RESEARCH ARTICLE OPEN ACCESS

An Implementation on Effective Robot Mission under Critical Environemental Conditions by Using Temporal Logic Based Approaches

Chinnam Subbarao[1], Dr.M M Naidu[2]
[1]*Research Scholar, Dept. Of MCA, Acharya Nagarjuna University, Guntur(A.P), India.*
[2]*Professor And Director Of CSE,IT,MCA, RVR & JC College Of Engineering, Guntur(A.P), India*

ABSTRACT
Software engineering is a field of engineering, for designing and writing programs for computers or other electronic devices. A software engineer, or programmer, writes software (or changes existing software) and compiles software using methods that make it better quality. Is the application of engineering to the design, development, implementation, testingand main tenance of software in a systematic method. Now a days the robotics are also plays an important role in present automation concepts. But we have several challenges in that robots when they are operated in some critical environments. Motion planning and task planning are two fundamental problems in robotics that have been addressed from different perspectives. For resolve this there are Temporal logic based approaches that automatically generate controllers have been shown to be useful for mission level planning of motion, surveillance and navigation, among others. These approaches critically rely on the validity of the environment models used for synthesis. Yet simplifying assumptions are inevitable to reduce complexity and provide mission-level guarantees; no plan can guarantee results in a model of a world in which everything can go wrong. In this paper, we show how our approach, which reduces reliance on a single model by introducing a stack of models, can endow systems with incremental guarantees based on increasingly strengthened assumptions, supporting graceful degradation when the environment does not behave as expected, and progressive enhancement when it does.
Keywords: Robot Mission, Software Engineering, Temporal Logic Based Approaches, Automatic ntrollers.

I. INTRODUCTION

Software engineering is a field of engineering, for designing and writing programs for computers or other electronic devices. A software engineer, or programmer, writes software (or changes existing software) and compiles software using methods that make it better quality. Is the application of engineering to the design, development, implementation, testingand main tenance of software in a systematic method. Now a days the robotics are also plays an important role in present automation concepts. Motion planning and task planning are two fundamental problems in robotics that have been addressed from different perspectives. Bottom-up motion planning techniques concentrate on creating control inputs or closed loop controllers that steer a robot from one configuration to another [1], [2], while taking into account different dynamics and motion constraints. Controller synthesis and planning approaches based on temporal logic have proven useful for generating discrete event-based robot behaviors from high-level specifications (e.g. [4, 30, 29]). Such approaches rely on finite-state models that purport to represent the operating environment and how the robot can interact with it. However, any such model is by definition an abstraction of the real environment and its dynamics, and any such model entails a risk that it is not a true representation of the environment as encountered at runtime.

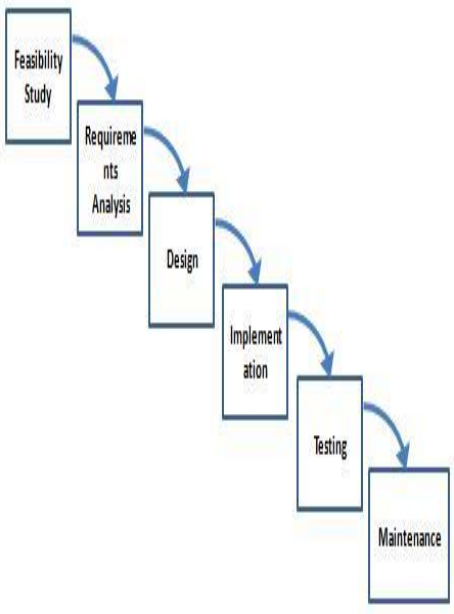

Fig: The Fundamental Steps in Software Engineering.

In some scenarios, this risk, when materialized, may lead to catastrophic failure of the mission. One means to cope with this uncertainty [12] is to use machine learning techniques that revise (or indeed generate from scratch) the models on which synthesis relies so that, over a period of time, the models converge upon a \realistic" description of the environment [27, 11, 14]. One drawback of using such techniques is the computational cost of learning, and the delay before the mission can begin in earnest, which may be prohibitive in some domains (e.g. safety-critical systems).Another drawback is that the learned model may be of such complexity that synthesis becomes computationally infeasible, and in the worst case nothing can be guaranteed in a world where anything can go wrong. There is therefore a benefit in having an element of manual abstraction involved in synthesizing robotic behaviors. To that end, we have proposed an approach [7] in which models at different levels of abstraction are used to synthesize a controller capable of gracefully degrading its guarantees when the runtime environment diverges from one of the more abstract models, and progressively enhancing its guarantees when the environment behaves as envisaged in the more idealized models.

Our approach uses a stack of models where higher models are more idealised and can be simulated by the lower models. A mission requirement is associated with each tier of the stack. Higher tiers allow to produce controllers guaranteeing stronger requirements, while lower tiers only allow for controllers with weaker requirements because of their more realistic description of the environment dynamics. Each tier of the stack can be regarded as an independent controller synthesis problem, but our approach combines the resulting controllers in such a way that a failure in a higher controller can be handled by a graceful degradation to the controller of a lower tier, resulting in a lower guaranteed `service level'. Likewise, if the environment conforms to a higher tier, we may attempt to synthesise a controller for a higher tier and so enhance the guaranteed service level.

In this paper, we show how synthesized controller stacks can be used to provide robust behavior for robot missions from high-level temporal logic specifications. We apply it to an existing case study involving a robot engaged in a surveillance mission and show how, in addition to automatic synthesis for cyclic missions (i.e. missions in which the goals are achieved infinitely many times, our approach enables the robot to handle invalid environment models. Our Paper mainly focus on to implement an novel method to resolve all these thing in an easy manner.

II. LITERATURE SURVEY

Software testing, Systematic testing is one of the most important and widely used techniques to check the quality of software. Testing, however, is often a manual and laborious process without effective automation, which makes it error-prone, time consuming, and very costly. Estimates are that testing consumes 30-50% of the total software development costs. The tendency is that the effort spent on testing is still increasing due to the continuing quest for better software quality, and the ever growing size and complexity of systems. For effective testing of software there are several approaches(Steps) are done previously. Such as

1). Labelled Transition Systems
2). Parallel Composition
3). Legal LTS
4). Simulation & LTS Control.

1). Labelled Transition Systems

Labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform. Definition 1. A labelled transition system is a 4-tuple hQ, L, T, q0i where – Q is a countable, non-empty set of states; – L is a countable set of labels; – $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation; – $q0 \in Q$ is the initial state.

2). Parallel Composition

(P || Q) expresses the parallel composition of the processes *P* and *Q*. It constructs an LTS which allows all the possible interleavings of the actions of the two processes. Actions, which occur in the alphabets of both *P* and *Q*, constrain the interleaving since these actions must be carried out by both of the processes at the same time. These shared actions synchronize the execution of the two processes. If the processes contain no shared actions then the composite state machine will describe all interleavings. In the following example, *x* is an action shared by the processes *A* and *B*.

A = (a -> x -> A).
B = (b -> x -> B).
||SYS = (A || B).

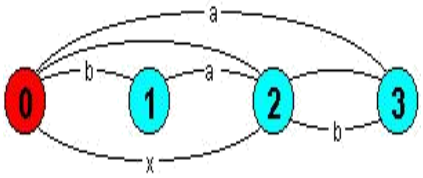

Fig: Composition Parallel.

The diagram depicts the LTS for the composite process *SYS*. It can be easily seen that, in this simple example, the two possible execution traces are < *a*,*b*,*x*> and <*b*,*a*,*x*>. That is the actions *a* and *b* can occur in any order. Composite process declarations are distinguished from primitive process declarations by prefixing with the symbol ||. Primitive processes may not contain the parallel composition operator and composite processes may not use action prefix, choice or recursion. This separation is partly to ensure that FSP can only generate finite systems. The parallel composition operator is n-ary. The following example, with three processes, is a system describing the behaviour of a garage shared between two cars.

CAR(I=1) = (car[I].outside -> car[I].enter -> car[I].ingarage -> car[I].exit -> CAR).
GARAGE(N=2) = (car[x:1..N].enter -> car[x].exit -> GARAGE).
||SHARE = (CAR(1) || CAR(2) || GARAGE).

Note that an action label may consist of more than one identifier (optionally indexed) joined by ".". Processes referred to in composite process definitions may be either primitive or composite. So, for example, *SHARE* can be constructed in two stages:

||CARS = (CAR(1) || CAR(2)).
||SHARE = (CARS || GARAGE).

3.(Legal LTS)

Is defined as ,Given LTSs M = (S_M;A,ΔM, SM_0) and E = (S_E; A; ΔE; sE0), where A is partitioned into actions controlled and monitored by M (A = A_C Ů A_M), we say that M is a legal LTS for E if for all (S_E; S_M) Є E//M.

4.Simulation

The simulation is defined as the relation between two LTSs is formally defined as follows,Let α be the universe of all LTSs with communicating alphabet A. Given E and F in α,we say that E simulates F, written E ≥F, when (E,F) is contained in some simulation relation R Belongs to α*α}.

Proposed Method

The central concept in our approach is that of the control stack, which has in each tier a controller synthesis problem for a particular mission requirement and environment model.

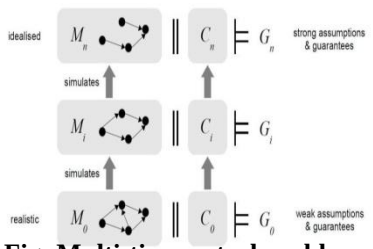

Fig: Multi-tier control problem

Overall the control stack specifies the robot's mission. The key requirements the approach imposes in order to guarantee graceful degradation and progressive enhancement are that (see below Figure): (i) higher-level environment models must be simulated by lower-level environment models, capturing a notion of idealisation of higher-level models; (ii) higher-level controllers used to achieve enhanced functionality must be simulated by lower levels controllers, ensuring a consistent overall strategy; (iii) the runtime infrastructure must be capable of detecting when an inconsistency between an environment model (in any tier) and the runtime environment occurs; (iv) a sound automated replanning procedure for each tier that is expressive enough to deal with the system requirements for its tier must be provided, allowing progressive system enhancement after inconsistencies have been detected. Our implementation of the approach provides the runtime infrastructure (iii) and planning procedure (iv), guarantees controller simulation (ii), and checks that the models given in a control stack specification satisfy (i).

The environment models are expected to be ranked in terms of the degree of idealisation of the environment they represent. The environment model M_0 is the least idealised and require that environment models further up the hierarchy allow strictly less behaviour. This can be formally captured via a simulation relation, M_i M_j for i < j. We require environment models to have the same communicating alphabets partitioned identically into con-trolled and monitored actions. Controlled actions are those that the robot may choose to perform, while monitored ac-tions are events that the robot observes in the environment. In summary, the less idealised the environment model is, the more behaviour (in terms of unexpected actions and non-determinism) may arise.Each tier i has an associated requirement (G_i) to be achieved by the system assuming that the runtime environment con-forms to the environment model for that tier (M_i).

Each tier introduces a control problem E_i = hM_i; G_ii. A solution to a control problem (a controller) is a deterministic LTS that, when composed with its environment, guarantees requirement G_i (i.e. M_ikC_i j= G_i). The control stack introduces an additional constraint: each controller must be simulated by controllers in lower tiers (C_i C_j for i j). Intuitively, this requires that a controller never do something that a lower-tier controller would not do, thus ensuring that if a controller must be stopped, because the assumptions for its tier are discovered not to hold, decisions made by it up to that point have been consistent with lower-tier controllers. This allows for graceful degradation, falling back to lower-tier controllers when needed. Section 4 describes how this constraint is satisfied.

Control stack synthesis is executed bottom-up through the tiers. The operation attempts to build a controller that solves the control problem in a tier while being simulated by the controller for the tier immediately below. We do not require that control problems for all tiers have solution. It is possible that the system starts in a degraded mode, with controllers solving problems up to level i. The system, as the current state evolves, may progressively enhance its behavior by synthesizing controllers for tiers beyond tier i.

After synthesis, the enactment procedure continuously monitors the environment and concurrently executes the stack of controllers giving priority to the controller of the upper-most enabled tier. It continuously updates the current state based on monitored actions and sensed state, disabling tiers at level i and above should an inconsistency be detected at tier i (Section 4 shows how this is achieved). At any point, to progressively enhance functionality, a re-planning attempt may be made for the lowest disabled tier. Based on the cur-rent state of the enabled tier immediately below, the state of the disabled tier is automatically approximated and an attempt is made to build a controller that will work despite the uncertainty about the current state of the tier. This demands that the controller synthesis procedure be capable of solving problems exhibiting non-determinism. Should a controller exist, it is put into the controller hierarchy and the tier is enabled. The approach does not prescribe when re planning must be attempted. In principle this can be done at any time, however in practice re planning may be associated with a clock or with heuristics related to the problem domain.

Implementation

The implementation of our framework consists of two main components: 1).planner , 2).Entactor

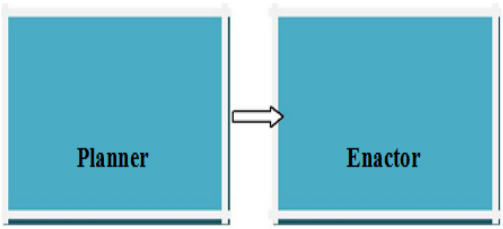

Fig: The Fundamental Steps in the way of Implementation.

1) **Planner:** a planner, which implements the controller synthesis algorithms The Modal Transition System Analyser (MTSA), is a tool for developing and analysing compositional models of concurrent systems, using the Finite State Processes (FSP) process algebra. Importantly for our approach, MTSA implements controller synthesis algorithms for Generalized Re-activity(1) (GR(1)) goals, which cover an expressive subset of linear temporal logic including safety and liveness proper-ties. Our general approach is agnostic as regards the synthesis procedure, but GR(1) is expressive enough for many domains. We extended MTSA to support the specification and synthesis of complete control stacks A control stack C is specified in MTSA as follows.

```
controlstack ||C@{Controlled} {
tier(ENV, REQ)
...
}
```

where Controlled refers to a set of controlled actions, and where each tier consists of environment model ENV and mission requirement specification REQ. A control stack may consist of any number of tiers ordered such that the last tier has the most realistic environment model. Environment models and requirements are defined using existing support in MTSA for process and property specification in FSP and FLTL (fluent linear temporal logic),Synthesis of the control stack is achieved by solving the controller synthesis problem of each tier bottom-up from the lowest tier. If no solution is found for the problem in a particular tier, synthesis of the stack terminates at that tier. The procedure also includes a sanity check that the environments of tiers simulate the one immediately above.

Synthesis for a single tier i consists of the following steps:

1. Compose the tier's environment model Ei in parallel with the controller Ci 1 generated by the tier below (if there is a tier below) to create E0 i. This ensures that the controller for tier i will be simulated by the controller of the tier below.
2. Solve the GR(1) controller synthesis problem for the tier's requirement on E0i, to produce controller Ci.
3. Complete controller Ci to produce $C"_i$. The completion consists of considering the monitored actions enabled in each state of the controller, and adding transitions to a designated exception state for any monitored actions which are not enabled. These transitions capture behaviors of the environment that have not been anticipated in the present tier's environment model. If the runtime environment does not behave as the model describes, one of these transitions will be taken to the exception state. A single extra transition, which we call an exception marker, is added at the exception state which indicates to the enactor that a particular tier has been disabled. It is these transitions that enable the

enactor to detect inconsistencies The final control stack state machine CS is a parallel composition of the completed controllers C"$_i$, i.e. CS = complete(C1)||…..||complete(Cn)=$_1'$||….|| $'$ .This composition guarantees the requirements of every tier of the stack until the exception marker for tier i occurs, at which point it only guarantees the requirements of the tiers up to i-1.

2. Enactor

An enactor, which handles run-time execution of the control stack. The enactor extends [3] to execute control stacks rather than individual controllers. It keeps track of the stack's current state, executing controlled actions (via domain-specification implementations as in [3]) and responding to monitored environment events. When the current state is controlled, the enactor selects an enabled action at random. When the state is uncontrolled or a mixed controlled/uncontrolled state, the enactor waits to receive an environment event. In states where the only enabled action is an exception marker for some tier i, the enactor notes the degradation of the service to i-1 and reports this to the rest of the framework. In e ect, this disables the controller for tier i.
The planner may attempt at any point an enhancement by re-synthesizing a controller for tier i (or above).

Experimental Results

Fig: Nao executing mission.

We have experimented with our synthesis and enactment infrastructure [3] in various robotic settings, including an AR Drone 2.0, a Katana robotic arm and a Nao H25 humanoid robot. A video of the latter executing a synthesized mission control stack similar to the one described above can be found at
XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXX XXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXX XXXXXXXX http://www.doc.ic.ac.uk/~das05/naoq3.avi.

Controller enactment for these settings requires implementing each of the controlled actions in the control stack specification in terms of the existing behaviors provided by the robot's API. For instance, for the control of the Nao robot, the detection of various types of reward is achieved by recognizing balls of different colors using the Nao's on-board camera. The balls are presented to the Nao upon arrival in each region. The location of the Nao within the environment (an o ce) is determined using trilateration with respect to a number of landmarks in positions known a priory (i.e. a structured environment). The landmarks them-selves are recognized using the on-board camera. Similarly, rewards and locations can be recognized on the AR Drone using its front and bottom cameras respectively. The synthesized mission control stack is executed by the enactor, which starts by assuming the runtime environment behaves like the model in the upper tier. Initially, in the video, we allow this assumption to hold by providing the Nao with the reward it is expecting. Later, we break the bound on the number of damage events expected in the uppermost tier, forcing the enactor to gracefully degrade the level of service. Execution continues seamlessly such that the Nao immediately seeks a repair, as demanded by the lower tier requirement. The experiments demonstrate that our general approach can be deployed in a robotics setting on top of a high-level API that encapsulates the complexities of, for instance, con-trol of the system dynamics that allows stable movement of the AR Drone or the localization of the Nao robot. The resulting system can then ensure that mission-level guarantees can be gracefully and automatically degraded (or enhanced) when necessary to cope with unexpected mission-level events in the environment.

IV.CONCLUSION & FUTURE SCOPE

In this paper we have presented an approach for robust high-level control synthesis for robot missions, and applied it in a various scenarios. In contrast to the `all or nothing' approach of other work based on temporal logic, our approach allows a mission specification to include a range of requirements of different `strengths' which entail different levels of risk when operating in the runtime environment. Our implementation ensures that when the stronger requirements of higher tiers

cannot be met due to environmental uncertainty, the level of service degrades gracefully to a level at which requirements can be guaranteed. It then permits progressive enhancement at a later stage. In future work we are interested in quantifying the level of risk associated with the tiers of our control stack, and combining the approach with techniques that can learn appropriate environment models for disabled tiers in the stack before progressive enhancement.

REFERENCES

[1]. R. Bloem, A. Cimatti, K. Greimel, G. Ho erek, R.
[2]. Konighofer,• M. Roveri, V. Schuppan, and R. Seeber.
[3]. Ratsy – a new requirements analysis tool with synthesis. In Proceedings of the 22Nd International Conference on Computer Aided Veri cation, CAV'10, pages 425{429, Berlin, Heidelberg, 2010. Springer-Verlag.
[4]. A. Bohy, V. Bruyere, E. Filiot, N. Jin, and J.-F. Raskin. Acacia+, a tool for ltl synthesis. In Proceedings of the 24th International Conference on Computer Aided Veri cation, CAV'12, pages 652{657, Berlin, Heidelberg, 2012. Springer-Verlag.
[5]. V. Braberman, N. D'Ippolito, N. Piterman, D. Sykes, and S. Uchitel. Controller synthesis: From modelling to enactment. In Proceedings of the 2013 International Conference on Software Engineering, pages 1347{1350. IEEE Press, 2013.
[6]. I. Cizelj and C. Belta. Control of noisy di erential-drive vehicles from time-bounded temporal logic speci cations. In Robotics and Automation (ICRA), 2013 IEEE International Conference on, pages 2021{2026, May 2013.
[7]. L. de Alfaro and T. A. Henzinger. Interface automata. In ESEC / SIGSOFT FSE, pages 109{120. ACM, 2001.
[8]. N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. ACM Tran. Softw. Eng. Methodol., 22, 2013.
[9]. N. D'Ippolito, V. A. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel. Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In ICSE, pages 688{699, 2014.
[10]. N. D'Ippolito, V. A. Braberman, N. Piterman, and Uchitel. Synthesis of live behaviour models for fallible domains. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, pages 211{220. ACM, 2011.
[11]. N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. Mtsa: The modal transition system analyser. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pages 475{476, Washington, DC, USA, 2008. IEEE Computer Society.
[12]. R. Ehlers. Symbolic bounded synthesis. In Proceedings of the 22Nd International Conference on Computer Aided Veri cation, CAV'10, pages 365{379, Berlin, Heidelberg, 2010. Springer-Verlag.
[13]. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In ICSE 2009, pages 111{121. IEEE, 2009.
[14]. N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In R. de Lemos, H. Giese, H. A. Muller,• and M. Shaw, editors, Software Engineering for Self-Adaptive Systems, volume 7475 of Lecture Notes in Computer Science, pages 214{238. Springer, 2010.
[15]. T. Fraichard and J. J. K. Jr. Guaranteeing motion safety for robots. Auton. Robots, 32(3):173{175, 2012.
[16]. C. Ghezzi, M. Pezze, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In ICSE, pages 277{287, 2014.
[17]. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-11, pages 257{266, New York, NY, USA, 2003. ACM.
[18]. R. C. Hill, J. E. R. Cury, M. H. de Queiroz, D. M. Tilbury, and S. Lafortune. Multi-level hierarchical
[19]. interface-based supervisory control. Automatica, 46(7):1152{1164, July 2010.
[20]. B. Jobstmann and R. Bloem. Optimizations for ltl synthesis. In Proceedings of the Formal Methods in Computer Aided Design, FMCAD '06, pages 117{124, Washington, DC, USA, 2006. IEEE Computer Society.
[21]. B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property

synthesis. In Proceedings of the 19th International Conference on Computer Aided Veri cation, CAV'07, pages 258{262, Berlin, Heidelberg, 2007. Springer-Verlag.

[22]. H. Kress-Gazit, G. Fainekos, and G. Pappas. Temporal-logic-based reactive mission and motion

[23]. planning. Robotics, IEEE Transactions on, 25(6):1370{1381, Dec 2009.

[24]. O. Kupferman and M. Y. Vardi. Model checking of safety properties. Form. Methods Syst. Des., 19(3):291{314, Oct. 2001.

[25]. M. Lahijanian, J. Wasniewski, S. Andersson, and C. Belta. Motion planning and control from temporal logic speci cations with probabilistic satisfaction guarantees. In Robotics and Automation (ICRA), 2010 IEEE International Conference on, pages 3227{3232, May 2010.

[26]. A. Medina Ayala, S. Andersson, and C. Belta. Temporal logic control in dynamic environments with probabilistic satisfaction guarantees. In Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, pages 3108{3113, Sept 2011.

[27]. R. Milner. Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[28]. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. Lecture notes in computer science, 3855:364{380, 2006.

[29]. A. Pnueli, Y. Sa'ar, and L. D. Zuck. Jtlv: A framework for developing veri cation algorithms. In Proceedings of the 22Nd International Conference on Computer Aided Veri cation, CAV'10, pages 171{174, Berlin, Heidelberg, 2010. Springer-Verlag.