JTLV*: A Framework for Developing Verification Algorithms**

Amir Pnueli, Yaniv Sa'ar[1], and Lenore D. Zuck[2]

[1] Weizmann Institute of Science yaniv.saar@weizmann.ac.il
[2] University of Illinois at Chicago lenore@cs.uic.edu

1 Introduction

JTLV is a computer-aided verification scripting environment offering state-of-the-art Integrated Developer Environment for algorithmic verification applications. JTLV may be viewed as a new, and much enhanced TLV [18], with Java (rather than TLV-basic) as the scripting language. JTLV attaches its internal parsers as an Eclipse editor, and facilitates a rich, common, and abstract verification developer environment that is implemented as an Eclipse plugin.

JTLV allows for easy access to various (low-level) BDD packages with a (high-level) Java programming environment, without the need to alter, or even access, the implementation of the underlying BDD packages. It allows for the manipulation and on-the-fly creation of BDD structures originating from various BDD packages, whether existing ones or user-defined ones. In fact, the developer can instantiate several BDD managers, and alternate between them during run-time (of a single application) so to gain their combined benefits.

Through the high-level API the developer can load into the Java code several (SMV-like) *modules* representing programs and specification files, and directly access their components. The developer can also procedurally construct such modules and specifications, which enables loading various data structures (e.g., statecharts, LSCs, and automata) and compile them into modules.

JTLV offers users the advantages of the numerous tools developed by Java's ecosystem (e.g., debuggers, performance tools, etc.). Moreover, JTLV developers are able to introduce software methodologies such as multi-threading, object oriented design for verification algorithms, and reuse of implementations.

2 JTLV: Architecture

JTLV, described in Fig. 1, is composed of three main components, the API, the Eclipse Interface, and the Core.

* JTLV homepage: http://jtlv.sourceforge.net

** This material was based on work supported by the National Science Foundation, while Lenore Zuck was working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this article are those of the author and do not necessarily reflect the views of the National Science Foundation.

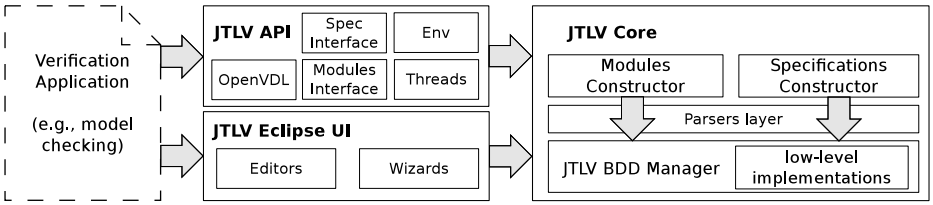

Fig. 1. JTLV Architecture.

API. In the following we present a set of *sample* functionalities. An exhaustive list of the API is in [20]

- TLV-like [18] ability to procedurally create and manipulate BDD's on-the-fly, a useful feature when dealing with abstractions and refinements ([1]);
- seamlessly alternate BDD packages at run-time (due to the factory design pattern [22]);
- save (and load) BDDs to (and from) the file system;
- load modules written in *NuSMV*-like language enriched with *Cadence SMV*-like parsing of loops and arrays of processes;
- procedurally access module's fields as well as its BDD's;
- perform basic functionalities on a module, e.g., compute successors or predecessors, feasible states, shortest paths from one state to another, etc.;
- procedurally create new modules and add, remove, and manipulate fields ;
- load temporal logic specification files;
- procedurally create and access the specification objects.

JTLV supports *thread*s, that are Java native threads coupled with dedicated BDD memory managers. Each thread can execute freely, without dependencies or synchronization with other threads. To allow for BDD-communication among threads, JTLV provides a low-level procedure that copies BDDs from one BDD manager into another. Our experience has shown that for applications that accommodate compositionality, execution using threads outperforms its sequential counterparts.

Assisted by the API, the user can implement numerous types of verification algorithms, some mentioned in the next section. It also contains the *OpenVDL* (Open Verification Developer Library), which is a collection of known implementations enabling their reuse.

Eclipse User Interface. Porting the necessary infrastructure into Java enables plugging JTLV into Eclipse, which in turn facilitates rich new editors to module and specification languages. See Appedix B for snapshots. A new JTLV project automatically plugs-in all libraries. JTLV project introduces new builders that take advantage of the underlying parsers, and connects them to these designated new editors.

Core. The core component encapsulates the BDD implementation and parses the modules and specifications. Through the JAVA-BDD ([22]) factory design pattern, a developer can use a variety of BDD packages (e.g., CUDD [21], BUDDY [14], and CAL [19]),

or design a new one. This also allows for the development of an application regardless of the BDD package used. In fact, the developer can alternate BDD packages during run-time of a single application. The encapsulation of the memory management system allows JTLV to easily instantiate numerous BDD managers so to gain the combined benefits of several BDD packages simultaneously. This is enabled by APIs that allow for translations among BDD's generated by different packages, so that one can apply the functionality of a BDD-package on a BDD generated by another package.

3 Conclusion, Related, and Future Work

We introduced JTLV, a scripting *environment* for developing algorithmic verification applications. JTLV is not a dedicated model checker (e.g. [2,13,10]) – its goal is to provide for a convenient development environment, and thus cannot be compared to particular model checkers. Yet, our implementation of invariance checking at times outperforms similar computations in such model checkers (see Table 1). Appendices A and B include screen shots obtained from some simple examples of work with JTLV.

Check Invariant	Muxsem 56	Bakery 7	Szymanski 6
JTLV	11	39.9	34.4
TLV	21.4	36.2	19
NuSMV	18.1	37.8	19.4
Cadence SMV	24.6	53.6	36.7

Table 1. Performance results (in sec.) of JTLV, compared to other model checkers.

We are happy to report that JTLV already has a small, and avid, user community, including researchers from Imperial College London [15], New York University [5,4,6], Bell Labs Alcatel-Lucent [5,4,6], Weizmann Institute [8,9], Microsoft Research Cambridge, RWTH-Aachen, California Institute of Technology [24,23], GRASP Laboratory University of Pennsylvania [7], and University of California Los Angeles. In these works JTLV is applied to: Streett and Rabin Games; Synthesis of GR(k) specifications; Compositional multi-theaded model checking; Compositional LTL model checking; Automata representation of LSCs and Statecharts; Synthesis of LSCs and of hybrid controllers.

The JTLV library (see [20]) already includes numerous model checking applications, including LTL and CTL* model checking [12], fair-simulation [11], a synthesis algorithm [17], Streett and Rabin games [16], compositional model checking ([3]), and compositional multi threaded model checking [6]. The API can also facilitate the reduction of other models into the verification framework (see, e.g., [8] where LSCs are reduced to automata).

We are currently developing a new thread-safe BDD package to allow concurrent access from multiple clients. Integrating a thread-safe BDD package into JTLV will entail a new methodology, which will streamline the development of multi-threaded symbolic

algorithms. This calls for an in-depth overview of many symbolic applications. We are also in the process of developing new interfaces to non-BDD managers (e.g. SAT and QBF solvers).

References

1. I. Balaban, Y. Fang, A. Pnueli, and L. D. Zuck. IIV: An Invisible Invariant Verifier. In *CAV'05*, pages 408–412, 2005.
2. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *CAV'99*, pages 495–499. Springer-Verlag, 1999.
3. A. Cohen and K. S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *CAV'08*, volume 5123 of *LNCS*, pages 149–161. Springer, 2008.
4. A. Cohen, K. S. Namjoshi, and Y. Sa'ar. A dash of fairness for compositional reasoning. Submitted for consideration to CAV'10, 2010.
5. A. Cohen, K. S. Namjoshi, and Y. Sa'ar. Split: A compositional LTL verifier. Submitted for consideration to CAV'10, 2010.
6. A. Cohen, K. S. Namjoshi, Y. Sa'ar, L. D. Zuck, and K. I. Kisyova. Parallelizing a symbolic compositional model-checking algorithm. in preparation, 2010.
7. H. K. Gazit, N. Ayanian, G. Pappas, and V. Kumar. Recycling controllers. In *IEEE Conference on Automation Science and Engineering*, Washington, August 2008.
8. D. Harel, S. Maoz, and I. Segall. Using automata representations of LSCs for smart play-out and synthesis. in preparation, 2010.
9. D. Harel and I. Segall. Synthesis from live sequence chart specifications. in preparation, 2010.
10. G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.
11. Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. In *CAV'03*, pages 381–392, 2003.
12. Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. LTL model checking with strong fairness. *Formal Methods in System Design*, 2002.
13. Cadence Berkeley Lab. Cadence SMV. http://www-cad.eecs.berkeley.edu/kenmcmil/smv, 1998.
14. J. L. Nielson. Buddy. http://buddy.sourceforge.net.
15. N. Piterman. Suggested projects. http://www.doc.ic.ac.uk/∼npiterma/projects.html, 2009.
16. N. Piterman and A. Pnueli. Faster solutions of rabin and streett games. In *LICS*, pages 275–284, 2006.
17. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.
18. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *CAV'96*, pages 184–195, 1996.
19. R. K. Ranjan, J.V. Sanghavi, R. K. Brayton, and A. S. Vincentelli. High performance BDD package based on exploiting memory hierarchy. In *DAC'96*, pages 635–640, June 1996.
20. Y. Sa'ar. *JTLV – web API*. http://jtlv.sf.net/resources/javaDoc/API1.3.2/.
21. F. Somenzi. CUDD: CU Decision Diagram package. http://vlsi.colorado.edu/ fabio/CUDD/, 1998.
22. J. Whaley. JavaBDD. http://javabdd.sourceforge.net.
23. T. Wongpiromsarn, U. Topcu, and R. M. Murray. Automatic synthesis of robust embedded control software. submitted to AAAI'10, 2010.
24. T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon control for temporal logic specifications. HSCC'10, 2010.

A Simple Examples

A.1 Invariant Property

Fig. 2 presents a simple JTLV application (somewhat cramped, to fit paper format). The preamble declares that this application uses CUDD ([21]). The first part loads `muxsem4.smv` (a simple mutual exclusion with semaphores protocol) and retrieves a Module `m`, which is a pure Java object representing `main`. The second part loads a "specification file" `muxsem4.spc` (the mutual exclusion property of the SMV code) and stores the first `Spec` object in `spc`. The third part retrieves the BDD representing the initial states of `m` and computes the reachable states with a simple fixpoint calculation. The fourth part performs the final test `m` cannot reach a state that violates `spc`. (JTLV have an API method that can be used to perform invariance testing.)

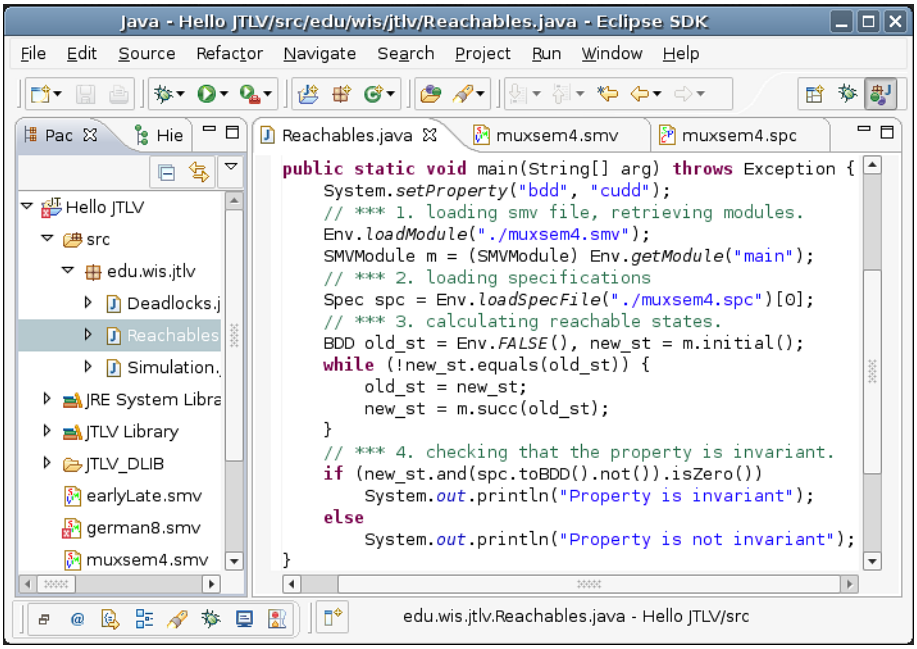

Fig. 2. JTLV reachable application.

A.2 Simulation Relation

Fig. 3 presents a simple JTLV application that takes two modules `As` and `Cs`, and the correlation between their states expressed by a BDD (`cor`). The code first refines the correlation to express the closure of the set of states that are "winning" for `As`. This is accomplished by computing the maximal fixpoint of the set of states (abstract and concrete) where every concrete step implies the existence of a matching abstract step. The latter is computed by the procedure `step`, where, formally:

$$[[\texttt{step}\,(\texttt{As},\texttt{Cs},\varphi)]] = \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \forall \boldsymbol{x}',\ (s,\boldsymbol{x}') \models \rho_{\texttt{Cs}} \rightarrow \exists \boldsymbol{y}' \text{ s.t. } (s,\boldsymbol{x}',\boldsymbol{y}') \models \rho_{\texttt{As}} \\ \text{and } (\boldsymbol{x}',\boldsymbol{y}') \in [[\varphi]] \end{array} \right\}$$

The second part checks that for every initial concrete state there is a matching abstract state. (As a matter of fact, the code can be replaced by a single call to an API method that tests for simulation.)

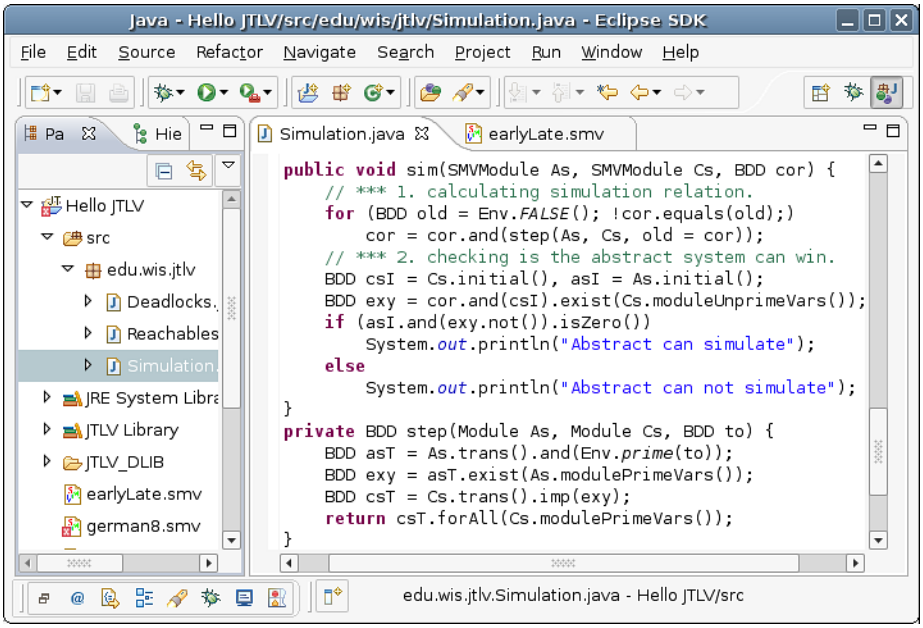

Fig. 3. JTLV simulation application.

B Eclipse Editors

In the following Fig. 4 we present the SMV and specifications editors. Note the error indicators in both editors.

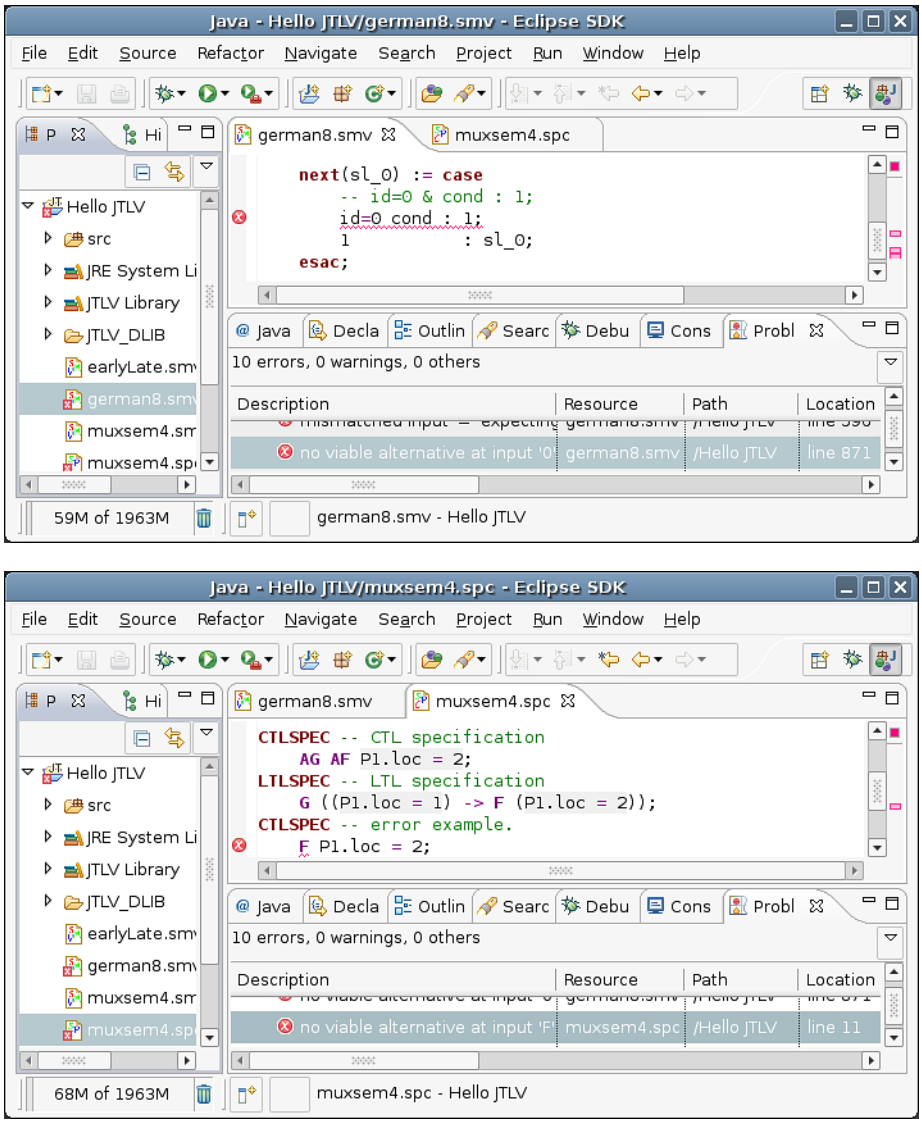

Fig. 4. SMV and specification editors, both indicating errors.