirements and an abstract specification. Writing an operational model with reference to the same specification alphabet and adding other items representing internal state variables corresponds to what is typically done in the first phases of the development cycle, when one outlines the system architectures and the means by which the required properties can be ensured. Often these first steps in the development cycle are formalized in terms of a *refinement* operation, that provides a relation between two models (often one being derived from the other one) showing that certain logical-algebraic properties among them hold, which ensure that one of the two models is a correct implementation of the other one.

In our TRL example, we have two models that are obviously comparable because (except for the de or op subscripts) they refer to the same alphabet for the external predicate letters L, ON, OFF, and we are led to conjecture that they are equivalent, also based on the fact that the properties (DP1) and (DP2) on one side, and (OP1) and (OP2) on the other side, are (dis)proved by the Zot tool with the same results for the two models.

We can intuitively be convinced that the two models are equivalent because of their simplicity, but of course we seek for a method and a procedure to prove formally the equivalence, to be applied to practical cases, which are far more complex (so that our intuition can be easily deceived) and often critical (so that the consequences of a misjudgment would be severe).

We can exploit the generality and flexibility of our LTL-based approach and the availability of the Zot tool to provide a framework supporting analysis and verification, where we consider the two models (DM) and (OM) and assert their equivalence, under the obvious condition that the elements that correspond to the "external" components of the alphabet in the two models are identical. We therefore add the following identity condition:

$$\text{(ID)}\quad \mathcal{A}lw\begin{pmatrix} L_{op} \leftrightarrow L_{de} \wedge \\ ON_{op} \leftrightarrow ON_{de} \wedge \\ OFF_{op} \leftrightarrow OFF_{de} \end{pmatrix}$$

and we verify, using the Zot tool, the property of equivalence of the two models, namely $(DM \leftrightarrow OM)$, which for convenience we divide into the two implications $(OM \rightarrow DM)$ and $(DM \rightarrow OM)$. It is worth noticing that the first property $(OM \rightarrow DM)$ asserts that the operational model constitutes a correct implementation of the descriptive one, i.e., all executions/histories of the operational model related to the descriptive model by means of the identity condition (ID), satisfy it, while the second property, $(DM \rightarrow OM)$, added to the first one, states that the two models are completely equivalent.

The Zot tool proves the correct implementation:

$$(OM \wedge ID) \rightarrow DM$$

in 1.43 seconds with a time structure of $k = 15$ time points. On the contrary the opposite property, formalized by:

$$(DM \wedge ID) \rightarrow OM$$

is *not* proved, because Zot finds a counterexample: the two models, the operational and the descriptive one, are therefore not equivalent. An inspection of the counterexample shows that it satisfies the premise of the implication, $(DM \wedge ID)$, and falsifies the consequence, OM, by a combination of pressing of buttons and on or off light states of the lamp that satisfies the properties of the system (and hence the descriptive model DM) but it contains a set of values for the *count* variable that are inconsistent with the operational model OM, which is therefore, considered as a boolean formula, falsified.

Hence, if we wish to obtain a complete equivalence between the two models DM and OM, we should add further constraints to the identity condition (ID), to state that the values of the counter are consistent with the values of the other variables of the descriptive model, and in particular with the pressing actions on the buttons. The formulae that assert this consistence are stated in the following, with brief comments:

$$\begin{gathered} \mathit{count} = 0 \\ \leftrightarrow \\ \blacksquare_{\leq \Delta} \neg ON_{de} \vee (\neg OFF_{de} \wedge (\neg ON_{de}\ \mathcal{S}\ OFF_{de})) \end{gathered}$$

(the counter is null if and only if the *ON* button was not pushed in the last Δ t.u. or if no push of the *ON* button followed the last push in the past of the *OFF* button)

$$\forall x \left(0 < x \leq \Delta \rightarrow \left(\begin{array}{c} \mathit{count} = x \\ \leftrightarrow \\ \blacklozenge_{=\Delta+1-x} \mathit{ON}_{de} \wedge \\ \neg \mathit{OFF}_{de} \wedge \\ (\neg \mathit{OFF}_{de} \ \mathcal{S} \ \mathit{ON}_{de}) \end{array} \right) \right)$$

(*count* is equal to x greater than 0 if and only if the *ON* button was pressed Δ+1-x t.u. ago and no push of the *OFF* button occurred in the past since then)

With the addition of these two constraints the Zot tool completes the proof of equivalence of the two models in 2.9 seconds with a time structure of $k = 15$ time points.

It can be remarked that, even in the simplest example of the TRL system, the conditions added to relate the values of the internal *count* variable of the operational model to the values of the external variables of the descriptive model, with the purpose of proving the complete equivalence of the two models, are nontrivial. In non-toy examples or in practical cases the required effort, and the likelihood of errors in writing these constraints, which essentially formalize the meaning and the purpose of the internal state variables, can be comparable to those encountered in defining the operational model itself.

It is also to be noticed that in a typical process of system development through refinement of requirement specifications -formalized by a descriptive model- into an operational model that implements it, the formal proof of the relation of correct implementation (i.e., of the validity of the formula $OM \wedge ID \rightarrow DM$) can suffice, if one does not intend to use the operational model as a sort of benchmark with respect to which the descriptive model (i.e., the original requirement specification itself) should be validated in order to check that it ensures all the properties formalized by the operational model.

In a refinement-based development process then question may arise if, when one has produced both a descriptive model that abstractly specifies the requirements and an operational one that has been proved to be a correct implementation of it, it would be preferable to analyze some further, desired properties of the system under development with reference to the descriptive model or to the operational one. In the simple example of the TRL we have seen that the proof of the property P2 (in the two versions DP2 and OP2) was carried out more efficiently with respect to the operational model (proof time 1.6 versus 4 seconds). A possible cause for this could be that the operational model is more ”deterministic” than the descriptive one, so that the state space that the tool has to explore to prove the property is more limited in size. This is however only a conjecture, as the time needed to complete the proof might depend, in subtle and involved ways, on the size of the formula, on the depth of nesting of its temporal operators, on the size of the model alphabet, and possibly on other factors.

In the next section we further elaborate on these and other questions with reference to a series of less trivial examples and to benchmark case studies frequently adopted in the literature on timed, critical systems.

V. Experimental results

Here we briefly describe our three case studies. The interested reader can find a complete archive with the Zot input files used for the experiments, and the detailed outcomes in the Zot web page http://home.dei.polimi.it/pradella/.

A. Fischer’s protocol

As a first case study, we consider Fischer’s algorithm [16], a timed mutual exclusion algorithm that allows a number of timed processes to access a shared resource. These processes are usually described as timed automata, and are often used as a benchmark for timed automata verification tools.

We considered the system in two variants. The first one, called Fischer1, considers 3 processes with a delay after the request of 4 time units. The second one, called Fischer2, considers 4 processes with a delay after the request of 5 time units.

We used the tool to check the safety property of the system (*safety* in the tables of the following section), i.e. it is never possible that two different processes enter their critical sections at the same time instant.

As a last test for this system, we added a constraint to generate a behavior in which there is always at least an alive process in the system (*alive* in the tables).

For this case study, we used the mono-infinite encoding.

B. Kernel Railway Crossing

The Railway Crossing problem is a standard benchmark in real time systems verification [17]. It considers a railway crossing composed of a sensor, a gate and a controller. When a train is sensed to approach the crossing, a signal is sent to the controller. The controller then sends a command to the gate, closing the railway crossing to cars. The system operates in real time, ensuring safety (when the train is inside the railway crossing then the bar gate is closed) while maximizing utility (the bar should be open as long as possible). To this end, we adopt various assumptions on the minimum and maximum speed of trains (e.g., the minimum time it takes for a train to enter the crossing after being sensed) and on the bar speed (the time it takes for the bar to be moved up or down). The Kernel Railroad crossing problem is a simplified version, where there is only one track and hence only one train at a time may enter the crossing. The goal of the KRC specification is twofold: a formal definition of the KRC system, and the proof of the safety and utility properties.

KRC is a toy example per se, but in this case we are completely defining it with a temporal logic specification, thus obtaining a logic formula much bigger and more complex than those used in traditional model checking, where the KRC is defined with an automaton and short temporal logic formulae are used only to model safety or utility properties.

In our example we studied the KRC problem with a set of time constants that allow a high degree of nondeterminism on train behavior. In particular, the set of constants is: $d_{Max} = 9$ and $d_{min} = 5$ time units for the maximum and minimum time for a train to reach the critical region, $h_{Max} = 6$ and $h_{min} = 3$

for the maximum and minimum time for a train to be inside the critical region, and $\gamma = 3$ for the movement of the bar from up to down and viceversa.

Satisfiability of the specification, a safety property and refinement were considered for the experiments, using a mono-infinite encoding. We also considered a special case of refinement, that we called "mixed refinement". In fact, the KRC specification is composed of three modules: the bar, the train and the controller. The only system to be implemented in software or digital hardware is the controller, while the bar and the train are a part of the environment. Hence, in SW development it would not make sense to define operational models also for the bar and the train. The mixed refinement experiments consider exactly this case: only the controller is refined. This is possible, thanks to the flexibility of Zot that allows mixing operational and descriptive constraints.

C. *Real-time allocator*

The last case study consists of a real-time allocator which serves a set of client processes, competing for a shared resource. The purely descriptive version of it was originally presented in [6]. Here we compare the descriptive version with a new operational version.

Each process p requires the resource by issuing the message $\mathrm{rq}(p)$, by which it identifies itself to the allocator. Requests have a time out: they must be served within T_{req} time units, or else be ignored by the allocator. If the allocator is able to satisfy p's request within the time-out, then it grants the resource to p by a $\mathrm{gr}(p)$ signal. Once a process is assigned the resource by the allocator, it releases the resource, by issuing a rel signal, within a maximum of T_{rel} time units. The allocator grants the request to processes according to a FIFO policy, considering only requests that are not timed out yet and in a timely manner, i.e., no process will have to wait for the resource while it is not assigned to any other process.

In the following experiments we considered the case of a system with three processes and $T_{rel} = T_{req} = 3$. As in the previous case studies, we first used Zot to generate a simple "run" of the system (history generation); then we considered the following four hard real time properties.

Simple Fairness The first is a simple fairness property. If a process that does not obtain the resource always requests it again immediately after the request is expired, then if it requests the resource it will eventually obtain it. This property holds only for $T_{rel} < T_{req}$, hence not in our case, and Zot generates a counterexample.

Conditional Fairness A second, more complex property may be intuitively described as a sort of "conditional fairness". Let us first define the notion of "unconstrained rotation" among processes: a process will require the resource only after all other ones have requested and obtained it. Notice that this requirement does not impose any precise ordering among the requests made by the processes (though, once requests take place in a given order, the order remains unchanged from one round among processes to the next one). Under this assumption of "unconstrained rotation" the allocator system is fair for all processes: if a process, when it requests the resource and does not obtain it, always requests it again after the request is expired, then, when it requests the resource, it will eventually obtain it.

Precedence The third property is about precedence: the allocator system cannot grant the resource to a process a asking for it after another process b, if the resource has not yet been granted to b.

Suspend Fairness The last considered property is used to prove *simple fairness*, under the assumption that every process, after obtaining the resource, suspends itself for $T_{rel} \cdot n_p$ t.u., where n_p is the number of processes.

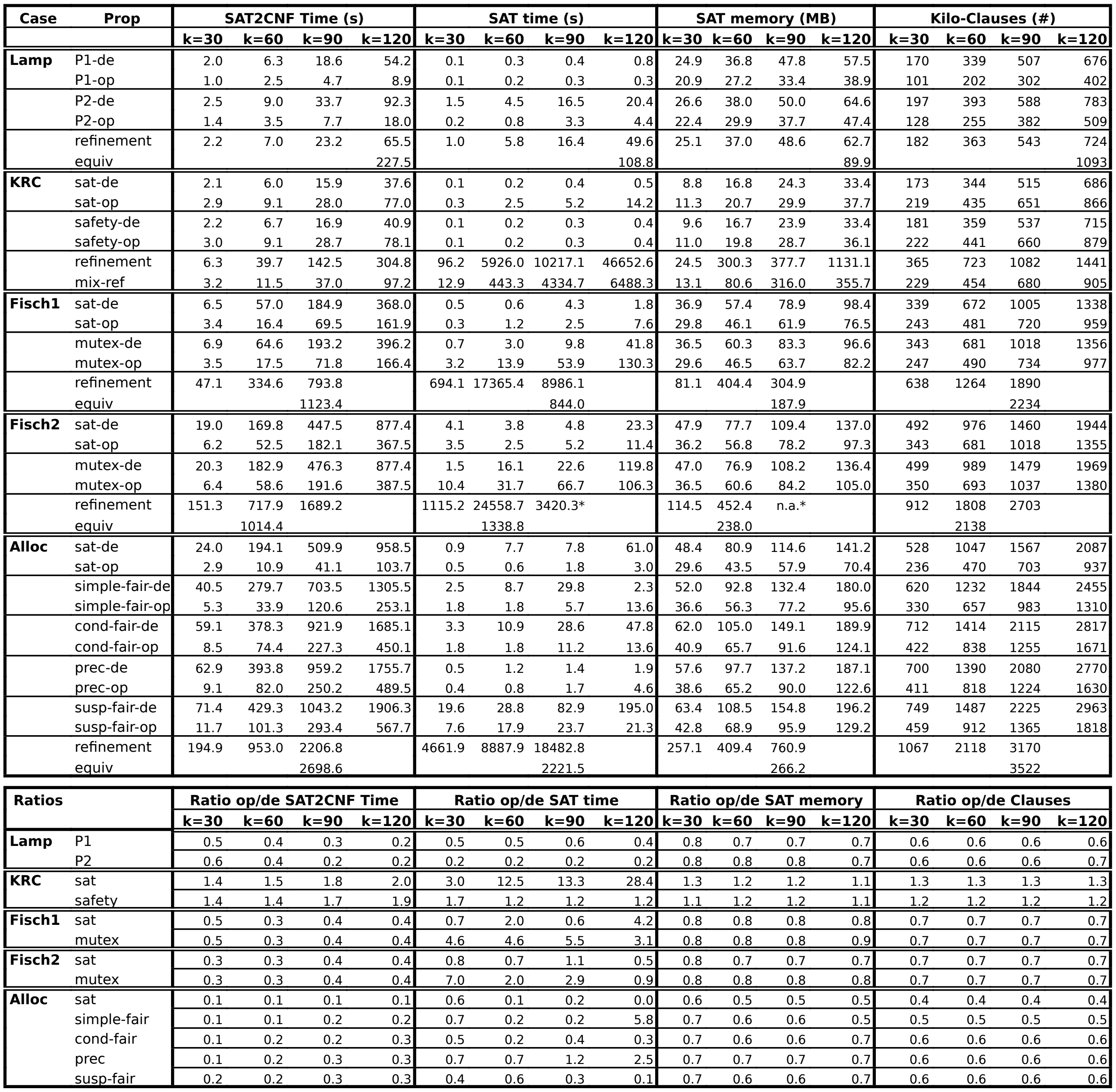
For simplicity, we used the bi-infinite encoding for this case study, because the assumption of a sequence of events that extends itself indefinitely in the past is a useful abstraction with respect to the start of the allocator system: a designer might prefer to ignore the behavior of the allocator right after its start and consider its properties only on regime behavior.

D. *Results*

The experimental results for the case studies described above are shown in Figure 3, with the figures of the simplest Lamp example reported in the first lines as a reference for further comparison. The table reports, for various values of the bound k (30, 60, 90 and 120), SAT time and memory for various properties and systems. To give an idea of the size of the examples, the total number of boolean clauses fed to the SAT solver are also included. Another measure, called SAT2CNF, reports on the time involved in traslating Zot output into the conjunctive normal form required by SAT solvers. This by far dominates translation into boolean formulae, especially on large examples. The experiments were run on a single core of a 4 Gbyte RAM machine with a 2.67 Ghz Intel Core 2 Quad Processor Q6700 (on a Optiplex Dell 755). The SAT solver is MiniSat 1.14 (the only experiment marked with "*" was performed with MiraXT - which does not indicate the memory used - in a case where MiniSat was not able to complete the proof before the cutoff).

For every example, the suffix *-de* indicates the descriptive version of the model, while *-op* is used for the operational version. Every row considers an experiment: *sat* stands for a satisfiability check (i.e., an unconstrained history generation); *refinement* naturally stands for refinement proofs, i.e. that the operational model implies the descriptive one. *Equiv* refers to the reverse implication, i.e., the descriptive model implies the operational one, used together with refinement to actually prove the satisfiability equivalence of the two models. For the latter experiment we considered only one significant case each, just to validate our comparison. Experiments had a cut-off time at 24 hours. The cut-off occurred only for the largest bounds ($k = 90, 120$) and only for some of the computationally most expensive models (e.g. the real-time allocator).

Refinement is by far the most time-consuming operation, affecting mostly SAT solver time. In fact, SAT2CNF time is only dependent on the size of the model, while SAT solving time depends more on the "intrinsic" hardness of the problem,



Case	Prop	SAT2CNF Time (s)				SAT time (s)				SAT memory (MB)				Kilo-Clauses (#)			
		k=30	k=60	k=90	k=120	k=30	k=60	k=90	k=120	k=30	k=60	k=90	k=120	k=30	k=60	k=90	k=120
Lamp	P1-de	2.0	6.3	18.6	54.2	0.1	0.3	0.4	0.8	24.9	36.8	47.8	57.5	170	339	507	676
	P1-op	1.0	2.5	4.7	8.9	0.1	0.2	0.3	0.3	20.9	27.2	33.4	38.9	101	202	302	402
	P2-de	2.5	9.0	33.7	92.3	1.5	4.5	16.5	20.4	26.6	38.0	50.0	64.6	197	393	588	783
	P2-op	1.4	3.5	7.7	18.0	0.2	0.8	3.3	4.4	22.4	29.9	37.7	47.4	128	255	382	509
	refinement	2.2	7.0	23.2	65.5	1.0	5.8	16.4	49.6	25.1	37.0	48.6	62.7	182	363	543	724
	equiv				227.5				108.8				89.9				1093
KRC	sat-de	2.1	6.0	15.9	37.6	0.1	0.2	0.4	0.5	8.8	16.8	24.3	33.4	173	344	515	686
	sat-op	2.9	9.1	28.0	77.0	0.3	2.5	5.2	14.2	11.3	20.7	29.9	37.7	219	435	651	866
	safety-de	2.2	6.7	16.9	40.9	0.1	0.2	0.3	0.4	9.6	16.7	23.9	33.4	181	359	537	715
	safety-op	3.0	9.1	28.7	78.1	0.1	0.2	0.3	0.4	11.0	19.8	28.7	36.1	222	441	660	879
	refinement	6.3	39.7	142.5	304.8	96.2	5926.0	10217.1	46652.6	24.5	300.3	377.7	1131.1	365	723	1082	1441
	mix-ref	3.2	11.5	37.0	97.2	12.9	443.3	4334.7	6488.3	13.1	80.6	316.0	355.7	229	454	680	905
Fisch1	sat-de	6.5	57.0	184.9	368.0	0.5	0.6	4.3	1.8	36.9	57.4	78.9	98.4	339	672	1005	1338
	sat-op	3.4	16.4	69.5	161.9	0.3	1.2	2.5	7.6	29.8	46.1	61.9	76.5	243	481	720	959
	mutex-de	6.9	64.6	193.2	396.2	0.7	3.0	9.8	41.8	36.5	60.3	83.3	96.6	343	681	1018	1356
	mutex-op	3.5	17.5	71.8	166.4	3.2	13.9	53.9	130.3	29.6	46.5	63.7	82.2	247	490	734	977
	refinement	47.1	334.6	793.8		694.1	17365.4	8986.1		81.1	404.4	304.9		638	1264	1890	
	equiv			1123.4				844.0				187.9				2234	
Fisch2	sat-de	19.0	169.8	447.5	877.4	4.1	3.8	4.8	23.3	47.9	77.7	109.4	137.0	492	976	1460	1944
	sat-op	6.2	52.5	182.1	367.5	3.5	2.5	5.2	11.4	36.2	56.8	78.2	97.3	343	681	1018	1355
	mutex-de	20.3	182.9	476.3	877.4	1.5	16.1	22.6	119.8	47.0	76.9	108.2	136.4	499	989	1479	1969
	mutex-op	6.4	58.6	191.6	387.5	10.4	31.7	66.7	106.3	36.5	60.6	84.2	105.0	350	693	1037	1380
	refinement	151.3	717.9	1689.2		1115.2	24558.7	3420.3*		114.5	452.4	n.a.*		912	1808	2703	
	equiv		1014.4				1338.8				238.0				2138		
Alloc	sat-de	24.0	194.1	509.9	958.5	0.9	7.7	7.8	61.0	48.4	80.9	114.6	141.2	528	1047	1567	2087
	sat-op	2.9	10.9	41.1	103.7	0.5	0.6	1.8	3.0	29.6	43.5	57.9	70.4	236	470	703	937
	simple-fair-de	40.5	279.7	703.5	1305.5	2.5	8.7	29.8	2.3	52.0	92.8	132.4	180.0	620	1232	1844	2455
	simple-fair-op	5.3	33.9	120.6	253.1	1.8	1.8	5.7	13.6	36.6	56.3	77.2	95.6	330	657	983	1310
	cond-fair-de	59.1	378.3	921.9	1685.1	3.3	10.9	28.6	47.8	62.0	105.0	149.1	189.9	712	1414	2115	2817
	cond-fair-op	8.5	74.4	227.3	450.1	1.8	1.8	11.2	13.6	40.9	65.7	91.6	124.1	422	838	1255	1671
	prec-de	62.9	393.8	959.2	1755.7	0.5	1.2	1.4	1.9	57.6	97.7	137.2	187.1	700	1390	2080	2770
	prec-op	9.1	82.0	250.2	489.5	0.4	0.8	1.7	4.6	38.6	65.2	90.0	122.6	411	818	1224	1630
	susp-fair-de	71.4	429.3	1043.2	1906.3	19.6	28.8	82.9	195.0	63.4	108.5	154.8	196.2	749	1487	2225	2963
	susp-fair-op	11.7	101.3	293.4	567.7	7.6	17.9	23.7	21.3	42.8	68.9	95.9	129.2	459	912	1365	1818
	refinement	194.9	953.0	2206.8		4661.9	8887.9	18482.8		257.1	409.4	760.9		1067	2118	3170	
	equiv			2698.6				2221.5				266.2				3522	

Ratios		Ratio op/de SAT2CNF Time				Ratio op/de SAT time				Ratio op/de SAT memory				Ratio op/de Clauses			
		k=30	k=60	k=90	k=120	k=30	k=60	k=90	k=120	k=30	k=60	k=90	k=120	k=30	k=60	k=90	k=120
Lamp	P1	0.5	0.4	0.3	0.2	0.5	0.5	0.6	0.4	0.8	0.7	0.7	0.7	0.6	0.6	0.6	0.6
	P2	0.6	0.4	0.2	0.2	0.2	0.2	0.2	0.2	0.8	0.8	0.8	0.7	0.6	0.6	0.6	0.7
KRC	sat	1.4	1.5	1.8	2.0	3.0	12.5	13.3	28.4	1.3	1.2	1.2	1.1	1.3	1.3	1.3	1.3
	safety	1.4	1.4	1.7	1.9	1.7	1.2	1.2	1.2	1.1	1.2	1.2	1.1	1.2	1.2	1.2	1.2
Fisch1	sat	0.5	0.3	0.4	0.4	0.7	2.0	0.6	4.2	0.8	0.8	0.8	0.8	0.7	0.7	0.7	0.7
	mutex	0.5	0.3	0.4	0.4	4.6	4.6	5.5	3.1	0.8	0.8	0.8	0.9	0.7	0.7	0.7	0.7
Fisch2	sat	0.3	0.3	0.4	0.4	0.8	0.7	1.1	0.5	0.8	0.7	0.7	0.7	0.7	0.7	0.7	0.7
	mutex	0.3	0.3	0.4	0.4	7.0	2.0	2.9	0.9	0.8	0.8	0.8	0.8	0.7	0.7	0.7	0.7
Alloc	sat	0.1	0.1	0.1	0.1	0.6	0.1	0.2	0.0	0.6	0.5	0.5	0.5	0.4	0.4	0.4	0.4
	simple-fair	0.1	0.1	0.2	0.2	0.7	0.2	0.2	5.8	0.7	0.6	0.6	0.5	0.5	0.5	0.5	0.5
	cond-fair	0.1	0.2	0.2	0.3	0.5	0.2	0.4	0.3	0.7	0.6	0.6	0.7	0.6	0.6	0.6	0.6
	prec	0.1	0.2	0.3	0.3	0.7	0.7	1.2	2.5	0.7	0.7	0.7	0.7	0.6	0.6	0.6	0.6
	susp-fair	0.2	0.2	0.3	0.3	0.4	0.6	0.3	0.1	0.7	0.6	0.6	0.7	0.6	0.6	0.6	0.6

Fig. 3. Summary of collected experimental data (*: MiraXT solver, using 4 cores).

rather than only on its sheer size. This can be justified, on an intuitive basis, considering that the proof of refinement is analogous to the proof of a property in traditional model-checking, where the size of the formula specifying the property has roughly the same size as the model itself.

On the other hand, the proof of the specific system properties (e.g., safety and fairness, mutual exclusion etc.) are always feasible in a "reasonable" time for all cases studies, both w.r.t. the descriptive and the operational model. It can be noticed that, in most cases, the size of the formula to be analyzed by the SAT-solver (measured in terms of Kilo-Clauses) is smaller for the operational model than for the descriptive one, and, when this occurs, in general also the time required for the proof is inferior in the case of the operational model. Indeed, for the timed lamp, Fischer's protocol, and the real-time allocator the size of the CNF boolean formula generated for operational models is 40% to 70% of the corresponding size for descriptive models. This fact directly impacts on translation times, especially for SAT2CNF. An exception is provided by the KRC, where both the number of clauses and proof time are lower for the descriptive than for the operational case. A possible reason for this could be the fact that, in designing the operational model for the KRC, we paid particular attention to partition the model into loosely-coupled modules, so to allow for a mixed refinement, where only the controller module is refined into an operational model. We conjecture that as a consequence of its modular structure, the operational model for the KRC could be redundant, e.g. having more state variables than those strictly needed.

It is easy to check that the growth in the cost of the analysis (both for the refinement correctness and for the proof of properties) is more than proportional to the size of the formulae (measured in terms of Kilo-clauses). A likely explanation of this is that the cost of the analysis derives primarily from the system's *degree of nondeterminism*: for instance, in Fischer's protocol the degree of nondeterminism grows significantly with the number of processes.

As a general comment, however, we point out that more extensive experimentations and deeper a analysis are needed to better substantiate our tentative explanations of the reported figures.

VI. Conclusions

We have illustrated and discussed a methodology for defining operational and descriptive models of time critical systems and to state a relation of correct refinement among them. The extensive use of an automatic tool has shown that the analysis of desired properties can be conducted, most often with comparable results, on both the descriptive and the operational model of the system under development.

Further developments of the method here described may originate by the remark that operational and descriptive models are not incompatible and they may be combined, as shown in the mixed refinement of the KRC example, where two modules (the train and the bar) are left in the descriptive form, while the controller module is refined into an operational version. In this line, a complex, highly structured system could be designed and developed by partitioning it into independent modules, and its analysis and verification be performed incrementally: some system components (the most critical ones or simply the ones to be implemented in hardware and/or software) could be developed by refining their requirements specification (i.e., their descriptive model) into a provably correct operational model, while the other components could be left in the more abstract form of a descriptive model, to be used as sort of "stub" or "driver" modules during the integration phases.

Acknowledgements

We thank Paolo Lischetti for developing the operational version of the Real-time Allocator.

Work partially supported by *FME Small Project*.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," *Lecture Notes in Computer Science*, vol. 1579, pp. 193–207, 1999. [Online]. Available: citeseer.ist.psu.edu/article/biere99symbolic.html

[2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *DAC '01: Proceedings of the 38th Conf. on Design automation*. New York, NY, USA: ACM Press, 2001, pp. 530–535.

[3] N. Eén and N. Sörensson, "An extensible SAT-solver." in *SAT Conference*, ser. LNCS, vol. 2919. Springer-Verlag, 2003, pp. 502–518.

[4] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *CAV '02: Proceedings of the 14th Intern. Conf. on Computer Aided Verification*. London, UK: Springer-Verlag, 2002, pp. 359–364.

[5] D. Jackson, "Automating first-order relational logic," in *ACM-SIGSOFT Symposium of Foundation of Software Engineering*. ACM Press, 2000.

[6] M. Pradella, A. Morzenti, and P. San Pietro, "The symmetry of the past and of the future: Bi-infinite time in the verification of temporal properties," in *Proc. of The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC/FSE*, Dubrovnik, Croatia, September 2007.

[7] M. Felder and A. Morzenti, "Validating real-time systems by history-checking TRIO specifications." *ACM Trans. Softw. Eng. Methodol.*, vol. 3, no. 4, pp. 308–339, 1994.

[8] A. Morzenti, M. Pradella, P. San Pietro, and P. Spoletini, "Model-checking TRIO specifications in SPIN," in *FME*, ser. Lecture Notes in Computer Science, K. Araki, S. Gnesi, and D. Mandrioli, Eds., vol. 2805. Springer, 2003, pp. 542–561.

[9] K. Y. Rozier and M. Y. Vardi, "LTL satisfiability checking," in *SPIN*, ser. Lecture Notes in Computer Science, vol. 4595. Springer, 2007, pp. 149–167.

[10] A. Gargantini and A. Morzenti, "Automated deductive requirements analysis of critical systems." *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 3, pp. 255–307, 2001.

[11] J. A. W. Kamp, *Tense Logic and the Theory of Linear Order (Ph.D. thesis)*. University of California at Los Angeles, 1968.

[12] C. Ghezzi, D. Mandrioli, and A. Morzenti, "TRIO: A logic language for executable specifications of real-time systems." *Journal of Systems and Software*, vol. 12, no. 2, pp. 107–123, 1990.

[13] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *Logical Methods in Computer Science*, vol. 2, no. 5, pp. 1–64, 2006.

[14] C. A. Furia, M. Pradella, and M. Rossi, "Dense-time MTL verification through sampling," in *Proceedings of FM'08*, ser. LNCS, vol. 5014, 2008.

[15] M. Lewis, T. Schubert, and B. Becker., "Multithreaded SAT solving," in *12th Asia and South Pacific Design Automation Conference*, 2007.

[16] L. Lamport, "A fast mutual exclusion algorithm," *ACM TOCS-Transactions On Computer Systems*, vol. 5, no. 1, pp. 1–11, 1987.

[17] C. Heitmeyer and D. Mandrioli, *Formal Methods for Real-Time Computing*. New York, NY, USA: John Wiley & Sons, Inc., 1996.