# TLA$^+$ 2012: International Workshop On The TLA$^+$ Method And Tools

Leslie Lamport and Stephan Merz

August 27, 2012

The TLA$^+$ workshop 2012 is organized as a satellite event of FM 2012, the 18th International Symposium on Formal Methods, in Paris. It is intended as a forum for practitioners and researchers interested in the use and further development of the TLA+ specification language and its associated tools. The present collection contains abstracts of the nine submissions that are presented at the workshop. The contributions present current tool developments, experiences gained with the use of TLA$^+$ in industry and academia, and reports on the use of TLA$^+$ in university courses.

The program committee of the workshop consists of:

- Damien Doligez, Inria, Paris, France
- John Douceur, Microsoft Research, Redmond, U.S.A.
- Leslie Lamport, Microsoft Research, Mountain View, U.S.A.
- Stephan Merz, Inria, Nancy, France
- Chris Newcombe, Amazon, Seattle, U.S.A.
- Werner Stephan, DFKI, Saarbrcken, Germany

We are grateful to the chairs and organizers of FM 2012 for having selected TLA$^+$ 2012 as a satellite event of the main symposium, and for the support with the logistics of the workshop, including the printing of the present collection.

# Contents

# TLA2B: A New Validation Tool for TLA$^+$

*Dominik Hansen and Michael Leuschel*
*Institut für Informatik, Universität Düsseldorf*[*]
*Universitätsstr. 1, D-40225 Düsseldorf*

## 1 Architecture

TLA$^+$ and B are both state-based formal methods rooted in predicate logic, combined with arithmetic and set theory. In this paper we present a new tool for TLA$^+$ based on a translation to B. The translator is called TLA2B and builds upon the SANY parser to translate a TLA$^+$ module, along with an optional TLC configuration file, into a B machine. TLA2B supports both TLA$^+$'s concepts of modularization (EXTENDS, INSTANCE) and TLC's ability to modify an existing module in the configuration file (constant/definition assignment and replacement). The translator has been integrated into the ProB validation tool, providing animation, model checking, constraint solving and graphical visualization capabilities. The Atelier-B provers can also be applied to the B translation.

A main difference between TLA$^+$ and B are the concepts of typing. While TLA$^+$ is untyped, B is strongly typed. As such, we had to develop a type inference algorithm for TLA$^+$. TLA2B supports model values and distinguishes between integers, boolean values, strings, sets, functions, records and sequences. The B typing requires all elements of a set to have the same type. This also restricts functions and sequences, which are based on sets. The TLA2B type inference algorithm automatically adds missing type declarations for variables and constants to achieve a valid B machine. Declarations in the configuration file are also taken into account to infer the types of constants.

Due to the common foundation of TLA$^+$ and B, most TLA$^+$ built-in operators can be mapped to B operators. This includes operators of the standard module Naturals, Integers and Sequences. Other operators such as IF/THEN/ELSE or EXCEPT can be expressed by a combination of B operators. An exception is TLA$^+$'s CHOOSE operator. In B there is no way to express its general functionality. However, the ProB tool, upon which we build, supports externally defined (polymorphic) functions. This enabled us to encode the CHOOSE operator as such an external function, whose semantics is expressed in the ProB kernel. In TLA$^+$ the CHOOSE operator is often combined with recursive functions such as determining the sum of a set. Currently ProB only supports constant recursive functions and TLA2B can only translate a standard pattern (a recursive function using the IF/THEN/ELSE construct) to the supported scheme. Certain other patterns can be directly translated to B built-in operators (e.g., sum or product of a set).

[*]Part of this research has been sponsored by the EU funded FP7 projects 214158 (DEPLOY) and 287563 (ADVANCE).

## 2 Features and Case Studies

TLA2B has been integrated into ProB[1]; opening a TLA$^+$ module ProB invokes TLA2B to translate the module. While displaying the module in the editor, ProB runs the translated B machine in the background. All of ProB's features are also applicable to TLA$^+$ modules: evaluation of predicates and expressions in an interactive console,[2] graphical visualization of the full or reduced state space, visualization of a single state, visualization of predicates such as the pre-condition of an operation, constraint-based deadlock and invariant checking, and of course model checking with automatically inferred symmetry reduction.

We believe the new tool to be complementary to TLC: On the one hand, TLC is very good at explicit state model checking on very large state spaces, thanks to its efficient disk based algorithm. ProB is generally slower for explicit state model checking. On the other hand, it seems that TLC's symmetry reduction does not cope well in the presence of many symmetrical states (incidentally, a situation where symmetry reduction could be particularly useful). Here, ProB seems to scale much better. ProB also provides many additional features complementary to TLC (see above). Finally, a key ingredient of ProB is its ability to solve constraints. This opens up possibilities such as constraint-based deadlock or invariant checking. It also enables the tool to validate very high-level specifications with which TLC cannot cope. An interesting example is a TLA$^+$ model for graph isomorphism specification. TLC finds the first isomorphism after over two hours, our translator combined with ProB finds all 8 isomorphisms in less than a second. More examples and experimental results can be found in a technical report at: `http://www.stups.uni-duesseldorf.de/w/Special:Publication/HansenLeuschelTLA2012`, which is an extended version of an article at iFM'2012. Note, however, that since then, we have further extended the tool to support, e.g., the CHOOSE operator and a limited form of recursive functions. At the workshop we want to present the tool to the TLA$^+$ community, in the hope of getting feedback about the usefulness of the tool and to guide the future research and development. Indeed, we want to discuss several possible avenues of further developments: support for real numbers and the temporal operators, relaxing of the strong typing requirement, integration into the TLA$^+$ toolbox, improved symmetry reduction, application of ProB's constraint solving to validation tasks, etc. Finally, we believe that a more extensive exchange of ideas between the TLA$^+$ and B communities would be very fruitful to both.

[1] See `http://www.stups.uni-duesseldorf.de/ProB/index.php5/TLA` for instructions.

[2] See also `http://www.stups.uni-duesseldorf.de/ProB/index.php5/ProB_Logic_Calculator`.

# Current State of Distributed TLC

*Markus A. Kuppe*
*Department of Computer Science, University of Hamburg*

TLC – the explicit state model checker component of the TLA+ tools package – detects errors in a finite-state model of a TLA+ specification. To do so, it traverses the dynamically generated state space of the model and checks the specified safety properties for every state found. The shape and size of the state space is directly proportional to the size of the model. Thus, it is paramount for a model checker to come with an efficient implementation to handle even large models. Still, the model size that a (sequential) model checker can handle is bounded by the hardware it is running on.

To exploit current day distributed environments on commodity hardware has been the task of a multi month long internship at MSR-INRIA joint centre and Microsoft Research during which the implementation of the TLC model checker has been analyzed, improved and prepared for distributed deployments on hundreds of nodes. Additionally, a seamless integration of distributed TLC into the existing TLA+ Toolbox has been part of the internship.

This presentation will show the results and findings of this work. To motivate this and visualize the potential areas for distribution, the first part of this presentation will start with the basics of (parallel) model checker algorithms. This helps to present the architecture of the TLC model checkers which includes iterating of its elementary building blocks. Furthermore limitation as well as implementation shortcomings of distributed TLC will be named.

The second part of the presentation will discuss the performance results that have been measured by experimental benchmark analyses of the TLC model checker including a brief discussion of the benchmarking model and its characteristics. It also contrasts the performance results of various distributed TLC's modes/variations and ends with heuristics and lessons learned from running distributed TLC on research networks like Grid5000 or commercial cloud vendors.

The second part bridges into a live demo of distributed TLC to exemplify the seamless integration into the TLA+ Toolbox as well as to show the low barrier to running TLC models distributively. Special emphasis is put on how to deploy a distributed experiment in a cloud environment and how CPU, RAM, and I/O resources are best allocated for optimal performance. Fault-tolerance trade-offs will be discussed too.

Finally the presentation will outline, how these results will help to improve the performance of (distributed) TLC in future work. For example, faster data structures with better distribution characteristics and different algorithms will be in the scope of this section.

This presentation summarizes the current state of the (distributed) TLC model checker and the TLA+ Toolbox and collects targeted areas for further enhancement which will be part of my masters thesis in the next month to come.

# Harnessing SMT Solvers for TLA$^+$ Proofs

*Stephan Merz*[1] *and Hernán Vanzetto*[1,2]

[1] *Inria Nancy Grand-Est & LORIA, Villers-lès-Nancy, France*
[2] *Microsoft Research-INRIA Joint Lab, Saclay, France*

**Introduction.** The TLA$^+$ proof system TLAPS [1] is an interactive proof environment in which users can deductively verify safety properties of TLA$^+$ specifications. TLA$^+$ contains a declarative language for writing hierarchical proofs, and TLAPS is built around a *proof manager*, which interprets the proofs, expands the necessary module and operator definitions, generates corresponding proof obligations, and passes them to backend verifiers. While TLAPS is a proof assistant that relies on users guiding the proof effort, it integrates automatic backend provers to discharge proof obligations that users consider trivial. The main backends of the current version of TLAPS are Zenon, a tableau prover for first-order logic with equality that includes extensions for TLA$^+$ for reasoning about sets and functions, and Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ in the Isabelle proof assistant, which provides automated proof methods based on first-order reasoning and rewriting. The current release of TLAPS also provides a backend [3] for the use of SMT (satisfiability modulo theories) solvers for discharging "shallow" proof obligations mixing set theory, functions, and integer arithmetic.

In this contribution, we outline two approaches for encoding (non-temporal) TLA$^+$ proof obligations into SMT input languages. Since TLA$^+$ is untyped whereas SMT input languages are sorted, the main challenge consists in assigning SMT sorts to the subexpressions of a TLA$^+$ proof obligation. The released backend relies on typing hypotheses for constants, variables. and operators. Work in progress, based on a suggestion by Ken McMillan [2], does not need typing hypotheses, but requires the SMT solver to handle more quantified formulas.

**From TLA$^+$ to SMT input.** The input languages of state-of-the-art SMT solvers[1] are based on many-sorted first-order logic. This allows us to design a generic translation from TLA$^+$ expressions to an intermediate language, from which the translation to the actual target languages of particular SMT solvers is straightforward. TLA$^+$ formulas are translated to quantified first-order formulas over the theory of linear integer arithmetic, extended with free sort and function symbols. In particular, we make heavy use of uninterpreted functions and quantified formulas.

The first approach for translation [3] relies on a typing discipline, which is compatible with the logics of SMT solvers. The translation of the proof obligation makes use of types assigned to expressions. For example, equality between two integer expressions will be translated differently from equality between two sets or two functions. Observe that correct type assignment is relevant for soundness: a proof obligation that is unprovable according to the semantics of untyped TLA$^+$ must not become provable due to incorrect type annotation. As a trivial example, consider the formula $x+0=x$, which should be provable only if $x$ is known to be of arithmetic sort. Therefore, the

[1] Our backends translate to SMT-LIB, the *de facto* standard input format for SMT solvers, as well as to an extension of SMT-LIB for Z3, and to the native input language of Yices.

type inference algorithm essentially relies on hypotheses of the proof obligation that ascertain the types of symbols (variables or operators). These *typing hypotheses* are of the forms $x \approx exp$ and $\forall \vec{y} \in \vec{S} : f(\vec{y}) \approx exp$, where $\approx \in \{=, \in, \subseteq\}$ and *exp* is an expression whose type can already be inferred. Type inference may fail because not every set-theoretic expression is typable according to our typing discipline, and in this case the backend aborts. This approach requires the presence of typing hypotheses even where they are logically not necessary.

We have recently studied a different approach, suggested in [2], for an SMT translation that does not require inferring types for TLA$^+$ expressions. Instead, expressions are encoded as SMT terms of a single sort $\mathcal{U}$, representing the universe of TLA$^+$ values. For connecting the SMT integer sort with $\mathcal{U}$, we declare an uninterpreted, injective function $\phi : Int \to \mathcal{U}$ that maps SMT integers to TLA$^+$ values. For example, the axiom $\forall x, y : Int.\ \phi(x) +_{\mathcal{U}} \phi(y) = \phi(x+y)$ defines TLA$^+$ addition $+_{\mathcal{U}}$ over the image of $\phi$, where $+$ on the right-hand side denotes the built-in addition over SMT integers. In this way, the link between SMT operations and their TLA$^+$ counterparts is effectively defined only for embedded values, and type inference is performed by the SMT solver during the proof attempt.

However, the "lifting" of operators as in the addition axiom introduces quantified formulas in the input to the SMT solver that are not needed for the translation based on type assignments. It is not hard to understand the soundness of this encoding, given that is based on a small set of fixed axioms, and that the translation is essentially a syntactic rewriting.

**Preliminary results.** We have used both SMT encodings with good success on several examples that had previously been proved interactively using TLAPS and have observed significant reductions of proof sizes and running times. Preliminary results show that automation can be significantly improved by using SMT solvers for the verification of "shallow" TLA$^+$ proof obligations. The backends can handle a useful fragment of TLA$^+$, including first-order logic, sets, functions, linear arithmetic, records, tuples, except that the typed encoding does not currently handle sets of sets.

Comparing the two encodings, the translation based on type inference appears to be more efficient when it works, and allows the backend to handle large proof obligations that no other backend can currently handle. As explained above, type inference may sometimes fail, and logically valid obligations cannot be proved without additional typing hypotheses. In such cases, the second encoding works better and may therefore be preferred by users of TLA$^+$ who are used to an untyped framework. We are currently working on combining both encodings to get the best of the two methods. The version of the SMT backend relying on type inference is currently available at `http://www.msr-inria.inria.fr/~doligez/tlaps/` as part of the release of TLAPS.

## References


[1] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. *Verifying Safety Properties with the TLA$^+$ Proof System.* 5th IJCAR. LNCS 6173, Springer, 2010.

[2] Ken McMillan. *A Proposal for Translating TLA$^+$ to SMT.* Unpublished, 2011.

[3] Stephan Merz and Hernán Vanzetto. *Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers.* 18th LPAR. LNCS 7180, Springer, 2012.

# Experience of Software Engineers Using TLA+, PlusCal and TLC

*Chris Newcombe, Principal Engineer*
*Amazon Web Services (database services group)*

Amazon now has four teams using TLA+ and PlusCal, and adoption is increasing. We have specified the subtle parts of several large distributed systems, and a few non-distributed concurrent algorithms. We have successfully used Distributed TLC to find subtle bugs in the design of customer-facing services, before launch.

This short talk will cover:

- Effective ways to persuade engineers to try something as apparently esoteric as TLA+
- Common misconceptions of engineers when learning TLA+
- Suggestions that may help encourage further adoption

## 1 Persuading engineers

Engineers in industry have very little time to experiment with new tools or methods. To persuade one or two people to even try, it is necessary to have a believable real-world example. The only persuasive public example we have found so far is Pamela Zave's work on model-checking Chord, using Alloy. To achieve real adoption, it is vital that the first in-house trials of the method demonstrate tangible value.

## 2 Common misconceptions

Engineers have different established meanings for some of the terminology. A common question is, "So, 'specification' means inputs and outputs, right?". Also, engineers tend to think in terms of the concepts offered by their favorite programming languages. To bridge from programming to TLA+ we have found it helpful to refer to TLA+ and PlusCal as "exhaustively-testable pseudo-code". Shortly after, we introduce the unfamiliar 'style' of pseudo-code, e.g. "a TLA+ 'specification' is pseudo-code that *recognizes* if a particular behavior is in the set of 'good' behaviors". When using PlusCal, we push engineers to read the translated TLA+, to 'understand the math', not just treat PlusCal as another programming language.

We remind the audience of the 'levels of abstraction' with which they are already familiar; design down through Java or C++ code, then perhaps JVM byte-code, then machine-code, micro-code, transistors. This establishes context to later expand the 'design' layer to include high-level properties, high-level specifications, and lower-level specifications. This also allows us to raise and then address a common confusion about refinement; "How can an implementation, with its vast amount of state, be exhibiting a *subset* of the behaviors of a very simple high-level specification or property?".

Other common concerns include: the focus on global state and atomic actions ("where is the concurrency?"), how to introduce non-determinism, the pros and cons of 'interleaving' vs. 'non-interleaving' specifications, the difficulty of expressing 'intuitive' notions of correctness as precise properties, the difficulty in choosing granularity and names of actions, accidentally writing actions that access the state of other processes (e.g. to obtain data that should have been sent as part of a message payload).

Engineers are so trained to worry about computational efficiency that this can affect the style of TLA+ expressions that they write. E.g. Engineers aren't used to contemplating sets of all possible functions of a particular type, or defining things in terms of powersets; they may avoid using these simply because they worry about how TLC will cope with them. Another kind of blindspot is the power of TLA+ expressions; engineers often start out by writing 'functional programs' as TLA+ operators, instead of using set-comprehensions.

We also have some more advanced conceptual questions. E.g. Liveness is a property of an infinitely-long behavior, so what does TLC's "liveness checking" actually tell us? How can we tell if it is safe to use a symmetry-set when model-checking? How can TLC check inductive invariance, when by definition an inductive invariant has to be preserved by the Next action even in *non-reachable* states? Are there heuristics to help find an inductive invariant?

# 3 Suggestions to encourage further adoption

We would welcome more real-world examples; the recent work on Pastry is very useful. Syntax highlighting for PlusCal. A cookbook of 'specification idioms'. A cookbook of ways to use and trouble-shoot TLC. More examples of relatively small but illustrative proofs, including informal hierarchical proofs (most engineers haven't done any kind of written proof in a long time.) Tools to deploy distributed TLC quickly and easily (we may be able to help with that).

In the longer term, it would be great to have some kind of "performance modeling" feature. For example, the ability to assign a 'cost class' to each action - reads from disk N times, sends N network messages, uses X seconds of CPU-time - then have TLC output 'performance' information.

# Formal Verification of Pastry Using TLA$^+$

*Tianxiang Lu[1,2], Stephan Merz[1], Christoph Weidenbach[2]*

[1] *Inria & LORIA, Villers-lès-Nancy, France*
[2] *Max-Planck-Institut für Informatik, Saarbrücken, Germany*

## 1 Motivation

Pastry [1, 2] is an algorithm that provides a scalable distributed hash table over an underlying P2P network. Several implementations of Pastry are available, but to the best of our knowledge the correctness of the algorithm has not been verified formally. Since Pastry combines complex data structures, asynchronous communication, concurrency, resilience to churn and fault tolerance, we believe that it makes an interesting target for verification using TLA$^+$.

## 2 Contributions

We have modeled a significant part of the Pastry algorithm, including message routing and joining and departure of nodes. No correctness properties are asserted in [1]; our main objective is to verify that there can never be two different Pastry nodes that consider themselves responsible for any single key.

The TLC model checker was an invaluable help for understanding the behavior of the algorithm and for validating our TLA$^+$ model, which are based on different versions of the algorithm, since every single description lacked important details. We used TLC for discovering candidate properties and, more frequently, non-properties [3].

After TLC could find no more errors, we embarked on a formal proof of the property, for arbitrary instances. In a first step [4], we postulated hypothetical invariants of the underlying data structures, and used TLAPS to prove that these imply our global correctness property. Again, TLC helped us fine-tune the formulation of these invariants, and verify them over small instances. We have meanwhile refined these invariant properties to lower-level invariants and proved that these are indeed inductive for a restricted model where no nodes are allowed to leave the network and where nodes are assumed not to join concurrently in the same region of the Pastry ring. Our current models consist of about 1kloc for the specification and 15kloc for the proof. We plan to progressively relax the constraints so that nodes can freely leave and join the network.

In related work, Zave [5] has used Alloy to model Chord at a much higher level of abstraction where operations such as *join* or *stabilize* are considered atomic and non-interfering. Focusing on eventual consistency, she has found a flaw in the original description of the algorithm and suggests a repair that may be correct—however, Alloy is not supported by a theorem prover like TLAPS to formally prove the invariants.

# 3 Our Experience With The TLA$^+$ Tools

The trace exploration features provided by TLC through the TLA$^+$ toolbox were invaluable for understanding counter-examples produced for corner cases and for improving our models. It might be possible to improve the display of the (part of an) action that is causing a state transition, perhaps by "zooming into" the action predicate that is evaluated.

Once TLC did no longer quickly produce error traces, we turned to the command-line version and let the model checker run on a 8-processor server. Our models generate more than 30 billion states even for instances with just four nodes; hash collisions are therefore highly probable. The use of several worker threads during state exploration led to a significant speed-up and worked without a glitch. However, after letting TLC run for weeks, the process sometimes ended up with a huge memory footprint but without any more visible progress or even CPU usage. We suspect this to be a problem of the Java run-time and are eager to use the new distributed version of TLC.

TLAPS improved significantly since we started this work around the end of 2009. Use of the new SMT-based backend may help us shorten our proofs, which contain significant arithmetic reasoning. The toolbox was very helpful for zooming into parts of the proof, for non-linear proof editing, and for jumping back and forth between definitions and proofs. Checking the status of a proof works well for small and medium-sized proofs, thanks to fingerprinting. However, the memory footprint of the Eclipse editor and the proof manager can become critical for large proofs. For example, our main invariant proof has about 12500 lines, and even allocating 4GB of main memory to Eclipse is barely enough to generate all proof obligations for status checking. We believe that it should be possible to reduce the memory overhead, and perhaps generate proof obligations incrementally rather than upfront.

# References


[1] M. Castro, M. Costa, and A. I. T. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *DSN 2004*, pages 9–18, Florence, Italy, 2004. IEEE.

[2] A. Haeberlen, J. Hoye, A. Mislove, and P. Druschel. Consistent key mapping in structured overlays. Technical Report TR05-456, Rice University, Department of Computer Science, August 2005.

[3] T. Lu, S. Merz, and C. Weidenbach. Model checking the Pastry routing protocol. In *AVOCS 2010)*, pages 19–21, Düsseldorf, Germany, September 2010. Short contribution.

[4] T. Lu, S. Merz, and C. Weidenbach. Towards verification of the Pastry routing protocol using TLA$^+$. In *FMOODS/FORTE 2011*, volume 6722 of *LNCS*, pages 244–258, Reykjavik, Iceland, 2011. Springer.

[5] P. Zave. Using lightweight modeling to understand Chord. *Computer Communication Review*, 42(2):49–57, 2012.

# Inserting Intentional Bugs for Model Checking Assurance

*Thomas L. Rodeheffer and Ramakrishna Kotla*
*Microsoft Research, Silicon Valley*

Writing a formal specification for a system or system component forces one to be precise about the system's actions, and this process often flushes out design bugs just by itself. But once the specification is written, it is usually desirable to check that the specification conforms to some concept of correctness.

One way of checking a specification is to write invariants, or safety properties, and then prove that the specification maintains the invariants. Of course, there could be oversights in the proof, so to be certain that the proof is correct, it must be a formal proof checked by a mechanical proof checker. Unfortunately, such formal proofs tend to be lengthy and tedious, since the limited deductive power of current mechanical proof checkers requires detailed proof steps. So it is often better to be fairly confident that the specification is correct before investing the effort in writing a formal proof.

To get confidence that the specification maintains the invariants, one can use a model checker to explore the state space. Unfortunately, most interesting systems have specifications whose state space is enormous, if not infinite, and thus a model checker is limited to exploring a few relatively small, constrained configurations. If the model checker finds no errors, that is well and good, but if there was an error, would the model checker have explored a rich enough configuration to find it? One way to get some assurance of this is to introduce some errors on purpose, and see if the model checker finds them.

The full paper presents the results of model checking, with inserted errors, a TLA+ specification for a node in Pasture, a messaging library that provides secure offline access to data using a TPM. The model checking results give some assurance that the specification is correct; that is, that it maintains its invariants. The full paper also presents a formal proof of correctness, checked by the TLA+ Proof System.

The Pasture messaging library [5] provides secure offline access to data by using a Trusted Platform Module (TPM) [1, 2] in the user's computer to protect Pasture decryption keys and to maintain a cryptographic log of the user's decisions to obtain or revoke access to these keys. The corresponding encryption keys can be used by correct peers to send encrypted data to the user who can then make an offline decision to obtain or revoke access to the data by obtaining or revoking access to the protected Pasture decryption keys.

Pasture provides two safety properties: *access undeniability* and *verifiable revocation*. Access undeniability means that a user cannot deny any decision to obtain access to a Pasture decryption key and still survive an audit. Verifiable revocation means that a user can provide a *proof of revocation* for any decision made to revoke access to a Pasture decryption key. This proof establishes that the user never did and never will be able to access that key.

Since some of the internal state in a TPM is volatile, and is reset to its initial value on reboot, the main difficulty faced by Pasture is how to preserve its state across reboots. If an adversary could rollback Pasture state to an earlier point, arranging to vio-

late Pasture's safety properties of access undeniability and verifiable revocation would be easy.

Memoir [8] presented a general solution to this problem for any deterministic application. Pasture exploits the specific nature of its application to obtain a solution with much less overhead in its particular case.

When a complicated system is supposed to maintain a safety property in an adversarial setting, having some assurance that the system is correct is crucial. We wrote a formal specification of a Pasture node in TLA+ [7] and checked it using the TLC model checker. We used the TLA+ toolbox [6] to construct and manage models.

As expected, the number of distinct states and consequently the model checking run time increased enormously as the configuration parameters were increased. This limited the feasibility of model checking of the Pasture node specification to small configurations only. None of our model checking runs found any violation of the Pasture safety properties.

Normally, we like to check models with configuration parameter values up to at least three. Often a system will have interesting behavior when there is the chance for three instances of something to interact. In configurations of the Pasture specification, for example, this would mean up to three node reboots, three Pasture decryption keys, and so on. But in model checking the Pasture specification, before we even got close to such configurations, the model checking runs were taking many, many hours. This was disappointing.

So we intentionally added a bug to the specification to see if the model checker could find a violation within the small configurations that were feasible to check. The idea was to start with the smallest configuration and then carefully increase the configuration parameters until the model checker found a violation. We did this for 16 different bugs, and in 13 cases, the model checker indeed found a violation in a small configuration within seconds of run time.

We investigated the three other cases, and, to our surprise, these "bugs" turned out not to lead to safety violations, although they caused other problems. One bug, for example, made continued operation impossible after a reboot, which is a liveness violation but not a safety violation. The two other bugs made it possible for a node to retract "phantom" entries on its log that had no relation to obtaining or revoking access to Pasture decryption keys: strange behavior, but, again, not a safety violation.

Once we were confident that the Pasture node specification was correct, we proceeded to write a formal correctness proof and check it using the TLA+ Proof System [3]. The formal proof follows reasoning that can be expressed concisely in a few sentences of English, but the details are tedious and it took about two weeks to write. Incidentally, the two "phantom" entry bugs each violate this English reasoning and a specification containing one of these bugs would not pass the formal proof. This shows that the proof is stronger than strictly necessary. However, weakening the proof to account for this seems like it would add considerable detail to an already tedious proof.

The Pasture node specification runs 19 pages and the formal proof 68 pages [9]. Memoir also used TLA+ and the TLA+ Proof System and their specification runs 40 pages and formal proof 350 pages [4].

In conclusion, model checking with inserted bugs provides additional confidence that the specification is correct, because it gives some assurance that if there were a bug in the specification, it would likely have been found within configurations that are feasible to check. In some cases, an inserted "bug" turns out not to be a real bug and investigating this can lead to improved understanding of the specification.

# References


[1] Trusted Platform Module V1.2 Specification. http://www.trustedcomputinggroup.org.

[2] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, Jan 2008.

[3] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, pages 142–148, 2010.

[4] J. R. Douceur, J. R. Lorch, B. Parno, J. Mickens, and J. M. McCune. Memoir—formal specs and correctness proofs. Technical Report MSR-TR-2011-19, Feb 2011.

[5] R. Kotla, T. Rodeheffer, I. Roy, P. Steudi, and B. Wester. Secure offline data access using commodity trusted hardware. In submission.

[6] L. Lamport. The TLA toolbox. http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html.

[7] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[8] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy*, 2011.

[9] T. Rodeheffer and R. Kotla. Pasture: Specification, model checking, and correctness proof. In preparation.

# Automated Generation of Refinement Mappings

*Paul-David Brodmann, Hannes Lau, Uwe Nestmann*
*Technische Universitt Berlin*

The Inverse Implementation method is a new approach to verifying the conformity of Java software to specifications written in TLA+. It aims at automated verification, yet it still requires the programmer to provide detailed refinement mappings between his implementation and the specification. This talk introduces a new Java source code annotation language that aims to alleviate this problem, by allowing programmers to directly express the relation of the fields in their implementation to corresponding variables in a TLA+ specification. The refinement mapping generator converts the source code annotations into TLA+ refinement mappings which can then be used to verify the conformance of the program. As a result, the developer can verify his implementation without specific knowledge of the TLA+ Language or the concept of refinement mappings.

Our talk will start with a brief introduction to the Inverse Implementation method and the prototype model of the Java Virtual Machine that it is based on. The model can automatically be combined with the code of a concrete Java program to gain a TLA+ model of the execution of that program. This step can be seen as an inversion of the implementation, as the byte code model of the program is ́translatedback into the language the program was originally specified in. As the resulting execution model is produced in TLA+, the task of verifying the conformity of the program to its specification is reduced to the known problem of finding and proving a refinement relation between two formulas.

Once such a refinement mapping has been found, its correctness can be verified by manual proof techniques, by using an automated model checker, or by a combination of both. Using the model checker TLC reduces the necessary manual work to finding a refinement mapping and formalizing it in TLA+. Using the automated generator to create the refinement mappings, reduces it further to providing the annotations within the source (see example).

The annotation language allows for several kind of mappings, which differ in their expressiveness and complexity:

**direct mappings** directly relate the value of a field within the implementation to a variable of the specification.

**modified direct mappings** can apply basic algebraic operations to the implementation variable before it is mapped.

**combined mappings** extend direct mappings by combining the values of multiple fields with an algebraic operation and map the result to a single specification variable.

**delayed mappings** allow to bypass an unobservable initialization of a field within the implementation. Until the initialization of the program is complete, the variable that is mapped to the specification is set to a custom defined value.

HourClock example in Java with embedded mapping

```
@ConformsTo(module = 'HourClock', specification = 'HC')
class HourClock {

    @ModifiedDirectMapping(mapTo = 'hr', modification = '+1')
    private int currentHour;


    public void tick() {
        currentHour = (currentHour+1) % 12;
    }

    public static void main(String[] args) {
        HourClock myHourClock = new HourClock();
        for(;;) { myHourClock.tick(); }
    }
}
```

**proxy mappings** introduce shadow variables within the mapping to hide changes of the implementation fields until a given condition is true. This can be used to hide unobservable state changes within the implementation until they become observable.

**functional mappings** execute a specific implementation function to get the value that is then mapped to the specification variable.

Additionally to field mappings, entire TLA+ formulas can directly be embedded into the Java source code. Thus giving a developer with a larger TLA+ knowledge an easy way to test certain constraints within his implementation. The tool will generate the specification containing the formulas and map the variables and field names within the formula to the source code. The formula can then be tested with the TLC model checker.

During the presentation we will give a brief introduction to the concept of refinement mappings. We discuss how our tool derives the different kind of refinement mappings from the the Java source code annotations and demonstrate its use in the context of software verification. We will further present the possibility to write entire specifications within the source code by using annotations that contain TLA+ formulas.

This submission is a snapshot of the current work on Brodmann's bachelor thesis and the results of Lau's diploma thesis. It demonstrates work in progress. Therefore parts of the described functionalities are still in development.

# Teaching Transition Systems and Formal Specifications with TLA+

*Philippe* MAURAN *Philippe* QUÉINNEC *Xavier* THIRIOUX
*Université de Toulouse, IRIT*
*2 rue Camichel, BP 7122*
*F-31071 Toulouse Cedex 7, France*
*http://www.irit.fr/-Equipe-ACADIE-*

**Abstract**

We present our experience with teaching two courses using TLA+. The first course concerns transition systems, and TLA+ is used to describe, reason about, and analyze transition systems. The second course is about formal specifications, refinement, and simulation, and TLA+ is used to check refinements.

## 1 Context

The courses are taught in two different "engineer" degrees (equivalent to graduate or master degrees) at INPT/ENSEEIHT, one in computer science and applied mathematics (80 students), the other one in computer science and networking (15 students). In both cases, students have a solid scientific background with rather good mathematics bases. However they have no previous exposition to formal specification and modeling.

All four members of the teaching unit belong to IRIT/ACADIE team which studies methods and tools to verify and certify distributed embedded critical systems. As such, we have a good knowledge both of distributed systems, and of formal techniques. Moreover, two of us are regular users of TLA+ for our research activities.

## 2 Why TLA+?

We had done previous attempts with other formalisms (Unity, Promela, B...) but none were truly satisfactory. The main reasons were that tools were either not available or too complex to master, or that the formalism was too specialized. Our research experience with TLA+ led us to believe that it could be a good formalism for teaching: light and rather simple, but rigorous, complete (w.r.t to temporal logic and transition systems), modular, and with automatic tools. Neither courses are pure TLA+ courses, TLA+ is mainly a tool to express and manipulate formal concepts. Hence, only the relevant parts of TLA+ for a given course are taught.

# 3 Courses Content

## 3.1 Transition Systems

This course[1] is concerned with the specification, modeling, and validation of systems, especially concurrent systems. State transition systems are used as the basis for modeling. Temporal logics (CTL, LTL) are used to specify safety, liveness, and fairness properties. Verification methods (model checking, axiomatic proofs) are used to validate models.

This course is taught in 15 lessons of 1h45: 10 lectures with exercises, and 5 assisted labs. During its presentation, this course mixes formal subjects (transition systems, temporal logics) and "applied" subjects, where TLA+ is used to express the formal concepts previously seen. The presentation of TLA+ approximately follows Lamport's book *Specifying Systems*, using different examples.

## 3.2 Formal Specifications

This course introduces the formal notion of refinement in open systems, by presenting labeled transition systems with (un)observable events. Simulation and bisimulations relations are defined. Modules and their execution semantics are presented, and refinement between modules is studied. A in-house small framework based on TLA+ is used in labs to verify refinement properties.

# 4 Analysis

## 4.1 Students' Difficulties

- Quantifiers are believed easier than set theory: $\forall x \in S : \exists y \in S' : p(x) = y$ is more used than $p(S) \subseteq S'$
- Non-determinism is difficult: even $x' \in S$ is like magic.
- Fairness is very difficult. This may be the most important point which is not understood by a significant part of the students.
- Axiomatic proofs are manageable with regard to invariants. However, liveness formal proofs are too complex.
- The difference between checked properties (i.e. invariants, leadsto) and module specification (i.e. actions) is not clear: students want the checked properties to behave like constraints on the module (especially the usual *TypeInvariant*).
- Absence of a type system, to catch minor errors (e.g. $\in$ instead of $\subseteq$).

## 4.2 Success!

- By first teaching transition systems and transition predicates, students easily grasp that an action is not an assignment.
- LTL is easy and rather intuitive, CTL is not.
- Basic TLA+ is simple enough to be quickly understood.
- Complex notions (e.g. fairness) are accessible.
- TLC. An *essential* tool for student to experiment.

[1] Slides, in French, are available here: `http://queinnec.perso.enseeiht.fr/Ens/ens-st.html`.

# Integrating Formal Methods into Computer Science Curricula at a University of Applied Science

*Paul Tavolato and Friedrich Vogt*

This paper discusses the topic of integrating formal methods in computer science curricula at universities of applied sciences on a general scale: Why is the teaching of formal methods within such curricula essential? And how could we do that?

The main points are: Formal methods bring essential improvements to the daily practice of software development. Alumni from universities of applied sciences are the most important messengers in this field. Motivating students to realize the importance of formal methods for real world problems is therefore an important educational challenge in general.

Our approach to this challenge comprises three different courses in a master program: Theoretical Foundations of Computer Science, Software Engineering and Model Checking. It includes ideas with regard to contents as well as concepts for solving purely didactic questions. We are currently on the way to implement these concepts within a master program in Information Security at the University of Applied Sciences in St. Plten, Austria where Model Checking is fully grounded on the TLA+ Language and the recent version of the Toolbox.

The course Theoretical Foundations of Computer Science focuses on modeling and tries to convince students that technology as well as science is based on the building of some kind of model and that you need some kind of language for describing these models. Especially in computer science these languages are required to be formal. One of the examples for such a language taught in this course is logic. This course lays the ground for using formal methods.

The course Software Engineering gives a rather standard view of the field, using UML notation for the analysis and design of systems. We also give a short introduction to OCL to show how more formal definitions of requirements and of system behavior could be integrated into a design.

The course on Model Checking focuses on concrete models built with the TLA+ language. Within this, it pins the overall idea of modeling from the introduction of the Foundation course to small application domains. Therefore within this part the aim is focused to teach/learn the importance of the ability to create models by using proper abstractions. Consequently a general survey including a first look and feel of the features offered by the TLA+ Toolbox in applying very simple examples is given. The content of each of the following block lectures consists of two parts. The first part consists of some introduction to logic (in addition to the Foundations course!) with emphasis of natural deduction for propositional and predicate logic, temporal logic and some discussion of the issue and role of liveness properties in a specification. The second part emphasizes just the TLA+ Toolbox by explaining a wide range of different examples.

With each example it is stressed by application that a much better way to support the development process is to start with a model, with which one is able to check the

important properties, before the system is built and to change the model in case of failure and/or changes of system requirements at the model level if necessary. However current feedback shows that the aim of pointing out the practical relevance and usefulness of this paradigm is not fully understood and transparent to the students. Insofar I assume that this part has to be further strengthened by problems coming from and being applied by industry together with a clear view who are the current appliers of TLA+ in practice.

The overall conclusion is, that the Toolbox is definitely essential in teaching and learning (in addition the Hyper-Book proved to be very useful to foster self learning in addition!). How to bridge the gap between the worlds of small examples to the real world application is one of the issues which we do not have a satisfactory answer up till now. In addition we certainly have to focus on a more tight integration of the different problem solving strategies within all three courses. This could e.g. be tackled by using the same or similar examples throughout all courses. In addition we are currently planning to apply examples specifically targeted to security related issues since the overall frame of all of our courses are within the master course on Information Security.

We are right in middle of the implementation process of our course designs. We gained some experience within the last years and we are now reworking the concept a bit and will put the reworked/redesigned concept in execution in the next term this fall.