Algorithmic Methods for Formal Verification

Ph.D. Final Report by: Yaniv Sa'ar

Supervisors: Prof. Amir Pnueli, Prof. Lenore D. Zuck, and Prof. David Harel

DEPARTMENT OF COMPUTER SCIENCE AND APPLIED MATHEMATICS, WEIZMANN INSTITUTE OF SCIENCE

October 03, 2010

1 Introduction

Over the past several decades, both industry and academic communities developed a variety of tools and methodologies to address verification, namely whether a given program meets its specifications. Most verification tools are based on two common elements. On one hand the computational model which represents the system implementation, and on the other hand the specification language which represents the expectation from the implementation.

The computational model provides a general uniform representation for the various programming languages and diverse constructs suggested for synchronization and communication between the concurrent processes comprising a reactive system. The computational model assigns a semantics to each reactive system. This semantics associates with each program a behavior, which is a single or a set of computation structures that represent its possible executions. In our case, the semantics of a program is the set of its computations, where each computation is a sequence of states that can be generated in a single execution of the program.

The specification language, is essentially temporal logic augmented by some program specific predicates and functions, referring to the additional programming constructs needed to fully describe a state in the computation of a reactive program. In order to relate a specification presented by a formula in the logic to the program it is supposed to specify, it is necessary that the computational structures defined to be the semantics of a program can serve as models (in the logical sense) for the formula, which means that we can evaluate the formula on each of these structures and find whether it holds (is true) on the structure. Then, we say

This final report is based on papers: [PSZ10b], [PPS06], [BPPS10], [MS11], [CNS$^+$10c], [CNS10a], [CNS10b], [PS08]. In the following, we give a bird's eye summary of these papers. For an indepth details, the reader is referred to the relevant paper.

that the program satisfies (or implements) the specification given by the formula φ, if φ holds over each of the computation structures.

For example, in the linear semantics, the specification language is linear temporal logic, whose models are arbitrary sequence of states, where each state has a labeling among a finite set of propositions on which the logic is interpreted. Since the semantics of a program is a set of computations, which are also sequence of states, the specification φ is valid over the program P, if it holds over all the computations of P.

Under the choice of linear semantics of a program P, concurrent activity of two parallel processes in the program is represented by the interleaving of their atomic actions (transitions). To compensate for this simple representation of concurrency by interleaving, we add the notion of *fairness* to the computational model. As suggested by Lamport [Lam77], these should come in two flavors: *weak fairness* (to which we refer as *justice*) and *strong fairness* (to which we refer as *compassion*). Given a justice requirement J, a computation σ is just if it contains infinitely many occurrences of states that satisfy J. Given a compassion requirement $\langle p, q \rangle$, a computation σ is compassionate if either σ contains only finitely many states that satisfy p, or σ contains infinitely many occurrences of states that satisfy q.

An important observation is that justice is a special case of compassion. This is because the justice requirement J can also be expressed as the degenerate compassion requirement $\langle 1, J \rangle$, where we write 1 to denote the assertion *True* which holds at every state. In view of this observation, one may raise the natural question of the necessity of keeping these two separate notions of fairness. Several answers can be given to this question, one of which is that the treatment of com-

passion requirements is usually considered to be more complex than the treatment of justice requirements. In our first hypothesis for this report, we claim that this is not always true, namely, in some cases, the treatment of compassion requirements is conceptually not more complex than the treatment of justice requirements.

To support our claim, we present a new deductive rule for verifying response properties under the assumption of compassion requirements ([PS08, PSZ10a]). We resolve an open problem of previous rules. All previous approaches to verification of liveness under strong fairness requirements called for a recursive application of the rule, that led to cumbersome and highly unnatural style of proofs.

We present a much improved version of a proof rule that contains no recursion in its application. As such, it significantly streamline the methodology of deductive verification of temporal properties. We prove that the rule is sound, and present a constructive completeness proof for the case of finite-state systems. For the general case, we present a relative completeness proof. We report about the implementation of the rule in PVS and illustrate its application on some simple but non-trivial examples.

We shell return to this point later when discussing compositional algorithms. There we present a compositional algorithm for verification of parameterized systems that can also handle compassion requirements.

Using the common elements discussed above, many verification tools and methodologies have been presented over the years, yet not much effort was made to introduce a dedicated developer environment. An exception is TLV (*Temporal Logic Verifier*) introduced by [PS96a, PS96b]. TLV is a computer-aided verification environment, constructed as an additional layer, superimposed on top of CMU SMV ([McM93]). TLV presented its own scripting language (TLV-BASIC) to de-

velop formal algorithms, using a set of tools provided by its libraries. As a result of the development of TLV-BASIC, an important property that TLV holds, is an on-the-fly interactive prompt. TLV slightly changed the usual design perspective. Still TLV's objective was to design a platform for combining deductive with algorithmic verification. TLV-BASIC is procedural (as opposed to object oriented), and is not very advanced with respect to programming environments available today.

The goal of bringing the verification algorithm developer closer to the front-end of software engineering, suggests that she will be able to take the advantage of advanced software techniques, such as programming reuse and abstraction, multi-threaded applications, etc. Our main hypothesis for the second part of our work is that adopting advanced programming techniques, implies that the developer is now able to leverage her skills, as will become evident in the followings.

Our work starts by introducing a new computer-aided verification framework, that provides a state-of-the-art, Integrated Developer Environment (IDE) for algorithmic verification applications ([PSZ10b]). The framework, called JTLV (*Java Temporal Logic enVironment*), is an eclipse plugin. JTLV aims to facilitate a rich, common, and abstract Java API for the verification developer. The API is given in a high-level programming language; Java. The underlying space complexity derived from using BDD, is implemented in a low-level language; C, and is invisible to the user. And thus, the user is given a high-level programming environment, without having to worry about low-level space complexity.

We then continue to discuss one of the most ambitious and challenging problems in computer science; the automatic synthesis of programs and (digital) designs from logical specifications. A solution to this problem would lift programming from the current level, which is mostly imperative, to a declarative, logical

style. There is some evidence that this level is preferable, in particular when concurrency plays an important role.

We present a novel type of game called *General Reactivity* of rank 1 (GR(1)). We show symbolic algorithms to solve its game, to build a winning strategy and several ways to optimize the winning strategy and to extract a system from it ([PPS06, BPPS10]). We demonstrate its solution with a JTLV-based application.

Based on the GR(1) fragment of LTL, we then continue to present *AspectLTL* ([MS11]), a temporal-logic based language for the specification and implementation of crosscutting concerns. AspectLTL enables the modular declarative specification of expressive concerns, covering the addition of new behaviors, as well as the specification of safety and liveness properties. The language is supported by a JTLV-based prototype tool, that literally lifts declarative programming into practice.

We continue our quest to leverage the verification developer skills, by tackling the framework of compositional reasoning. Standard model checking algorithms prove safety properties through a reachability computation, computing an inductive assertion (the reachable states) that is defined over the full state vector. They often suffer from the state explosion problem [CG87]; for concurrent programs, this is manifested as an exponential growth of the state space with increasing number of components.

A promising approach to ameliorate the state explosion problem, is to decompose the verification task so that the reasoning is as localized as possible. The local reasoning algorithm is a mechanization of the classical Owicki-Gries compositional method [OG76]. The foundation is a construction of a *vector* of local (i.e., per-process) inductive invariants, $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$. The invariants are

mutually interference-free. Such a vector is called a *split-invariant*, as the conjunction of its components, $(\bigwedge_i \theta_i)$, is always a *globally inductive* invariant. Locality is enforced by syntactically limiting each process assertion to the variables visible to that process.

Both global inductiveness, and syntactical limitation strongly suggest that JTLV would be superior to further investigate such a technique. We start by presenting a multi-threaded application of the compositional technique ([CNS+10c]). We then continue to develop a new compositional algorithm that can handle fairness requirements locally ([CNS10a]). [CNS10a] also supports our prior claim, namely, that in some cases, the treatment of compassion requirements is conceptually not more complex than the treatment of justice requirements. Finally, we present SPLIT ([CNS10b]), that is to the best of our knowledge, the first tool to implement a fully automated compositional method for both safety and liveness properties.

The rest of the report is organized as follows. In Section 2 we present the first part of our work, and discuss fairness in general. We then continue to present in Subsection 2.2 a new deductive rule to prove response. In Section 3 we start the main part of our work by presenting JTLV. Section 4 discuss the synthesis problem, and in Subsection 4.1 we briefly describe the fragment of GR(1) and its solution. In Subsection 4.2 the new aspectual LTL-based language called AspectLTL is presented. Section 5 discuss the compositional framework. Subsection 5.1 presents a new multi-threaded model checking algorithm, Subsection 5.2 presents a new compositional algorithm that can handle strong fairness, and Subsection 5.3 presents a new tool that implements the compositional framework. Finally, in Section 6 we conclude the report.

2 Handling Compassion

An important component of the formal model of reactive systems is a set of *fairness requirements*. As suggested by Lamport [Lam77], these should come in two flavors: *weak fairness* (to which we refer as *justice requirements*) and *strong fairness* (to which we refer as *compassion*). Originally, these two distinct notions of fairness were formulated in terms of enableness and the activation of transitions within a computation, as follows:

- The requirement that transition τ is *just* implies that if τ is continuously enabled from a certain position on, then it is taken (activated) infinitely many times.

  An equivalent formulation is that every computation should contain infinitely many positions at which τ is disabled or has just been taken.

- The requirement that transition τ is *compassionate* implies that if τ is enabled infinitely many times in a computation σ, then it is taken infinitely many times.

Justice requirements are used in order to guarantee that, in a parallel composition of processes, no process is neglected forever from a certain point on. Compassion, which is a more stringent requirement, is often associated with coordination statements such as semaphore *request* y (equivalently *lock* y) operations or message passing instructions. It implies fair arbitration in the allocation of an atomic resource among several competing processes.

In a more abstract setting, a justice requirement is associated with an *assertion* (first-order state formula) J, while a compassion requirement is associated with a pair of assertions $\langle p, q \rangle$. With these identifications, the requirements are:

- A computation σ is just with respect to the requirement J, if σ contains infinitely many occurrences of states that satisfy J.
- A computation σ is compassionate with respect to the requirement $\langle p, q \rangle$, if either σ contains only finitely many p-positions or σ contains infinitely many q-positions.

To see that these definitions are indeed generalizations of the transition-oriented definition, we observe that the requirement that transition τ be just can be expressed by the abstract justice requirement $J_\tau = (\neg En(\tau) \vee \mathit{Taken}(\tau))$, while the requirement that transition τ be compassionate can be expressed by the abstract compassion requirement $C_\tau = \langle En(\tau), \mathit{Taken}(\tau) \rangle$. In these assertions, $En(\tau)$ is true at all states on which τ is enabled. Similarly, $\mathit{Taken}(\tau)$ is true at all states that can result by taking transition τ.

As discussed in Section 1, justice is a special case of compassion, which leads to the natural question of the necessity of keeping these two separate notions of fairness. Several answers can be given to this question. On the modeling level, the argument is that these two notions represent different phenomena. Justice represents the natural independence of parallel processes in a multi-processing system. Compassion is typically used to provide an abstract representation of queues and priorities which are installed by the operating system in order to guarantee fairness in coordination services provided to parallel processes.

Another answer to this question is a different cost associated with the implementation of these two notions. In a multi-processor system, justice comes for free and is a result of the independent progress of parallel processes. In a multi-programming system, where concurrency is simulated by scheduling, justice can be implemented by any scheduling scheme that gives each process a fair chance to

progress, such as round-robin scheduling. Compassion, in both types of systems, is usually implemented by maintenance of queues and use of priorities.

There is also a proof-theoretic answer to this question which is based on the fact that, up to now, all the proposed deductive rules for proving properties under the assumption of compassion were significantly more complex than the rule under the assumption of justice alone. Our main claim here is this need not necessarily be the case, and there exist deductive rules for verification in which the price of compassion is comparable to that of justice.

2.1 The Legacy Recursive Rule

In the way of a background, we present rule F-WELL which is derived from the proof rule presented in [MP91] and is representative of the different prices traditionally associated with the distinct notions of fairness. It is modified in order to represent the transition from the computational model of *fair transition systems* (used in [MP91]) to that of *fair discrete systems* (FDS) which we use here. The rule is presented in Fig. 1.

The FDS $(\mathcal{D}\backslash\{\langle p_i, q_i\rangle\})$ is obtained by removing from $\mathcal{D}$ the compassion requirement $\langle p_i, q_i\rangle$. Thus, $(\mathcal{D}\backslash\{\langle p_i, q_i\rangle\})$ has one compassion requirement less than $\mathcal{D}$.

The rule considers a system (FDS) which has both justice requirements ($\mathcal{J}$) and compassion requirements ($\mathcal{C}$). It establishes for this system the temporal property $p \Longrightarrow \diamondsuit\, q$ claiming that every p-state is followed by a q-state. The rule relies on "helpful" fairness requirements $F_1, \ldots, F_n$ which may be either justice or compassion requirements. Premise W3 imposes different conditions on each fairness requirement F_i according to whether F_i is a compassion or a justice requirement.

Rule F-WELL

For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions $p, q, \varphi_1, \ldots, \varphi_n$,
fairness requirements $F_1, \ldots, F_n \in \mathcal{J} \cup \mathcal{C}$,
and ranking functions $\Delta_1, \ldots, \Delta_n$ where each $\Delta_i : \Sigma \mapsto W$

W1. $p \implies q \vee \bigvee_{j=1}^{n} \varphi_j$

For each $i = 1, \ldots, n$,

W2. $\varphi_i \wedge \rho \implies q' \vee (\varphi_i' \wedge \Delta_i = \Delta_i') \vee \bigvee_{j=1}^{n} (\varphi_j' \wedge \Delta_i \succ \Delta_j')$

W3. If $F_i = \langle p_i, q_i \rangle \in \mathcal{C}$ then

C3. $\varphi_i \implies \neg q_i$

C4. $(\mathcal{D} \backslash \{\langle p_i, q_i \rangle\}) \models (\varphi_i \implies \diamondsuit(p_i \vee \neg\varphi_i))$

Otherwise ($F_i = J_i \in \mathcal{J}$),

J3. $\varphi_i \implies \neg J_i$

$\mathcal{D} \models (p \implies \diamondsuit q)$

Figure 1: Legacy (recursive) rule F-WELL

Consider first the special case in which all the helpful requirements are justice requirements. In this case, we only invoke premise J3 as an instance of W3. For such a case, the rule provides a real reduction by establishing a temporal property, based on premises which are all first-order.

On the other hand, if some of the helpful requirements are compassionate, then some of the premises will include instances of C3 and C4. In this case, some of the premises are temporal and have a syntactic form similar to that of the conclusion. In such a case, one may ask whether this is not a circular rule in which the premises are not necessarily simpler than the conclusion. As observed above, the rule is not really circular because the premise C4 requires the establishment of a similar temporal property but over a system with fewer compassion requirements. So

while the methodology is still sound, it appears cumbersome and its application often requires explicit induction on the number of compassion requirements in the analyzed system.

This explanation serves to illustrate that the application of this rule is significantly more complex and cumbersome in the case that we have compassion requirements, and the situation is much simpler if all the fairness requirements are of the justice type. We refer to this phenomenon by saying that the application of this rule is *recursive* in the presence of compassion requirements.

2.2 All You Need is Compassion [PS08, PSZ10a]

The main result here is based on a new deductive rule for response properties which does not need any recursion in order to handle compassion requirements. The rule, called FAIR-RESPONSE, is presented in Fig. 2.

Rule FAIR-RESPONSE

For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions $p, q, \varphi_1, \ldots, \varphi_n$,
compassion requirements $\langle p_1, q_1 \rangle, \ldots, \langle p_n, q_n \rangle \in \mathcal{C}$,
and ranking functions $\Delta_1 \ , \ldots \ , \Delta_n$ where each $\Delta_i : \Sigma \mapsto W$

R1. $p \implies q \ \vee \ \bigvee_{j=1}^{n} (p_j \wedge \varphi_j)$

For each $i = 1, \ldots, n$,

R2. $p_i \wedge \varphi_i \wedge \rho \implies q' \ \vee \ \bigvee_{j=1}^{n} (p'_j \wedge \varphi'_j)$

R3. $\varphi_i \wedge \rho \implies q' \ \vee \ (\varphi'_i \wedge \Delta_i = \Delta'_i) \ \vee \ \bigvee_{j=1}^{n} (p'_j \wedge \varphi'_j \wedge \Delta_i \succ \Delta'_j)$

R4. $\varphi_i \implies \neg q_i$

$p \implies \diamondsuit q$

Figure 2: Deductive rule FAIR-RESPONSE

For simplicity, we presented the rule for the case that the system contains only compassion requirements but no justice requirements. This is not a serious restriction since any original justice requirement $J \in \mathcal{J}_\mathcal{D}$ can be represented by an equivalent compassion requirement $\langle 1, J \rangle$. Similarly to the previous version of this rule, the rule relies on a set of premises guaranteeing that a computation which contains a p-state that is not followed by a q-state leads to an infinite chain of descending ranks. Since the ranks range over a well-founded domain $\mathcal{A} : (W, \succ)$, this leads to a contradiction.

In view of the simple form of the rule, it appears that, in many cases, the study and analysis of fair discrete systems can concentrate on the treatment of compassion requirements, and deal with justice requirements as a special case of a compassion requirement. This does not imply that we suggest giving up the class of justice requirements altogether. For modeling and implementation of reactive systems, we should keep these two classes of fairness requirements distinct. However, the main message of this paper is that, when verifying temporal properties of FDS's, the treatment of compassion requirements is conceptually not more complex than the treatment of justice requirements. Computationally, though, justice is simpler in the same way that checking emptiness of generalized Büchi automata is simpler than checking emptiness of Streett automata.

The new rule has been implemented in the theorem prover PVS [OSRSC01]. In fact, it has been added as an additional rule (and associated strategy) within the PVS-*based temporal prover* TLPVS [PA04]. In order to do so, we had to prove the soundness of the FAIR-RESPONSE rule within PVS. We continue to present a constructive completeness proof for the case of finite-state systems, and a relative completeness proof for the general case. We illustrate its application on some

simple but non-trivial examples.

3 JTLV: A Framework for Developing Verification Algorithms [PSZ10b]

Most existing verification tools are designed to serve as verifiers where, to implement all but the simplest algorithm, a developer is assumed to be intimately familiar with the internal structure and implementation details of the system. For reasons of efficiency, verification systems are commonly implemented in a low-level C code.

We present JTLV (*Java Temporal Logic enVironment*), a tool that provides an abstract framework for developing verification applications in a high-level programming environment. JTLV allows the developer to focus on the verification goals at hand without sacrificing performance or dealing with low-level details of the verification tool.

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 [PS96a], with Java rather than TLV-basic as the scripting language. JTLV attaches its internal parsers as Eclipse editors, 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,

[†] JTLV homepage: http://jtlv.ysaar.net

whether existing ones (e.g., CUDD [Som98], BUDDY [Nie], and CAL [SRBV96]) 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.

We are happy to report that JTLV already has a small, and avid, user community, including researchers from Imperial College London [Pit09], New York University [CNS10b, CNS10a, CNS$^+$10c], Bell Labs Alcatel-Lucent [CNS10b, CNS10a, CNS$^+$10c], Weizmann Institute [HMS10, HS10], Microsoft Research Cambridge, RWTH-Aachen, California Institute of Technology [WTM10b, WTM10a], GRASP Laboratory University of Pennsylvania [GAPK08], 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; Verifying heap properties ([BPSZ10]); Automata representation of LSCs and Statecharts; Synthesis of LSCs and of hybrid controllers.

The JTLV library (see [Sa']) includes numerous model checking applications,

including LTL and CTL* model checking [KPRS02], fair-simulation [KPP03], a synthesis algorithm [PPS06], Streett and Rabin games [PP06], compositional model checking ([CN08]), and compositional multi threaded model checking [CNS+10c]. The API can also facilitate the reduction of other models into the verification framework (see, e.g., [HMS10] where LSCs are reduced to automata).

4 Synthesis

We address the problem of automatically synthesizing digital designs from linear-time specifications. We consider various classes of specifications that can be synthesized with effort cubic in the number of states of the reactive system, where we measure effort in symbolic steps.

The synthesis algorithm is based on a novel type of game called General Reactivity of rank 1 (GR(1)), with a winning condition of the form

$$(\Box \diamondsuit p_1 \wedge \cdots \wedge \Box \diamondsuit p_m) \rightarrow (\Box \diamondsuit q_1 \wedge \cdots \wedge \Box \diamondsuit q_n),$$

where each p_i and q_i is a Boolean combination of atomic propositions. We show symbolic algorithms to solve this game, to build a winning strategy and several ways to optimize the winning strategy and to extract a system from it. We also show how to use GR(1) games to solve the synthesis of LTL specifications in many interesting cases.

We continue to present *AspectLTL* ([MS11]), a new declarative programming language that is based on the GR(1) fragment of LTL. AspectLTL is a temporal-logic based language for the specification and implementation of crosscutting concerns, that lifts declarative programming into practice. It enables the modular declarative specification of expressive concerns, covering the addition of new be-

haviors, as well as the specification of safety and liveness properties. Given an AspectLTL specification, consisting of a base system and a set of aspects, we provide AspectLTL with a composition and synthesis-based weaving process, whose output is a correct-by-construction executable artifact. The language is supported by a JTLV-based prototype tool and is demonstrated using a running example.

4.1 Synthesis of Reactive(1) Designs [PPS06, BPPS10]

The synthesis problem was first identified by Church [Chu63]. Several methods have been proposed for its solution [BL69, Rab72]. The two prevalent approaches to solving the synthesis problem were by reducing it to the emptiness problem of tree automata, and viewing it as the solution of a two-person game. In these preliminary studies of the problem, the logical specification that the synthesized system should satisfy was given as an S1S formula and the complexity of synthesis is non-elementary.

The problem was considered again in [PR89] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL). This followed two previous attempts [CE81, MW84] to synthesize programs from temporal specification, which reduced the synthesis problem to satisfiability, ignoring the fact that the environment should be treated as an adversary. The method proposed in [PR89] for a given LTL specification φ starts by constructing a Büchi automaton $\mathcal{B}_\varphi$, which is then determinized into a deterministic Rabin automaton. This double translation necessarily causes a doubly exponential time complexity [Ros92].

The high complexity established in [PR89, Ros92] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners

from ever attempting to use it for any sizeable system development. Yet there exist several interesting cases where the synthesis problem can be solved in polynomial time, by using simpler automata or partial fragments of LTL [WHT03, AT04, HRS05, JGB05]. Representative cases are the work in [AMPS98] which presents an efficient quadratic solution (N^2) to games (and hence synthesis problems) where the acceptance condition is one of the LTL formulas $\Box p$, $\Diamond q$, $\Box \Diamond p$, or $\Diamond \Box q$. A more recent paper is [AT04] which presents efficient synthesis approaches for the LTL fragment consisting of a Boolean combinations of formulas of the form $\Box p$.

Our work can be viewed as a generalization of the results of [AMPS98] and [AT04] into the wider class of *Generalized Reactivity*(1) formulas (GR(1)), i.e., formulas of the form

$$(\Box \Diamond p_1 \wedge \cdots \wedge \Box \Diamond p_m) \quad \rightarrow \quad (\Box \Diamond q_1 \wedge \cdots \wedge \Box \Diamond q_n). \tag{1}$$

Here, we assume that the specification is an implication between a set of *assumptions* and a set of *guarantees*. Following the results of [KPP05], we show how any synthesis problem whose specification is a GR(1) formula can be solved with effort $O(N^3)$, where N is the size of the state space of the design and effort is measured in symbolic steps, i.e., in the number of preimage computations [BGS06]. Furthermore, we present a symbolic algorithm for extracting a design (program) which implements the specification.

We show that GR(1) formulas can be used to represent a relatively wide set of specifications. First, we show that we can include past LTL formulas in both the assumptions and the guarantees. Second, we show that each of the assumptions and guarantees can be a deterministic Just Discrete System (Büchi automaton). Thus, our method does not suffer from the exponential blow-ups incurred in LTL

synthesis for the translation of the formula to an automaton and for the determinization of the automaton because the user provides the specification as a set of deterministic automata. (But note that the state space of the system is the product of the sizes of the automata, which may cause an exponential blowup). Furthermore, a symbolic implementation of our algorithm is easily obtained when the automata are represented in symbolic form. One drawback is that our formalism is less expressive than LTL. In particular, Reactivity (Streett) conditions can not be expressed.

The reader may suspect that GR(1) specifications place an undue burden on the user or that its expressive power is too limited. We argue that this is not the case. Intuitively, many specifications can naturally be split into assumptions on the environment and guarantees on the system. (Cf. [Pnu85].) Often, assumptions and guarantees can naturally be written as conjunctions of simple properties that are easily expressed as deterministic automata.

As an evident to our argument, [BGJ$^+$07] have shown two case studies of small but realistic industrial modules. Their first case study concerns a generalized buffer from IBM, a tutorial design for which a good specification is available. The second concerns the arbiter for one of the AMBA buses [Ltd99], a characteristic industrial design that is not too big.

We stresses the compositionality of synthesis from LTL specifications and the structure of specifications as a guide to efficient synthesis. At the same time, we emphasizes the symbolic analysis of state space through the usage of BDDs.

Our work presents the algorithms and techniques used to synthesize systems. We show how to solve Generalized Reactive(1) games symbolically, compute a winning strategy, and extract a correct program, if it exists. We present the JTLV

implementation, and finally show how the techniques developed can be used to synthesize systems from temporal specifications.

4.2 AspectLTL: An Aspect Language for LTL Specifications [MS11]

A common characteristic of languages of the aspect-oriented programming paradigm [KLM+97] and of related advanced modularity paradigms, is that their structural building blocks specify separate, yet possibly inter-dependent crosscutting concerns. This poses a major challenge, which is the automated composition or weaving of these separate specifications or program pieces into a single correct implementation, one which can be programmatically executed and indeed supports and follows the different concerns.

To address this challenge, a balance needs to be made between the extent of modularity of the program specification or code to go beyond traditional abstraction boundaries, the language's expressive power in specifying and manipulating system behavior, and the ability to automatically transform or weave such a modular specification or code into an executable correct artifact. The more modular or structurally and syntactically separate yet semantically inter-dependent and expressive the language constructs are, the more difficult it becomes to automatically generate a correct implementation.

Our work presents *AspectLTL*, a language for the *specification and implementation* of crosscutting concerns, based on linear temporal logic (LTL) [Pnu77]. The aspects of AspectLTL, called LTL aspects, enable the declarative specification of expressive crosscutting concerns. These include the specification of safety properties, which may be used to prevent a base system from visiting 'bad states', the

specification of liveness (fairness) properties, which may be used to force a base system to visit 'good states' (infinitely often), and the addition of new behaviors to a base system, which is done by specifying the existence of new transitions and new states as necessary. To use the categorization by Katz [Kat06], LTL aspects can specify spectative, regulative, and invasive aspects.

Moreover, we provide AspectLTL with a synthesis-based JTLV weaving application, whose output is a correct-by-construction executable artifact. Following a composition of the specified aspects with a base system, using symbolic disjunctive and conjunctive operations, we formulate the problem of correct weaving as a synthesis problem [PR89], essentially a game between the environment and the (augmented) base system. An algorithm based on [PPS06] is used to solve the game, that is, to provide the augmented system with a winning strategy, if any.

If a winning strategy is found, it is presented as a deterministic, executable automaton, which represents an augmented base system whose behavior is guaranteed to adhere to the specified aspects, in all possible environments. If a winning strategy is not found, we know that it does not exist, that is, that no system exists which is based on the base system and can adhere to the specified LTL aspects in all environments. Thus, LTL aspect composition and synthesis is sound and complete.

Our work can be viewed as demonstrating how correct aspect weaving can be reduced to (a variant of) the classical synthesis problem. An AspectLTL specification is made of a base system, given as a finite-state machine specified in an SMV [SMV] format, and a set of LTL aspects, each of which is specified in a similar SMV-like format, containing a symbolic representation of the aspect's added behaviors (transitions) and a related LTL specification. As the base system as-

sumes nothing about the LTL aspects it may be woven with, AspectLTL supports *obliviousness*. Moreover, the use of the symbolic representation provides *quantification:* rather than relating to concrete states, a single formula typically relates to a set of states. These two language features, obliviousness and quantification, are considered a distinguishing characteristic of aspect languages [FF05], and so, indeed, AspectLTL is an aspect language.

An aspect language has a *join point model* (JPM), which defines the points where an aspect may interfere with a base, how these points may be specified, and how the additional aspect behavior is defined. AspectLTL features a very general and permissive JPM: it allows new transitions to be added at any state of the base system: all states are possible *join points* and *pointcuts* are not specified explicitly. The "advice" of LTL aspects has not only local and specific effect on selected points along the execution, but also a global, temporal and general effect on ongoing, infinite executions.

Some previous works have formally characterized aspects using LTL or automata, mainly in order to prove aspect correctness using model-checking techniques (see, e.g., [GK07, KFG04]). Other works translate LTL properties into corresponding monitors written using aspect code (e.g., in AspectJ), as a means for LTL runtime verification (see, e.g., [SB05]). In contrast to the above two lines of work, our main goal is to use an LTL characterization of aspects as an input for a composition and synthesis process, in order to produce a correct-by-construction executable system. In other words, we use LTL not only as a specification language but also as a programming language, leading directly into an executable artifact.

Moreover, it is important to distinguish AspectLTL synthesis from other forms

of composition and program synthesis that have attracted research attention in recent years. Correct composition of features (see, e.g., [KA08, TBKC07]), for example, is typically discussed at the level of safe type checking, and not at the level of the actual semantics of the features involved: features can be composed if the resulting program is type-safe and compiles, not if its semantics (in terms of input/output or event sequences) indeed complies with each of the features' specifications.

AspectLTL is supported by a prototype Eclipse plug-in, which we have developed on top of JTLV. We used this prototype to define several AspectLTL specifications, to weave them, and to run the generated executable artifacts. To demonstrate our ideas, we presented a running example. The example is initially built from an underlying base system, which models a service for students exams. We start off with a base system, and use it as a minimal basis on which to add aspects.

5 Compositional Methods

The verification of concurrent programs remains an elusive goal, in spite of numerous advances in model checking methods. The main difficulty is state explosion – the verification question is PSPACE-hard in the number of components. In practice, this means that the size of the state space is often exponential in the number of processes. Symbolic BDD-based approaches have ameliorated this difficulty to some extent.

A promising approach to further ameliorate state explosion is to decompose a verification task so that the reasoning is as localized as possible. The local reasoning algorithm is a mechanization of the classical Owicki-Gries compositional

method [OG76]. The setting is that of *asynchronous, shared-memory* protocols. The algorithm constructs a "local proof", which is a collection of per-process invariants, $\{\theta_i\}$, whose conjunction (i.e., $\theta_1 \wedge \theta_2 \ldots \wedge \theta_N$) is guaranteed to be an inductive whole-program invariant. This vector of local assertions is called a *split-invariant*, as the program invariant is in this conjunctive form. The computation of the strongest split invariant is a simultaneous fixpoint computation over the vector $(\theta_1, \theta_2, \ldots, \theta_N)$.

However, the strongest split invariant may be weaker than the set of reachable states, and therefore not strong enough to prove a safety property. [CN07] solved this problem by formulating a complete verification procedure which strengthens the split invariant by discovering and adding auxiliary shared variables to track local predicates. [CN08], used the split invariance as the basis for a new compositional algorithm for checking LTL properties. Experiments reported in these papers show that assertional local reasoning can be significantly faster than monolithic (i.e., non-compositional) model checking.

5.1 Parallelizing A Symbolic Compositional Model-Checking Algorithm [CNS$^+$10c]

We describe a parallel, symbolic, model-checking algorithm, built around a compositional reasoning method. The compositional method, called "local reasoning", builds a collection of per-process (i.e., local) invariants, which together imply a desired global safety property. The local proof computation is a simultaneous fixpoint evaluation, which lends itself to parallelization. Moreover, the locality of the computation makes it possible to partition work across several threads, each with its own BDD manager, while limiting the amount of cross-thread copy-

ing. Experimental results are encouraging, and show that the parallelized computation can achieve substantial speed-up, without incurring significant memory overhead.

In this work, we explore the parallelization of a symbolic compositional reasoning algorithm for checking safety properties. Prior approaches to parallelization partition the BDD representation of the reachability frontier, or the image computation itself, but nonetheless compute the exact set of reachable states.

Our method has a different starting point: we parallelize a local reasoning (i.e., compositional) approach to the verification of safety properties. Local reasoning consists of computing per-process (hence, "local") invariants, using limited information about the local state of other processes. Compositional reasoning has inherent advantages for parallelization: the analysis can be carried out separately for each process, and the information that must be exchanged between processes is limited, by the nature of the analysis. Previous works ([CN07, CN08]), have shown that local reasoning often out-performs the standard reachability-based method of verifying safety properties. We show that it also lends itself to parallelization. To the best of our knowledge, this is the first approach to parallel model checking based on automatic compositional analysis.

The computation of the strongest split invariant over the vector $(\theta_1, \theta_2, \ldots, \theta_N)$, is naturally parallelizable. In the simplest setting, each thread of a multi-thread system is responsible for computing one component of the fixpoint. Communication from thread i to another thread j is limited (by the analysis) to communicating the effect that the transitions of processes i have on the *shared* program state.

While it is easy to see how to parallelize the fixpoint computation, an actual implementation with BDDs is not straightforward. The BDD data structure is natu-

rally "entangled". Standard BDD libraries are not thread-safe. We show, however, that one can exploit the locality of the reasoning, and use multiple non-thread-safe BDD managers, one per thread. Synchronization is needed only during the phase of the algorithm where shared effects are communicated between threads.

The algorithm has been implemented using JTLV. The experimental results are encouraging. On several (parameterized) protocols, the parallel algorithm demonstrates speedup ranging from 3.5 to nearly 6.4 on a system with 8 cores. The memory overhead due to multiple BDD managers is small, usually about 5%, with an upper limit of 30%.

The extension of the local reasoning computation to liveness properties given in [CN08] is also easily parallelizable. In a nutshell, the liveness algorithm first computes the strongest split invariant, followed by an independent analysis of each component process. The second step is trivially parallelized.

To summarize, the parallelization of the local reasoning algorithm results in significant speedup, without incurring substantial memory overhead. As local reasoning is itself often more efficient than a global reachability computation, parallelization offers a multiplicative improvement over a sequential reachability computation. While our implementation and experiments are with finite-state protocols, the algorithmic ideas are more general, and apply also to non-finite domains, such as those used in static program analysis.

5.2 A Dash of Fairness for Compositional Reasoning [CNS10a]

Proofs of progress properties often require fairness assumptions. Directly incorporating global fairness assumptions in a compositional method is difficult, given the local flavor of such reasoning. We present a fully automated local reasoning

algorithm which handles fairness assumptions. Experiments demonstrate that the new algorithm shows significant improvement over standard model checking.

This work is the continuation of a line of research on mechanizing assertional (i.e., state-predicate based) compositional verification. The starting point is the computation of the strongest *split invariant*. Our work develops a new algorithm for compositional model checking with fairness assumptions, which tackles this problem with a successive refinement method. It also presents a new compositional proof rule for verification under fairness. Moreover, the model checking algorithm can be instrumented to generate a valid instantiation of the proof rule upon success.

Fairness assumptions are often needed for proofs of progress properties. It has long been understood how to incorporate fairness in standard model checking [CES86, EL87], but doing so is a challenge for compositional methods. The difficulty is that fairness assumptions commonly refer to local states from a number of processes. For example, a common (strong) fairness constraint is that "*for every process: if the process is enabled infinitely often, it is infinitely often executed*". As "enabledness" depends on local states, this assumption refers to the local states of every process. Since compositional reasoning is based on a per-process view, the presence of such global assumptions can be problematic.

As previously mentioned, the strongest split invariant may be weaker than the set of reachable states, and therefore not strong enough to prove a safety property. The local liveness method of [CN08] does not directly apply to fairness constraints. This is because the method is sound only for properties expressed over shared variables. Incorporating fairness into the specification, through the identity $M \models \mathsf{A}_{Fair}(Spec) \equiv M \models \mathsf{A}(Fair \Rightarrow Spec)$, results in a new

specification which names a number of local variables (due to $Fair$). One can, of course, turn all the local variables in $Fair$ into shared variables, but this defeats the purpose of local reasoning.

The new algorithm gets around this difficulty by a process of iterative refinement. The fairness constraint is replaced with a weaker form, which depends monotonically on the current split invariant, and is expressed over only the shared variables. This allows using the compositional algorithm from [CN08], with slight modifications. If verification succeeds with the weaker fairness assumption, the property is proved. If not, a bogus counter-example is produced, and analyzed to discover new local predicates which are then exposed as auxiliary shared variables. Exposing local state strengthens the split invariant in the next round of computation, which strengthens the abstracted fairness assumption by monotonicity. This is repeated until a decisive result (either success or a real counter-example) is obtained. The iterative process terminates—and is thus complete—for finite-state programs: eventually, enough of the local state is exposed to either prove a correct property or to disprove an incorrect one. Moreover, it is possible to disprove a property *without* building up the entire state space.

The algorithm, being predicate-based, has a simple implementation using JTLV. We carry out an evaluation with several parameterized protocols, where each instance of the protocol is finite-state. The experimental results show promise: the compositional verification is faster in almost all cases, sometimes by one or two orders of magnitude. Exposing a limited amount of local state suffices for both proofs and disproofs of properties, validating the basic premise behind compositional reasoning.

5.3 SPLIT: A Compositional LTL Verifier [CNS10b]

We present SPLIT, a new tool for the verification of shared-variable, asynchronous concurrent programs. SPLIT ameliorates state explosion through assertional (i.e., state-based) compositional reasoning, based on the classical Owicki-Gries method [OG76], as describe above.

SPLIT implements the techniques of [Nam07, CN07, CN08, CNS10a] for verifying safety and general LTL properties. The foundation is a computation of compact local invariants, one for each process, which are used for constructing a proof for the property. An automatic refinement procedure gradually exposes more local information, until a decisive result (proof/disproof) is obtained.

SPLIT implements a number of algorithms; together, they result in a fully automatic compositional model checker for general LTL properties.

1. A simultaneous least fixpoint algorithm [Nam07], which computes the strongest split invariant vector (A split invariant is usually weaker than the set of reachable states.)

2. A safety refinement method [CN07], which achieves completeness by gradually "exposing" local predicates (i.e., encoding them as shared variables)

3. A compositional algorithm which verifies arbitrary LTL properties [CN08], based on a split invariance computation and a counter-example based refinement scheme

4. A recently developed compositional algorithm [CNS10a], for the verification of progress properties under general fairness assumptions

Experimental results support the hypothesis that local reasoning allows verifying significantly larger systems without running into state explosion, and can result in order-of-magnitude improvements in run-time over monolithic model checking. It is interesting that basic local reasoning suffices for the proofs for many protocols, without a need for refinement. In many other cases, a proof or disproof is obtained by exposing a limited amount of local state, validating the basic argument for compositional verification. SPLIT has been used to verify protocols for cache coherence and mutual exclusion. To the best of our knowledge, this is the first tool to implement a fully automated compositional method for both safety and liveness properties.

6 Conclusion

We start our work by raising two hypothesises, both fundamental in the field of verification. The first concerns the treatment of fairness requirements, specifically whether the treatment of compassion requirements is harder than the treatment of justice requirements. Our second concerns the acute gap between verification developer skills and the lack of an adequate environment to support these skills.

As to the treatment of fairness requirements, we present two lines of work, one introducing a new deductive rule for verifying response properties where the compassion is treated without recursion ([PS08, PSZ10a]), and one that introduces a new algorithm that locally incorporates both justice and compassion requirements in compositional framework ([CNS10a]).

Computationally, justice is simpler in the same way that checking emptiness of generalized Büchi automata is simpler than checking emptiness of Streett automata. We show that when verifying temporal properties of FDS, the treatment

of compassion requirements is conceptually not more complex than the treatment of justice requirements.

To supply a developing environment for verification, we developed JTLV ([PSZ10b]), a new framework for developing verification algorithms. We demonstrate the power of JTLV, by applying it to the problem of synthesis, and to the problem of compositional reasoning: For synthesis we give a solution to the GR(1) fragment of LTL ([PPS06, BPPS10]), as well as present AspectLTL – a new temporal-logic based programming language ([MS11]). For compositional reasoning, we present a new compositional algorithm that exploits the benefit of a multi-core system ([CNS$^+$10c]), a new compositional algorithm that can handle fairness requirements locally ([CNS10a]), and a fully automated implementation of our compositional framework ([CNS10b]).

We conclude that the high complexity established for LTL synthesis, does not necessarily identify it as intractable, and that we can use the GR(1) fragment present a declarative style programming language. On the other hand we also conclude that compositional reasoning is a promising approach to ameliorate the state explosion problem. Thus we use JTLV for a variety of applications, ranging from the treatment of fairness properties, to synthesis, to compositional verification, and conclude that adopting an advanced environment leverage the developer's skills.

References

[AMPS98] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.

[AT04] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, pages 1–25, 2004.

[BGJ+07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, pages 3–16, 2007.

[BGS06] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. *Formal Methods in System Design*, pages 37–56, 2006.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, pages 295–311, 1969.

[BPPS10] Roderick Bloem, Barbara Jobstmann Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences, Special Issue in Honor of Amir Pnueli*, 2010. Under revision.

[BPSZ10] Ittai Balaban, Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. Verification of multi-linked heaps. *Journal of Computer and System Sciences, Special Issue in Honor of Amir Pnueli*, 2010. Under revision.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In

Proc. IBM Workshop on Logics of Programs, pages 52–71. Springer-Verlag, 1981.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1986.

[CG87] Edmund M. Clarke and Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *PODC*, pages 294–303, 1987.

[Chu63] Alonzo Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963.

[CN07] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *Proc. 19th Int. Conf. on Computer Aided Verification*, pages 55–67. Springer-Verlag, 2007.

[CN08] Ariel Cohen and Kedar S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *Proc. 20th Int. Conf. on Computer Aided Verification*, pages 149–161. Springer-Verlag, 2008.

[CNS10a] Ariel Cohen, Kedar S. Namjoshi, and Yaniv Sa'ar. A dash of fairness for compositional reasoning. In *Proc. 22nd Int. Conf. on Computer Aided Verification*, 2010.

[CNS10b] Ariel Cohen, Kedar S. Namjoshi, and Yaniv Sa'ar. SPLIT: A compositional LTL verifier. In *Proc. 22nd Int. Conf. on Computer Aided Verification*, 2010. web: `http://split.ysaar.net/`.

[CNS+10c] Ariel Cohen, Kedar S. Namjoshi, Yaniv Sa'ar, Lenore D. Zuck, and Katya I. Kisyova. Parallelizing a symbolic compositional model-checking algorithm. In *Proc. 6th Int. Haifa Verification Conf.*, 2010. to appear.

[EL87] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Sci. of Comp. Programming*, 1987.

[FF05] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[GAPK08] Hadas K. Gazit, Nora Ayanian, George J. Pappas, and Vijay Kumar. Recycling controllers. In *IEEE Conference on Automation Science and Engineering*, 2008.

[GK07] Max Goldman and Shmuel Katz. MAVEN: Modular aspect verification. In *TACAS*, pages 308–322. Springer-Verlag, 2007.

[HMS10] David Harel, Shar Maoz, and Itai Segall. Using automata representations of LSCs for smart play-out and synthesis. in preparation, 2010.

[HRS05] Aidan Harding, Mark Ryan, and Pierre-Yves Schobbens. A new algorithm for strategy synthesis in LTL games. In *Tools and Al-*

gorithms for the Construction and the Analysis of Systems, pages 477–492. Springer-Verlag, 2005.

[HS10] David Harel and Itai Segall. Synthesis from live sequence chart specifications. in preparation, 2010.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Proc. 17th Int. Conf. on Computer Aided Verification*, pages 226–238. Springer-Verlag, 2005.

[KA08] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *ASE*, pages 258–267. IEEE, 2008.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. In *T. Aspect-Oriented Software Development I*, pages 106–134, 2006.

[KFG04] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *SIGSOFT FSE*, pages 137–146. ACM, 2004.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP*, pages 220–242. Springer-Verlag, 1997.

[KPP03] Yonit Kesten, Nir Piterman, and Amir Pnueli. Bridging the gap between fair simulation and trace inclusion. In *Proc. 15th Int. Conf. on Computer Aided Verification*, pages 381–392, 2003.

[KPP05] Yonit Kesten, Nir Piterman, and Amir Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. and Comp.*, pages 36–61, 2005.

[KPRS02] Yonit Kesten, Amir Pnueli, Lion Raviv, and Elad Shahar. LTL model checking with strong fairness. *Formal Methods in System Design*, 2002.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *Trans. Soft. Eng.*, pages 125–143, 1977.

[Ltd99] ARM Ltd. AMBA specification (rev. 2). Available from www.arm.com, 1999.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[MP91] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, pages 97–130, 1991.

[MS11] Shahar Maoz and Yaniv Sa'ar. AspectLTL: An aspect language for LTL specifications. In *Proc. 10th Int. Conf. on Aspect-Oriented Software Development*, 2011. to appear.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, pages 68–93, 1984.

[Nam07] Kedar S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *Proc. 8th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, 2007.

[Nie] Jorn L. Nielsen. BuDDy. http://buddy.sourceforge.net.

[OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, pages 279–285, 1976.

[OSRSC01] Sam Owre, Natarajan Shankar, John M. Rushby, and Dave Stringer-Calvert. *PVS System Guide*. Menlo Park, CA, 2001.

[PA04] Amir Pnueli and Tamarah Arons. TLPVS: A PVS-Based LTL Verification System. In *Verification: Theory and Practice*, pages 598–625, 2004.

[Pit09] Nir Piterman. Suggested projects. http://www.doc.ic.ac.uk/∼npiterma/projects.html, 2009.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[Pnu85] Amir Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and Models of Concurrent Systems*, pages 123–144, 1985.

[PP06] Nir Piterman and Amir Pnueli. Faster solutions of rabin and streett games. In *Proc. 21st IEEE Symp. Logic in Comp. Sci.*, pages 275–284, 2006.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *Proc. 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer-Verlag, 2006.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, pages 652–671. Springer-Verlag, 1989.

[PS96a] Amir Pnueli and Elad Shahar. A platform for combining deductive with algorithmic verification. In *Proc. 8th Int. Conf. on Computer Aided Verification*, pages 184–195, 1996.

[PS96b] Amir Pnueli and Elad Shahar. The tlv system and its applications. Technical report, The Weizmann Institute, 1996.

[PS08] Amir Pnueli and Yaniv Sa'ar. All you need is compassion. In *Proc. 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 233–247, 2008.

[PSZ10a] Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. All you need is compassion. in preparation, 2010.

[PSZ10b] Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. JTLV: A framework for developing verification algorithms. In *Proc. 22nd Int. Conf. on Computer Aided Verification*, pages 171–174, 2010.

[Rab72] Michael O. Rabin. *Automata on Infinite Objects and Churc's Problem*, volume 13. Amer. Math. Soc., 1972.

[Ros92] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[Sa'] Yaniv Sa'ar. *JTLV – web API*. http://jtlv.ysaar.net/resources/javaDoc/API1.3.2/.

[SB05] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *5th Workshop on Runtime Verification*, pages 109–124. Elsevier, 2005.

[SMV] SMV model checker. http://www.cs.cmu.edu/~modelcheck/smv.html.

[Som98] Fabio Somenzi. CUDD: CU Decision Diagram package. http://vlsi.colorado.edu/ fabio/CUDD/, 1998.

[SRBV96] Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto S. Vincentelli. High performance BDD package based on exploiting memory hierarchy. In *Proc. 33rd Conf. on Design Automation*, pages 635–640, June 1996.

[TBKC07] Sahil Thaker, Don S. Batory, David Kitchin, and William R. Cook. Safe composition of product lines. In *GPCE*, pages 95–104. ACM, 2007.

[WHT03] Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Proc. of the Int. Conf. on the Implementation and Application of Automata*, pages 11–22. Springer-Verlag, 2003.

[WTM10a] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Automatic synthesis of robust embedded control software. submitted to AAAI'10, 2010.

[WTM10b] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications, 2010.