Doctoral School on Engineering Sciences
Università Politecnica delle Marche

Extended summary

Formal Methods for Practical Reverse Engineering and Software Verification

Curriculum: Ingegneria Informatica, Gestionale e dell'Automazione

Author

Francesco Spegni

Tutor(s)

Prof. Luca Spalazzi

Date: 31th January 2012

Abstract.
The software development process is committed at producing high quality software systems. In order to reach this goal it is possible to integrate formal methods software analysis and verification tools along the development process. In this work we present an integrated working environment that aims at guiding the software engineer along the most relevant moments of a software system lifetime: its development, its verification, its maintenance up to a complete re-structuring. The core of the proposed environment is the language XAL , a timed and parametric state-based language. After defining its syntax and semantics we show a novel cutoff theorem for it, proving that parametric and timed system can be model checked. We then describe two methodologies: the former helps in restructuring existing applications using XAL , extracting parameterized-timed-finite-state models from legacy code. The latter is about conducting a formal verification using XAL and its cutoff theorem, if needed. The proposed language and methodologies are used in two case-studies. The first case study describes a system for monitoring a network with many wireless devices. The second case study, instead, uses XAL in order to model check a data protection specification for a grid environment, namely the absence of privilege escalation.

Keywords.
Formal methods, parameterized model checking, refactoring, timed systems

1 Introduction

The software development process is committed at producing high quality software. Software systems have often a complex design in themselves: software infrastructures are relatively cheap to realize and always new and more complex features come to mind when designing a new system. The high degree of connectivity reached through computers and smart embedded devices, make it possible for a software system to have a very high impact on worldwide population.

All this accounts for the need of reliable means for stressing software systems and verifying their actually implementing the expected behaviors. If the system implements *more behaviors* than required, it must also be checked that such behaviors do not undermine the system infrastructure. This happens, for example, when adding a new feature to the system, that, by mistake, allows non authenticated users to access the system.

One way to control software quality is by controlling the steps during software production. Software engineering addresses this topic by comparing different software development methodologies and discussing when and how to design tests for the software components. Also very valuable is the definition of principles that guide software developers to simplify their architectures, following the basic idea that the less artifacts you have to check, and the less problems you are likely to encounter. One example of such guidelines is the so-called Don't-Repeat-Yourself (DRY) principle [1].

As some authors conjecture, the most dangerous and lethal bugs are inherently those having the least probability to be observed even on a high number of executions [2]. A long record of software faults that lead to disasters of different natures, testified the need for software systems we can rely upon [3].

This motivates the research about dependable software , whose aim is exactly to share knowledge about how to realize software we can depend upon[4] . This research leads several authors to the adoption of formal verification because it is insensible to probability of an event to occur.

In this work we present an effort for introducing formal methods into the software development process of an IT company, namely Computer VAR ITT. Together, we developed languages and methodologies trying to reuse what was already well known in the literature, integrating it with some new results when needed. In particular we focused on the integration of model checking algorithms for timed systems and parameterized systems. The problem of model checking parametric systems is known to be *undecidable* in general [5]. Model checking algorithms for timed systems and parametric systems have been defined separately, introducing suitable abstractions for those systems, whose state space is potentially infinite. We proved that systems that are *both* timed *and* parametric can be model checked as well, under some reasonable restrictions. This is at our knowledge a novel result.

In Section 2 the language XAL is defined, that is an extension of the theory of networks of Timed Automata to express parameterized timed systems. In Section 3 and Section 4, respectively, we will introduce a methodology suggesting how to use XAL in order to re-engineer an existing software system and verify a new or re-engineered software system. In Section 5 we give a description of the working environment. In particular we will underline

how the tools we realized in the context of this project and those that will be added, can be used by the software engineer in order to be guided during the software production cycle. Finally in Section 6 we will compare existing tools and notions with the ones we developed, and make some concluding remarks.

2 The XAL Language

XAL has been designed to extend the theory of *networks of timed automata* [6], [7]. XAL allows to express programs as the cooperation of finite states timed automata. With respect to networks of timed automata, XAL adds the possibility of *creating instances* and restricting the scope of the synchronizations using a notion of *local environment*. The role of local environments is to store the identifiers of acquaintances of a given automaton in XAL.

2.1 Syntax

Let us introduce a few preliminary notions, namely time constraints and id-expressions. A time constraint is an expression of the form:

$$TC_C := true | C \leqslant C | C \leqslant \mathbb{Q}^+ | TC_C \vee TC_C | \neg TC_C$$

where C denotes a set of clock variables, $\mathbb{Q}^+$ denotes the set of positive rational numbers. Id-expressions are instead terms like the following:

$$X := \emptyset | V | X \cup X | X \cap X | X \setminus X$$

while statements have the following form:

$$U := skip | V \leftarrow ? | V \leftarrow X | U ; U$$

Above, V denotes a set of local variables.

Intuitively a time constraint will be used to specify when a transition is enabled, with respect to the begin of the execution. Statements and id-expressions are used to specify how certain variables, denoting sets of instance identifiers, will be updated during the execution of a XAL program.

A XAL program has the following structure:

$$\langle \Sigma, S, V, \Gamma, \hat{s}, F, C, \tau \rangle$$

where:

- Σ is a set of symbols
- S is a set of states
- V is a set of local variables
- Γ is a set of synchronization channels
- $\hat{s} \in S$ is a distinguished initial state
- $F \subseteq S$ is the set of final states

- C is the set of local clock variables
- τ is a set of transitions

and all the mentioned sets are finite.

In XAL, programs are allowed several kinds of transitions: *symbol* transitions, *new* transitions, *send* transitions and *receive* transitions. A generic transition has the following form:

$$\tau=\{t \mid t \in S\times A\times TC_C\times 2^C\times U_V\times S\}$$

Given a transition $t=(s_1, a, \phi, R, u, s_2)$, its components have intuitively the following meaning:

- $s_1\in S$ is the source state
- a is a different action depending on the transition type:
  - $\sigma\in\Sigma\cup\{\epsilon\}$ for *symbol* transitions
  - $new(l), l\in\mathbb{N}$ for *new* transitions
  - $send(\gamma, v), \gamma\in\Gamma, v\in V$ for *send* transitions
  - $recv(\gamma, v), \gamma\in\Gamma, v\in V$ for *receive* transitions
- $\phi\in TC_C$ is the time constraint the enables/disable the transition at run-time
- $R\subseteq C$ is a set of variables that must be reset after transition is taken
- $u\in U$ is the update statement that describes how the local environment changes when the transition is taken
- $s_2\in S$ is the destination state-base

A XAL program, finally, is a collection of automaton definitions together with a distinguished main definition:

$$\langle\langle A_1 \ldots A_n\rangle, j\rangle$$

where A_i are automaton definitions and $1\leqslant j\leqslant n$ denotes the one that should be created at the being of the execution.

2.2 Operational Semantics

The semantics of XAL program interpretation is given in terms of labeled transitions systems, like usual for operational semantics [8]. Given a generic program $\langle\langle A_1 \ldots A_n\rangle, j\rangle$, the initial configuration is a term with the form $\langle L_1 \ldots L_n\rangle$ where L_i are lists of instances of dynamic size. Every item of L_i is an instance of the following form: $\langle s, clock, E\rangle$ where s denotes the current state of the instance, $clock : C\rightarrow\mathbb{R}^+$ is an assignment from local clock variables to positive reals and $E : V\rightarrow 2^{(\mathbb{N}\times\mathbb{N})}$ is the local environment that maps every variable to a set of pairs of integers. Every pair of integer *(l,i)* denotes an instance of definition *l* whose identifier is *i*.

The computation step of a configuration is defined by the following transition relation:

(delay) $\langle L_1 \ldots, L_k\rangle\rightarrow^d\langle L_1+d, \ldots, L_k+d\rangle$

where:

$d\in\mathbb{R}^+$

Above, notation $L_i + d$ denotes that all the clock assignments of instances in list L_i are incremented by an equal amount d.

(local step) $\langle \ldots L_l \ldots \rangle \rightarrow^{\lambda} \langle \ldots L_l' \ldots \rangle$

where:
$t = (s, \sigma, \phi, R, u, s')$ is a symbol-transition in definition k
$L_k(i) = \langle s, clock, E \rangle$
clock satisfies time condition ϕ
$L_k'(i) = \langle s', clock[R \leftarrow 0], E[u] \rangle$

In this last rule, $L_k(i)$ represents the i-th instance of definition k. The notation $clock[R \leftarrow 0]$ denotes a new clock assignment where variables contained in I are zeroed. Finally, $E[u]$ denotes a new environment where updates expressed by u are applied to E.

(creation) $\langle \ldots L_k \ldots L_l \ldots \rangle \rightarrow \langle \ldots L_k' \ldots L_l' \ldots \rangle$

where:
$t = (s, new(l), \phi, R, u, s')$ is a new-transition in definition k
$L_k(i) = \langle s, clock, E \rangle$
clock satisfies time condition ϕ
$L_k'(i) = \langle s', clock[R \leftarrow 0], E[NEW \leftarrow \{n_l + 1\}; u] \rangle$
$L_l' = L_l :: \langle \hat{s}_l, clock_l, \hat{E}_l \rangle$

In the above rule $E[NEW \leftarrow n_l + 1; u]$ means that before applying modification u to environment E, the variable NEW is associated with the index of the newly created instance in definition l, namely: $n_l + 1$. Furthermore, $L_l' = L_l :: \langle \hat{s}_l, \hat{clock}_l, \hat{E}_l \rangle$ means that at the end of list L_l is attached a new instance whose state is the initial state of definition l, the clock is the initial clock were all variables in C_l are assigned to zero, and the initial environment E_l associates the empty set to every variable in V_l.

(synch) $\langle \ldots L_k \ldots L_l \ldots \rangle \rightarrow \langle \ldots L_k' \ldots L_l' \ldots \rangle$

where:
$t_1 = (s_1, send(\gamma, v_1), \phi_1, R_1, u_1, s_1')$ is a send-transition in definition k
$t_2 = (s_2, recv(\gamma, v_2), \phi_2, R_2, u_2, s_2')$ is a receive-transition in definition l
$L_k(i) = \langle s, clock, E \rangle$
$L_l(j_h) = \langle s_{j_h}, clock_{j_h}, E_{j_h} \rangle$ for $h \in [1..m]$
clock satisfies time condition ϕ_1 and $clock_{j_1} \ldots clock_{j_m}$ satisfy ϕ_2
$s = s_1$ and $s_{j_1} = \ldots = s_{j_m} = s_2$
$L_k'(i) = \langle s', clock[R \leftarrow 0], E[DST \leftarrow \{(l, j_1), \ldots, (l, j_m)\}; u_1] \rangle$
$L_l'(j_h) = \langle \hat{s}_2', clock_{j_h}[R_2 \leftarrow 0], E_{j_h}[SRC \leftarrow \{(k, i)\}; u_2] \rangle$

In the synchronization rule we underline that *one* sending instance interact with a *group* of receiving instances. All of them must agree on the channel name. After the synchronization, the sending instance store in its local variable DST the identifiers of the receiving

instances, while all the receiving instances store in their local environment the identifier of the sending instance.

2.3 A logic for parametric timed processes

Let us introduce Indexed-Timed CTL*, a branching time temporal logic that unify Indexed-CTL* [9], [10] and Timed CTL [11], [12].

The terms that constitute valid formulas of IT-CTL* are the following:

$$\phi \;:=\; p(i) \mid \phi \wedge \phi \mid \neg\phi(i) \mid A\Phi \mid \wedge_i \phi$$

$$\Phi \;:=\; \phi \mid \Phi \wedge \Phi \mid \neg\Phi(i) \mid \Phi U_{\sim c} \Phi$$

The definition follows the classic definition of a branching time temporal logic [13] where the first set of formula denotes so-called *state formulas*, that is formulas that will be validated against a *single configuration*. The latter set of formulas define *path-formulas*, that is formulas whose truth value depend on a *sequence of configurations*.

Let us define a timed word as a sequence of configurations associated with a value in time: $w = (c_0,t_0)(c_1,t_1)\ldots$. A timed word is valid if (c_i,c_{i+1}) is a valid transition for the XAL program under analysis, and $t_i = t_{i+1}$ in case a non delay transition happens, otherwise $t_{i+1} = t_i + d$, where d is the delay value. A *timed run* is defined as $\rho : \mathbb{R}^+ \to Conf$, where Conf is a configuration of the program, and such that, given a timed word $w = (c_0,t_0)(c_1,t_1)\ldots$ then $\rho(t_i) = c_i$.

Using a XAL program as our Indexed-Timed Temporal Structure, we can define a satisfiability relation of Indexed-Timed CTL* formula as follows:

$$\begin{array}{lll}
c, t \mid= p(l_i) & \textit{iff} & c = \langle \ldots L_l \ldots \rangle \text{ and } L_l(i) = p \\
c, t \mid= \phi_1(l_i) \wedge \phi_2(l_i) & \textit{iff} & c \mid= \phi_1(l_i) \text{ and } c \mid= \phi_1(l_i) \\
c, t \mid= \neg\phi_1(l_i) & \textit{iff} & c \mid\neq \phi_1(l_i) \\
c, t \mid= A\Phi_1(l_i) & \textit{iff} & \text{for all paths } \rho \text{ such that } \rho(0) = c \text{ then } \rho, t \mid= \Phi_1(l_i) \\
c, t \mid= \wedge_{(l,i)} \phi_1(l_i) & \textit{iff} & \text{for any } 1 \leqslant l \leqslant k \text{ and } 1 \leqslant i \leqslant n_l \quad c, t \mid= \phi_1(l_i)
\end{array}$$

$$\begin{array}{lll}
\rho, t \mid= \phi_1(l_i) & \textit{iff} & \rho(t) \mid= \phi_1(l_i) \\
\rho, t \mid= \Phi_1(l_i) \wedge \Phi_2(l_i) & \textit{iff} & \rho, t \mid= \Phi_1(l_i) \text{ and } \rho, t \mid= \Phi_2(l_i) \\
\rho, t \mid= \neg\Phi_1(l_i) & \textit{iff} & \rho, t \mid\neq \Phi_1(l_i) \\
\rho, t \mid= \Phi_1(l_i) U_{\sim c} \Phi_2 & \textit{iff} & \exists t' : t' \geq t \wedge t' \sim c \;.\; \rho(t'), t' \mid= \Phi_2(l_i) \text{ and } \\
 & & \forall t'' : t'' \geq t \wedge t'' < t' \;.\; \rho(t''), t'' \mid= \Phi_1(l_i)
\end{array}$$

Let us underline that in state formulas, there is exactly one free variable l_i denoting the instance to which the formula is applied.

It is possible to that every XAL program can be converted onto an equivalent XAL program such that every automaton instance can stay in their initial state for an indefinite amount of time.
From this it is possible to show the XAL monotonicity lemma:

Theorem 1 (XAL monotonicity lemma)

Given a generic XAL program with definitions V_1 and V_2, and given an Indexed-Timed CTL* formula of the form $E\phi$ then:

$$(1)\forall n\geq 1 \;.\; (V_1\,V_2)^{(1,n)}|=E\phi(1_1)\Rightarrow(V_1\,V_2)^{(1,n+1)}|=E\phi(1_1)$$

$$(2)\forall n\geq 1 \;.\; (V_1\,V_2)^{(1,n)}|=E\phi(1_2)\Rightarrow(V_1\,V_2)^{(1,n+1)}|=E\phi(1_2)$$

The monotonicity lemma states that adding an instance to the system, a valid formula remains valid. Since A-formulas, that is formulas quantified over all-paths, are definable as $A\phi(l_i)=\neg E\neg\phi(l_i)$, the result holds also for A-formulas.

Theorem 2 (XAL bounding lemma)
Given a generic XAL program with definitions V_1 and V_2, and given an Indexed-Timed CTL* formula of the form $E\phi$, let us assume that $c_1 = |V_2| + 1$, $c_2 = |V_2| + 2$, then:

$$(1)\forall n\geq c_1 \;.\; (V_1\,V_2)^{(1,n)}|=E\phi(1_1)\Rightarrow(V_1\,V_2)^{(1,c_1)}|=E\phi(1_1)$$

$$(2)\forall n\geq c_2 \;.\; (V_1\,V_2)^{(1,n)}|=E\phi(1_1)\Rightarrow(V_1\,V_2)^{(1,c_2)}|=E\phi(1_1)$$

This states that there is no need to verify a property on a system bigger than the cutoff size c_1 or c_2, depending on the property.

3 Re-engineering using XAL

As already stated, one of the research area where we intend to apply the XAL language is the reengineering of an existing Software System, often called refactoring.

Restructuring a software system can be an expensive operation. Analogously to the usual software developing processes, the restructuring task can be thought as a cycle composed of several phases [14]:

1. Identify a new aspect of the system that deserves an improvement;
2. Isolate the components that need to be modified and the refactoring operations that should be applied;
3. Verify that the applied modifications preserves the behaviors of the system;
4. Estimate the effects of the change on the software or development process;
5. Update other software artifacts that depends on the code and may be inconsistent after the applied change (e.g. the documentation).

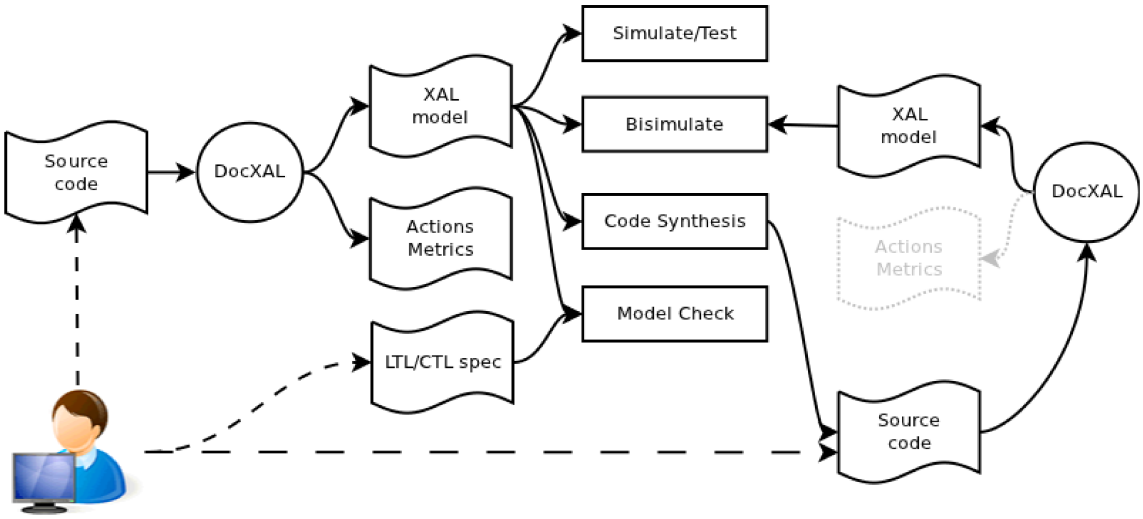

Figure 1: Re-engineering architecture based on XAL

In order to face the refactoring of a complex Software System, every functional and non-functional requirement of the system must be reconsidered, and associated with the components that implement each of them. A *reverse engineering* phase is thus needed, especially if the Software System is quite old. Such phase is then followed by the recoding phase. All this considered, the problem of refactoring a real-world Software System is likely to be overwhelming for a single programmer (or even a team of programmers). Although there cannot be such a thing as a silver-bullet solution to the problem of refactoring complex systems, whatever the available tools, formal methods can be a useful guidance for the developing team along the whole process.

The key idea of our architecture is to extract a XAL model from existing source code. The proposed methodology is depicted in Figure 1. In the proposed approach the engineer submit the source code to be analyzed. The DocXAL architecture extract the XAL model from it, and a collection of executable code., that would make the XAL model an executable program. Once the model is extracted, several actions are available: the code can be simulated to test its behavior and obtain a better insight of what the code is supposed to do. In case the project has been manually re-engineered, a new model can be extracted from the new source code and compared with the previous XAL model, to check whether they bisimulate each other, thus certifying the correct reimplementation of the code. If code was not rewritten manually, code synthesis techniques could be adopted [15]. The engineer may also provide temporal logic specification in order to model check the extracted model and verify it is correct.

To our knowledge this architecture, both in its manual and automatic use, is a novel approach to software refactoring and re-engineering.

4 Verifying using XAL

A verification methodology is usually a process operating on an object, the system under analysis, and trying to extract from it some knowledge with respect to a given hypothesis.

The methodology outcome is usually a yes/no answer but the system and the hypothesis can be expressed in several different ways. At some extent, if the methodology is automatized through proper tools, it can be seen as a means to obtain more insights on designed software. In principle, in fact, it may be used to ask questions about the computer system (i.e. compliance to some specification) and obtain a total or partial answer.

An architecture overview is sketched that describes how to use the available tools, in what sequence, and what data need to be provided. This overview is the *verification workflow*.

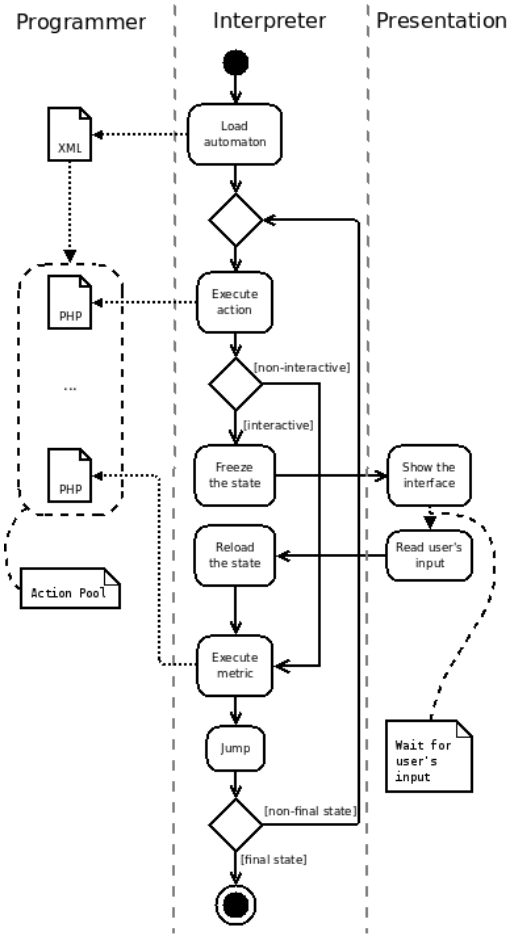

Figure 1: The XAL interpreter

Formalize specifications.

A first step in our verification methodology is to find a suitable Indexed-Timed CTL* formula that matches the intended specifications. For this task it is possible to use any formula allowed by Theorem 2.

Splitting.

XAL programs allow the designer to introduce variables that will restrict the scope of send/receive actions at run-time. Based on this it is possible to split a given definition in two sets of instances: those that will interact with another instance and those that will not.

Remove.

Given a formal specification, it is possible to isolate those definitions that interact, directly or indirectly, with the definitions mentioned in the specification. Automaton that do not interact can be abstracted away.

Reducing.

This stage consists in the application of Theorem 2, thus reducing a parametric system to a fixed size system, computing the cutoff for every definition, depending on its role with respect to the specification under analysis.

Model checking.

This is the final part of the work. The model must be converted onto a suitable specification for the chosen model checker. In our experiments we used UPPAAL for timed models, and NuSMV for untimed ones.

5 A working environment

At the moment, the implemented working environment consists of:

- the XAL interpreter
- the XAL IDE
- the DocXAL IDE

The XAL interpreter implements the XAL operational semantics described in Section 2.2. In order to make XAL models executable, every state in the specified automata can be attached to pieces of code written in PHP, representing the states behavior.

At the moment the synchronization among the instances must be implemented. It is instead possible to execute the XAL program interactively. This means that at every step the computation can stop waiting for a user to input some data. Depending on it, the XAL program can decide how to continue its computation. We are currently analyzing how to implement a step-by-step execution mode, in order to allow easier debugging facilities.

The XAL IDE allows now to generate XAL programs by drawing it. The IDE is realized as an Eclipse plugin. This on one side allows the programmer to switch from a high-level view, working on the XAL model of her/his programs, to a low-level view of the executable language used with XAL. On the other side allows to maintain a single-source project, and share the view and edit functionalities with the DocXAL IDE, that is instead meant to be used as a web application.

The DocXAL IDE allows the software engineer to send her/his source code to the DocXAL service and wait for its analysis to be completed. Since it is a rather complex task, it is preferable to run it on a remote machine. A messaging mechanism will notice the engineer when the analysis will terminate. Here the Rich Ajax Platform allows to reuse the XAL IDE code developed within the Eclipse-RPC, thus providing a unified interface to the engineer when using the XAL IDE on her/his own working station and the DocXAL IDE service.

6 Conclusions

One main result of this work is how theories of timed systems and parametric systems can be integrated. Often, it is possible to find in literature partial solutions to real-world problems, but then those partial solutions must be combined together and must be shown that the outcome of such combination is still a suitable framework.

Our cutoff for timed and parametric systems is a novel results, as far as we know. Other authors approached the problem of verifying parametric timed processes [16] but in their case the specifications can be expressed only in terms of *unsafe states* that must not be reached. In our result we allow a more flexible language for specifying properties to be checked, namely a subset of Indexed-Timed CTL*.

The restructuring methodology can be compared to existing tools for reverse engineering and design recovery, or other tools that extract models from source code. An example of the latter is Bandera [17]that extracts finite-state models from Java source code. It differs from our workflow because its main goal is to model check obtained models, thus they preprocess the program with a slicing phase followed by an abstraction phase. In our case, instead, the extracted XAL model is intended to be as complete as possible. The en-

gineer will decide, eventually, to abstract some parts of the code to consider them as a monolithic block that does not deserve to be analyzed and decomposed.

Other means for software translation, like TXL [18] which is a general purpose program rewriting grammar generator, are more focused on the translation mechanisms. On the contrary, our platform aims at providing an entire tool-chain that possibly guides the engineer. Nevertheless, it should be considered the possibility of letting the programmer to plug her/his own TXL transformation specification, in order to make the framework extensible.

About the verification methodology that we specified, it seems to us a novel approach to software verification. As far as we know, the basic verification tools are very well known, but none or few attempts have been made in order to integrate them in a developing framework to let the engineer use them embodied in her/his own preferred IDE, side by side with the debugger, for example. The exposed methodology is now a sketch of a wizard-like procedure that we are going to implement and test on real-world case-studies in order to better evaluate the impact of software verification when used on a day-by-day basis, during software programming.

References

[1] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[2] G. J. Holzmann, "Economics of software verification," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 80–89.

[3] R. I. Cook and M. F. O'Connor, "Thinking about accidents and systems," in *Improving medication safety. Mathesda, MD: American Society for Health-System Pharmacists*, 2005, pp. 1-21.

[4] D. Jackson, "A direct path to dependable software," *Communications of the ACM*, vol. 52, no. 4, p. 78, Apr. 2009.

[5] K. R. Apt and D. C. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Information Processing Letters*, vol. 22, pp. 307–309, 1986.

[6] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," Lectures on Concurrency and Petri Nets, vol. 3098, pp. 87-124, Springer, 2004.

[7] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183-235, Apr. 1994.

[8] G. D. Plotkin, "A structural approach to operational semantics," *Technical report, Dept. of Computer Science, Univ. of Aarhus, 1981, Reprinted in Journal of Logic and Algebraic Programming*, vol. 60-61. , pp. 3-15, Elsevier, 2004.

[9] E. M. Clarke, O. Grumberg, and M. C. Browne, "Reasoning about networks with many identical finite-state processes," *Proceedings of the fifth annual ACM symposium on Principles of distributed computing - PODC '86*, pp. 240-248, 1986.

[10] A. E. Emerson and V. Kahlon, "Reducing model checking of the many to the few," *Automated Deduction-CADE-17*, pp. 236–254, 2000.

[11] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pp. 414–425, 1990.

[12] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, vol. 111, p. 193--244, 1994.

[13] A. E. Emerson, "Temporal and modal logic - Handbook of Theoretical Computer Science," vol. 5, no. 3, Elsevier, 1995, pp. 995-1072.

[14] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.

[15] I. Technology, "Code Synthesis for Timed Automata,", Nordic Journal of Computing, vol. 9 (4), Winter 2002

[16] P. A. Abdulla and B. Jonsson, "Model checking of systems with many identical timed processes," *Theoretical Computer Science*, vol. 290, no. 1, pp. 241-264, Jan. 2003.

[17] J. Corbett, M. Dwyer, and J. Hatcliff, "Bandera: Extracting finite-state models from Java source code," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000, p. 439-448.

[18] J. R. Cordy, T. R. Dean, A. J. Malton, and K. a Schneider, "Source transformation in software engineering using the TXL transformation system," *Information and Software Technology*, vol. 44, no. 13, pp. 827-837, Oct. 2002.