Submitted, 2008 Conference on Decision and Control
http://www.cds.caltech.edu/~murray/papers/wm08-cdc.html

Formal Verification of an Autonomous Vehicle System

Tichakorn Wongpiromsarn and Richard M. Murray

Abstract— **Model checking is a widely used technique for formal verification of distributed systems. It works by effectively examining the complete reachable state space of a model in order to determine whether the system satisfies its requirements or desired properties. The complexity of an autonomous vehicle system, however, renders model checking of the entire system infeasible due to the state explosion problem. In this paper, we illustrate how to exploit the structure of the system to systematically decompose the overall system-level requirements into a set of component-level requirements. Each of the components can then be model checked separately. A case study is presented where we formally verify the state consistency between different software modules of Alice, an autonomous vehicle developed by the California Institute of Technology for the 2007 DARPA Urban Challenge.**

I. INTRODUCTION

Due to the increase in complexity of hardware and software, traditional testing-based verification techniques are not adequate to ensure the reliability of the system. In addition, since many systems are designed to operate in a wide range of environments and evolve over time, the range of possible test scenarios becomes extremely large and unmanageable. Formal specification and verification, an alternative approach which guarantees that the requirements or the desired properties are satisfied for any possible execution of the system, have therefore been active areas of research in the distributed system community for more than thirty years. The approach consists of two key elements: a specification language for describing the system and its requirements and an analysis to verify the correctness of the system specification, relative to the requirements.

Control systems are generally described by a set of differential equations. Theories of differential equations and tools such as the Lyapunov function approach and the sum of squares technique have been developed to verify stability and safety properties of the system [1], [2]. This framework, however, is not suitable for describing certain classes of systems and other logical properties such as liveness. In addition, it does not address the concurrency issues of distributed systems. Formal specification and verification techniques and tools from the distributed system community need to be employed in order to ensure the reliability of these systems.

Formal specification and verification of robotic systems has been a topic of growing interest since 1990 [3]. Since these systems are never purely continuous or discrete, the main challenge is to intimately combine the two components of the systems. In 1995, Nancy Lynch introduced the hybrid I/O automaton (HIOA) model which is capable of describing both continuous and discrete behavior [4]. Although this model is expressive, it requires a lot of calculations that need to be done manually. HIOA has been applied, for example, to prove the correctness of a vehicle deceleration maneuver part of an automated transportation system [5] and to verify the safety of the automated highway system of the California PATH project and the Traffic Alert and Collision Avoidance System (TCAS) that is used by aircraft to avoid midair collisions [6], [7].

The contribution of this paper is twofold. First, this paper illustrates how to exploit the structure of the system, imposed by the Canonical Software Architecture (CSA) [8], to address the state explosion issues in model checking and make formal verification of complex systems possible. Second, we apply formal specification and verification, the technique mainly used in the distributed system community, to prove certain properties of Alice, an autonomous vehicle developed by the California Institute of Technology for the 2007 DARPA Urban Challenge. Alice is a distributed system consisting of more than 100 threads whose interactions render the safety verification techniques based on control theory inapplicable. In addition, liveness properties also need to be verified in order to prove the correctness of the system. As opposed to safety properties which indicate that something bad will never happen, liveness properties indicate that eventually something good will happen. The remainder of this paper is organized as follows. Section II provides the background, including a brief description of the CSA, some existing specification languages and model checkers and approaches to composing specifications. Section III describes how to use CSA to systematically decompose system-level requirements into a set of component-level requirements. Section IV presents a case study where the overall system-level requirements are decomposed into a set of component-level requirements whose verification is computationally simple. Section V concludes the paper and discusses some future work.

II. BACKGROUND

A. Canonical Software Architecture

A Canonical Software Architecture has been developed to support a hierarchical decomposition and separation of functionality in the planning subsystem, while maintaining communication and contingency management. In CSA, we can think of the entire system as being broken up into "modules," each of which has a separate, dedicated function. There are two types of modules in CSA: estimation modules and

This work was supported by AFOSR.

T. Wongpiromsarn and R.M. Murray are with the Division of Engineering and Applied Science, California Institute of Technology, Pasadena, CA 91125, USA `nok@caltech.edu`, `murray@cds.caltech.edu`

control modules. This section only discusses CSA control modules, which are the focus of this paper.

The CSA imposes a structure on both the interface between control modules and the major operations that happen within a control module. As shown in Fig. 1, inputs to a CSA control module are restricted to be one of the following: state information, directives/instructions (from other modules wishing to control this module), and responses/status reports (from other modules receiving instructions from this module). The outputs can only be either status reports from this module or directives/instructions for other control modules.

For each directive that a control module is designed to accept, the following must be specified: (1) entry condition; (2) exit condition; (3) constraints that must be satisfied during the execution of the directive; and (4) performance criteria (performance or other items to be optimized). The entry and exit conditions define, respectively, what must be true before starting to execute this directive and what must be true to complete the execution of this directive. For each directive received, a response which indicates rejection, acceptance, failure or completion of the directive and the reason for rejection or failure must be reported to the source of the directive. Rejection or failure of a directive occurs when the entry or exit condition is not readily achievable, the deadlines aren't met, or one of the constraints cannot be satisfied.

To separate communiation requirements from the given module's core function requirements, the CSA decomposes a module into three components: *Arbitration*, *Control* and *Tactics*. *Arbitration* is responsible for (1) managing the overall behavior of the control module by issuing a merged directive, computed from all the received directives, to the *Control*; and (2) reporting rejection, acceptance, failure and completion of a received directive to the *Control* of the issuing control module. *Control* is responsible for (1) computing the output directives to the controlled module(s) or the commands to the hardware based on the merged directive, received responses and state information; and (2) reporting failure and completion of a merged directive to the *Arbitration*. *Tactics* provides the core functionality of the control module and is responsible for providing the logic used by the *Control* for computing output directives.

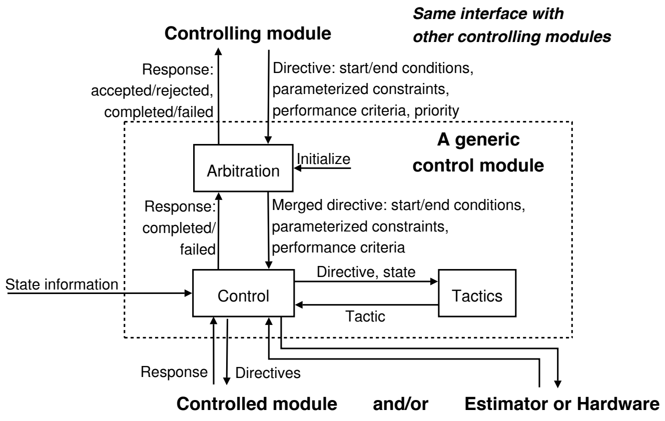
Controlling module
Same interface with other controlling modules
Response: accepted/rejected, completed/failed
Directive: start/end conditions, parameterized constraints, performance criteria, priority
Arbitration
Initialize
A generic control module
Response: completed/ failed
Merged directive: start/end conditions, parameterized constraints, performance criteria
State information
Control
Directive, state
Tactic
Tactics
Response
Directives
Controlled module
and/or
Estimator or Hardware

Fig. 1. A generic control module in the Canonical Software Architecture.

B. Specification Languages and Model Checkers

A variety of specification languages have been developed starting in the 1970's. In 1977, Amir Pnueli introduced the use linear temporal logic (LTL) as a specification language [9]. Examples of frequently used formulae are

- **always** p (invariance): $\Box p$
- **eventually** p (guarantee): $\diamond p$
- p **implies eventually** q (response): $p \Longrightarrow \diamond q$
- p **implies** q **until** r (precedence): $p \Longrightarrow q\, \mathcal{U}\, r$
- **always eventually** p (progress): $\Box \diamond p$
- **eventually always** p (stability): $\diamond\Box p$
- **eventually** p **implies eventually** q (correlation): $\diamond p \Longrightarrow \diamond q$

In practice, however, most systems cannot be described by a single LTL formula, so in the 1980's, Lamport introduced Temporal Logic of Actions (TLA) which makes it practical to describe a system by a single formula [10], [11], [12]. In TLA, a module is specified in terms of its behavior (a sequence of states). Behavior is described by an initial predicate and an action. TLA uses LTL to reason about behavior of systems but Lamport also introduced new kinds of temporal assertions to make the specifications simpler and easier to understand. A system is specified by specifying a set of possible behavior, for example, a specification can be written as

$$Spec \triangleq Init \wedge \Box Action.$$

From system's specifications, one can derive a theorem which is a temporal formula satisfied by every behavior. Generally, we have

$$Theorem \triangleq Spec \Longrightarrow \Box Properties.$$

In 1999, Yu et al. developed TLC, a model checker for specifications written in TLA+ which is a specification language based on TLA [13]. Similar to other model checkers, TLC relies on a finite-state model of the system and performs an exhaustive state space search to check that the system requirements are satisfied. TLC may never terminate if the set of reachable states is not finite.

Parallel to the development of TLA, Bell Labs invented Process Meta-Language (PROMELA) in 1979 [14]. This language was influenced by Dijkstra (1975), Hoare's CSP language and C. It emphasizes the modeling of process synchronization and coordination, not computation and is not meant to be analyzed manually. SPIN is a model checker for specifications written in PROMELA [14]. It has two modes: simulation and verification. The simulation mode performs random or iterative simulations of the modeled system's execution while the verification mode generates a C program that performs a fast exhaustive verification of the system state space. SPIN is mainly used for checking for deadlocks, livelocks, unspecified receptions, and unexecutable code, correctness of system invariants, non-progress execution cycles. It also supports the verification of linear time temporal constraints.

HyTech, a model checking tool for automatic verification of hybrid systems, was introduced in 1995 [15], [16]. Its successor, PHAVer, was designed to address many limitations of HyTech such as the overflow problem which prohibits the use of HyTech with complex systems [17]. Both HyTech and PHAVer are symbolic model checkers for linear hybrid automata, a subclass of hybrid automata which are defined by linear predicates and piecewise constant bounds on the derivatives (i.e. a system whose the dynamics of the continuous variables are defined by linear differential inequalities of the form $A\dot{x} \sim b$ where $\sim \in \{\leq, \geq\}$, A is a constant matrix and c is a constant vector).

C. Composing Specifications and Properties

As systems become more complex, writing specifications for the entire systems becomes more difficult and may lead to errors if not done carefully. Researchers have been studying approaches to composing specifications and properties (or requirements), two of which are discussed in this paper:

1) Composing components' specifications [18]: This approach uses shared state. First, components' specifications are described. The specification of the system is basically the conjunction of its components' specifications. System's properties can then be verified using a model checker such as TLC.
2) Composing components' properties [19], [20], [21]: An alternative to composing components' specification is to first verify components' properties from their specifications. System's properties are then derived from the components' properties. This approach hides substantial parts of correctness proofs in components verifications. It requires putting a substantial amount of effort into proving that a component has properties that are useful in composition. Proofs are achieved at the component level and can be reused each time the component is part of the new system. This approach is useful when the effort required to find and compose components is less than the effort required to design an entire system from scratch.

III. DECOMPOSITION OF SYSTEM REQUIREMENTS

As described in Section II-A, the inputs, outputs and major operations within a CSA control module have a well-defined structure. A natural way to decompose system requirements is therefore to break them down into the requirements for the *Arbitration* component and the requirements for the *Control* and *Tactics* components for each of the modules. The requirements for the *Arbitration* component specify the relationship between the received directives and the merged directives while the requirements for the *Control* and *Tactics* components specify the relationship between the merged directives, responses, state knowledge and the output directives as shown in the next section.

Besides module requirements, communication requirements such as bandwidth, packet drop, delay, etc also need to be specified. Modules' requirements and communiation requirements can then be composed either manually or by using tools such as theorem provers [22], [23] to verify that they are sufficient to ensure that the system requirements are satisfied.

IV. APPLICATIONS TO AN AUTONOMOUS VEHICLE SYSTEM

Alice is a modified Ford E350 van (see Fig. 2), equipped with mechanical actuators (brake, throttle, steering and transmission), sensors (LADARs, RADARs and cameras) and an Applanix INS (for state estimation). The sensing and planning subsystems have been developed so that Alice could navigate in a fully autonomous manner through a partially known environment populated with static and dynamic obstacles. The sensing subsystem provides a representation of the environment around the vehicle. The planning subsystem determines and executes desired motion of the vehicle to satisfy the mission goals, which include crossing GPS waypoints, avoiding obstacles, following traffic rules, etc.

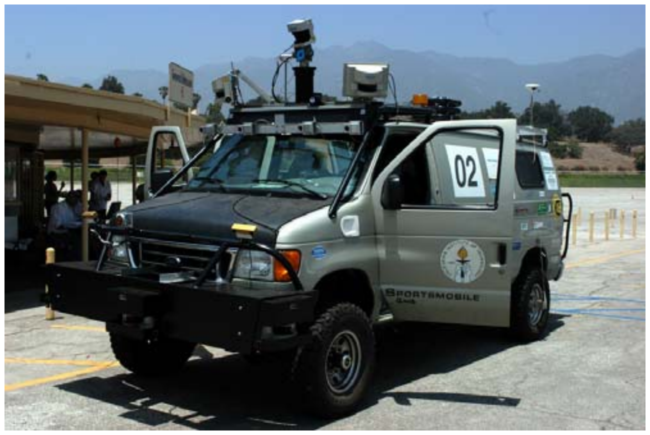

Fig. 2. Alice, Team Caltechs entry in the 2007 Urban Challenge.

The planning subsystem in Alice consists of four software modules: the mission planner, the trajectory planner, the trajectory follower and the drive control, as shown in Fig. 3 [8]. The example considered in this section focuses on the two lower-level modules: the trajectory follower and the drive control. The trajectory follower receives a reference trajectory and computes commands to throttle, brake, steering and transmission that enable Alice to track the trajectory. The drive control module consists of 4 CSA control modules: the actuation interface, the acceleration module, the transmission module and the steering module. It receives actuator commands from the trajectory follower and an emergency stop (estop) command from DARPA, and performs checking to make sure that the commands are reasonable. For example, the gear can be changed only when Alice is stopped. Based on the received commands and actuators' states, the drive control computes commands to all the actuators.

In this section, we show that CSA can ensure the state consistency between different software modules. Specifically, we want to prove that the trajectory follower, the module that commands a gear change, has the right knowledge about the

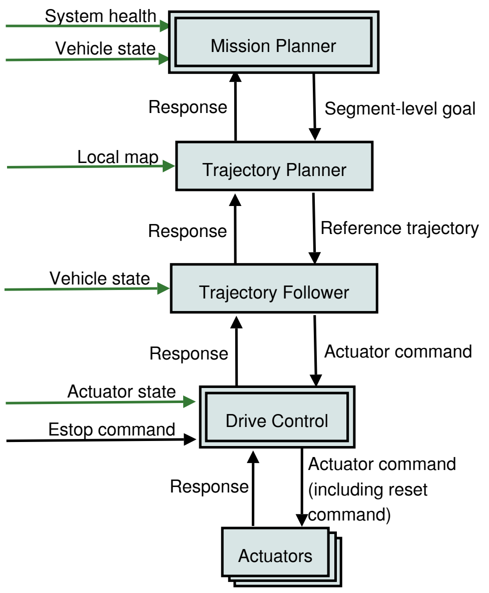

Fig. 3. The planning subsystem in Alice. Boxes with double lined borders are software modules that will be broken up into multiple CSA modules. Green arrows represent state knowledge while black arrows represent directives/responses.

gear Alice is currently in even though it does not talk to the actuator directly and sensors may fail. Otherwise, it will command full brake. This example involves six components—the *Control* of the trajectory follower, the actuation interface, the transmission module, the acceleration module, the actuators and the network—as shown in Fig. 4.

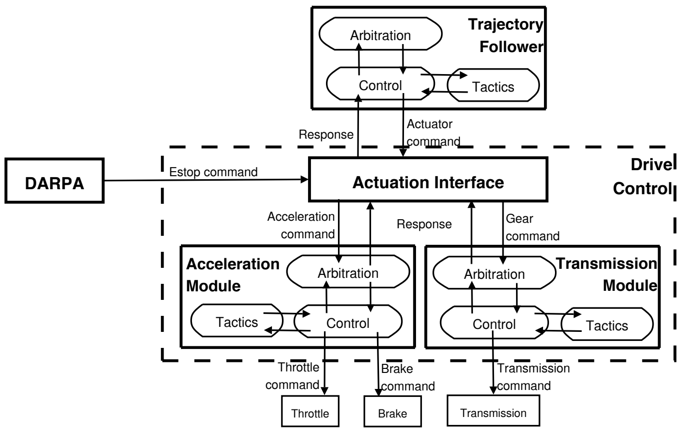

Fig. 4. The components involved in the CSA example.

In this example, we are only interested in acceleration and transmission commands. The following variables are involved in this example as shown in Fig. 5:

- $Trans_{f,s}$: transmission directive sent from the follower;
- $Trans_{f,r}$: transmission directive received by the actuation interface;
- $Trans_{a,s}$: transmission directive sent from the actuation interface;
- $Trans_{a,r}$: transmission directive received by the transmission module;
- $Acc_{f,s}$: acceleration directive sent from the follower;
- $Acc_{f,r}$: acceleration directive received by the actuation interface;
- $Acc_{a,s}$: acceleration directive sent from the actuation interface;
- $Acc_{a,r}$: acceleration directive received by the acceleration module;
- $TransResp_{f,s}$: response sent from the actuation interface;
- $TransResp_{f,r}$: response received by the follower;
- $TransResp_{a,s}$: response sent from the transmission module;
- $TransResp_{a,r}$: response received by the actuation interface.

Each of these variables is represented by a finite sequence, whose n^{th} element represents its value in the n^{th} cycle, with the following operators:

- $Last(s)$: The last element of sequence s;
- $Len(s)$: The length of sequence s;
- $s[n]$: The n^{th} element of sequence s.

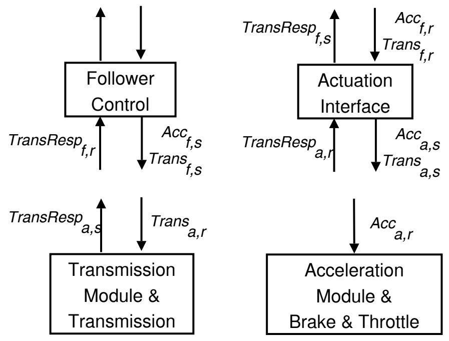

Fig. 5. The variables involved in the CSA example. $TransResp_{a,s}$, $TransResp_{a,r}$, $TransResp_{f,s}$, $TransResp_{f,r}$ $\in$ {C, F}, where C $\equiv$ COMPLETED and F $\equiv$ FAILED.

Let $Trans$ be the actual gear and $Trans_f$ be the gear that the trajectory follower thinks Alice is in. Assume that when $Len(Trans_{f,s}) = 0$ (i.e. before any command is sent from the trajectory follower), $Trans_f = Trans$, we can prove the following desired system-level properties:

1) The trajectory follower has the right knowledge about the gear that Alice is currently in, or it commands full brake. Mathematically, this can be written as:

$$\begin{aligned}\Box((&Len(TransResp_{f,r}) = Len(Trans_{f,s}) \wedge \\ &Last(TransResp_{f,r}) = \mathrm{C}) \\ &\implies Trans_f = Trans),\end{aligned} \qquad (1)$$

$$\Box(Trans_f = Trans \ \vee \ Acc_{f,s} = -1). \qquad (2)$$

2) At infinitely many instants, the trajectory follower has the right knowledge about the gear that Alice is currently in, or a hardware failure (HWF) occurs:

$$\Box \diamond (Trans_f = Trans \ \ \vee \ \ HWF). \qquad (3)$$

We assume the following component-level properties:

1) Transmission module and transmission actuator:

- The number of responses cannot be greater than the number of directives. This can be formalized by the following LTL formula:

$$\Box(Len(TransResp_{a,s}) \leq Len(Trans_{a,r})). \quad (4)$$

- For each of the directives the transmission module receives, a response will eventually be sent. If the gear is successfully changed, the completion of the directive will be reported. Otherwise, a hardware failure occurs and the failure will be reported. This assumption can be mathematically represented by the following three LTL formulae:

$$\begin{gathered}\Box(n = Len(Trans_{a,r}) \implies \\ \Diamond((Trans = Trans_{a,r}[n] \wedge \\ TransResp_{a,s}[n] = \mathrm{C}) \vee \\ (HWF \wedge TransResp_{a,s}[n] = \mathrm{F})));\end{gathered} \quad (5)$$

$$\begin{gathered}\Box(Last(TransResp_{a,s}) = \mathrm{C} \implies \\ Trans = Trans_{a,r}[Len(TransResp_{a,s})]);\end{gathered} \quad (6)$$

$$\Box(Last(TransResp_{a,s}) = \mathrm{F} \implies HWF). \quad (7)$$

2) Actuation interface: All the transmission directives and responses received are always sent (to the transmission module and to the follower, respectively). This assumption can be described by the following two LTL formulae:

$$\begin{gathered}\Box(Len(Trans_{a,s}) = Len(Trans_{f,r}) \wedge \\ \forall i \in \{1, \ldots, Len(Trans_{f,r})\} : \\ Trans_{a,s}[i] = Trans_{f,r}[i]);\end{gathered} \quad (8)$$

$$\begin{gathered}\Box(Len(TransResp_{f,s}) = Len(TransResp_{a,r}) \\ \wedge \ \forall i \in \{1, \ldots, Len(TransResp_{a,r})\} : \\ TransResp_{f,s}[i] = TransResp_{a,r}[i]).\end{gathered} \quad (9)$$

3) Network: All messages are eventually delivered. An example of this assumption for the transmission directive sent from the actuation interface and received by the transmission module, formalized in LTL, is given by

$$\begin{gathered}\Box(Len(Trans_{a,r}) \leq Len(Trans_{a,s})) \wedge \\ \forall i \in \{1, \ldots, Len(Trans_{a,r})\} : \\ Trans_{a,r}[i] = Trans_{a,s}[i]).\end{gathered} \quad (10)$$

4) The *Control* of the Trajectory Follower:
   - If the response is not yet received, send a brake command:

$$\begin{gathered}\Box(Len(TransResp_{f,r}) \neq Len(Trans_{f,s}) \\ \implies Acc_{f,s} = -1).\end{gathered} \quad (11)$$

   - If the last response indicates failure, send a brake command:

$$\begin{gathered}\Box(Last(TransResp_{f,r}) = \mathrm{F} \\ \implies Acc_{f,s} = -1).\end{gathered} \quad (12)$$

   - Do not send a new directive until a response for the last directive is received:

$$\begin{gathered}\Box(Len(Trans_{f,s}) \leq \\ Len(TransResp_{f,r}) + 1).\end{gathered} \quad (13)$$

   - Infinitely often, the number of the transmission directives is not greater than the number of the responses (i.e. once a response is received, follower processes it before sending out another directive):

$$\begin{gathered}\Box \Diamond (Len(Trans_{f,s}) \leq \\ Len(TransResp_{f,r})).\end{gathered} \quad (14)$$

   - If the last response indicates completion of the directive, follower updates $Trans_f$ to the corresponding directive:

$$\begin{gathered}\Box(Last(TransResp_{f,r}) = \mathrm{C} \implies \\ Trans_f = Trans_{f,s}[Len(TransResp_{f,r})]).\end{gathered} \quad (15)$$

To prove the system-level properties, we use the following lemmas and proposition:

Lemma 4.1: Any execution of the program satisfies the following properties:

1) $\Box(Len(TransResp_{a,r}) \leq Len(Trans_{a,s})$;
2) $\Box(Len(TransResp_{f,r}) \leq Len(Trans_{f,s})$;
3) $\Box(Len(Trans_{a,s}) \leq Len(TransResp_{a,r}) + 1)$.

Proof: These properties can be easily derived from the assumptions about the network, (4), (8), (9) and (13). ■

Lemma 4.2: Any execution of the program satisfies the following properties:

1) $\Box((Len(TransResp_{a,r}) = Len(Trans_{a,s})) \vee (Len(Trans_{a,s}) = Len(TransResp_{a,r}) + 1)$;
2) $\Box((Len(TransResp_{f,r}) = Len(Trans_{f,s})) \vee (Len(Trans_{f,s}) = Len(TransResp_{f,r}) + 1)$.

Proof:

1) Let

$$\begin{aligned} A &\equiv Len(Trans_{a,s}) \geq Len(TransResp_{a,r}) \\ B &\equiv Len(Trans_{a,s}) \leq Len(TransResp_{a,r}) \\ C &\equiv Len(Trans_{a,s}) = Len(TransResp_{a,r}) + 1 \end{aligned}$$

From Lemma 4.1, we get that any execution satisfies

$$\Box((A \wedge B) \ \vee \ (A \wedge C)).$$

Since

$$A \wedge B \equiv Len(TransResp_{a,r}) = Len(Trans_{a,s})$$

and

$$A \wedge C \equiv Len(Trans_{a,s}) = Len(TransResp_{a,r}) + 1,$$

this completes the proof.

2) This can be proved using Lemma 4.1(2) and property (13) and following the same steps as in the previous proof.

■

Lemma 4.3: Any execution of the program satisfies

$$\begin{aligned}
&\Box(Len(TransResp_{f,r}) = Len(Trans_{f,s})\\
\implies\ & Len(TransResp_{f,r}) = Len(TransResp_{f,s}) =\\
& Len(TransResp_{a,r}) = Len(TransResp_{a,s}) =\\
& Len(Trans_{a,r}) = Len(Trans_{a,s}) =\\
& Len(Trans_{f,r}) = Len(Trans_{f,s})).
\end{aligned}$$

Proof: This can be easily derived from the assumptions about the network, (4), (8) and (9). ■

Lemma 4.4: Any execution of the program satisfies

$$\begin{gathered}
\Box((Len(TransResp_{f,r}) \leq Len(TransResp_{f,s}) \leq\\
Len(TransResp_{a,r}) \leq Len(TransResp_{a,s})) \quad \wedge\\
(\forall i \in \{1, \ldots, Len(TransResp_{f,r})\} :\\
TransResp_{f,r}[i] = TransResp_{f,s}[i] =\\
TransResp_{a,r}[i]) = TransResp_{a,s}[i])).
\end{gathered}$$

Proof: This is clear from (9) and the assumptions about the network. ■

Proposition 4.1: The following propositional formula is a tautology:

$$((\neg A \vee B) \wedge (A \vee C)) \quad \implies \quad B \vee C.$$

Proof: This can be easily proved using the truth table. ■

The system-level properties can then be proved as follows:

Theorem 4.1: Any execution of the program satisfies

$$\begin{gathered}
\Box((Len(TransResp_{f,r}) = Len(Trans_{f,s}) \quad \wedge\\
Last(TransResp_{f,r}) = \text{COMPLETED})\\
\implies \quad Trans_f = Trans).
\end{gathered}$$

Proof: The case where $Len(Trans_{f,s}) = 0$ is trivial so we only consider the case where $Len(Trans_{f,s}) > 0$. Suppose $Len(TransResp_{f,r}) = Len(Trans_{f,s})$ and $Last(TransResp_{f,r}) = \text{COMPLETED})$. Then, we get

$$\begin{aligned}
Trans_f &\overset{(15)}{=} Trans_{f,s}[Len(TransResp_{f,r})]\\
&\overset{\text{Lemma4.3}}{=} Trans_{f,s}[Len(Trans_{a,s})]\\
&\overset{(8),\text{network}}{=} Trans_{a,s}[Len(Trans_{a,s})].
\end{aligned}$$

Also, from Lemma 4.3 and Lemma 4.4, we get

$$Last(TransResp_{a,s}) = \text{COMPLETED}.$$

Using (6), we can then conclude that

$$\begin{aligned}
Trans &= Trans_{a,s}[Len(TransResp_{a,s})]\\
&\overset{\text{Lemma4.3}}{=} Trans_{a,s}[Len(Trans_{a,s})]\\
&= Trans_f.
\end{aligned}$$

■

Theorem 4.2: Any execution of the program satisfies

$$\Box(Trans_f = Trans \quad \vee \quad Acc_{f,s} = -1).$$

Proof: From (11),

$$\begin{gathered}
\Box(Len(TransResp_{f,r}) \neq Len(Trans_{f,s}) \quad \implies\\
Acc_{f,s} = -1).
\end{gathered}$$

Or equivalently,

$$\begin{gathered}
\Box(Len(TransResp_{f,r}) = Len(Trans_{f,s}) \ \vee\\
Acc_{f,s} = -1).
\end{gathered}$$

Similarly, from (12), we get

$$\Box(Last(TransResp_{f,r}) = \text{C}) \ \vee \ Acc_{f,s} = -1).$$

Let

$$\begin{aligned}
A &\equiv Len(TransResp_{f,r}) = Len(Trans_{f,s})\\
B &\equiv Last(TransResp_{f,r}) = \text{COMPLETED}\\
C &\equiv Trans_f = Trans\\
D &\equiv Acc_{f,s} = -1
\end{aligned}$$

The system has the following property

$$\begin{gathered}
\Box(((A \wedge B) \Longrightarrow C) \ \wedge \ (A \vee D) \ \wedge \ (B \vee D)) \quad \equiv\\
\Box((\neg A \vee \neg B \vee C)) \ \wedge \ (A \vee D) \ \wedge \ (B \vee D)).
\end{gathered}$$

Applying Proposition 4.1 twice, we can complete the proof. ■

Theorem 4.3: Any execution of the program satisfies

$$\Box \diamond (Trans_f = Trans \quad \vee \quad HWF).$$

Proof: From Lemma 4.2(2),

$$\begin{gathered}
\Box((Len(TransResp_{f,r}) = Len(Trans_{f,s})) \quad \vee\\
(Len(Trans_{f,s}) = Len(TransResp_{f,r}) + 1)).
\end{gathered}$$

Consider an arbitrary k_1^{th} cycle. From (14) and Lemma 4.2, $\exists k > k_1$ such that in the k^{th} cycle, $Len(TransResp_{f,r}) = Len(Trans_{f,s})$.

Consider this k^{th} cycle. The case where $Len(Trans_{f,s}) = 0$ is trivial. (By assumption, $Trans_f = Trans$.) So we only consider the case where $Len(Trans_{f,s}) > 0$. If $Last(TransResp_{f,r}) =$ COMPLETED, then from Theorem 4.1, (15) and Lemma 4.3,

$$Trans_f = Trans = Last(Trans_{f,s}).$$

Otherwise, $Last(TransResp_{f,r}) = Last(TransResp_{a,s})$ = FAILED, so from (7), HWF. ■

V. CONCLUSION AND FUTURE WORK

Although model checking can provide useful information about a system's correctness and reveal subtle errors in design, its main disadvantage is the state explosion problem. A promising approach to address this problem is to reason about the system from its components. By exploiting the structure of the system imposed by the CSA, this paper illustrates how to decompose computationally demanding system-level requirements into a set of component-level requirements whose verification is computationally simple. The technique is applied to verify the state consistency between two software modules of Alice, an autonomous vehicle developed by the California Institute of Technology for the 2007 DARPA Urban Challenge.

Tools such as theorem provers [22], [23] may need to be explored to extend the technique presented in this paper to a more complex system. In contrast to model checking, theorem proving can deal directly with infinite state spaces. The drawback, however, is that it requires interaction with a human, so the theorem proving process is slow and often error prone. Combining model checking and theorem proving such that we benefit from the advantages of both approaches is therefore an obvious extension of this work.

Another direction of research is to extend this technique to a more general class of systems governed by a finite state machine. The CSA basically breaks up the system-level finite state machine into a set of component-level finite state machines such that the transitions between different finite state machines are simple and have a well-defined structure. We can potentially apply the idea to a more general class of systems whose finite state machine can be broken up such that the transitions between different component-level finite state machines are simple but not necessarily have the same structure as the CSA.

VI. Acknowledgments

The authors gratefully acknowledge Michel Charpentier for his help, suggestions and ideas regarding the use of TLA and its tools. This work was supported by AFOSR.

References

[1] S. Prajna, A. Papachristodoulou, and P. Parrilo, "Introducing SOS-TOOLS: A general purpose sum of squares programming solver," in *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pp. 741–746, 2002.

[2] H. Yazarel, S. Prajna, and G. J. Pappas, "S.O.S. for safety.," in *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pp. 461–466, 2004.

[3] B. Espiau, K. Kapellos, M. Jourdan, and D. Simon, "On the validation of robotics control systems part I: High level specification and formal verification," Tech. Rep. RR-2719, INRIA, November 1995.

[4] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg, "Hybrid I/O automata," in *DIMACS Workshop on Verification and Control of Hybrid Systems*, October 1995.

[5] N. Lynch and H. Weinberg, "Proving correctness of a vehicle maneuver: Deceleration," in *the Second European Workshop on Real-Time and Hybrid Systems*, June 1995.

[6] E. Dolginova and N. A. Lynch, "Safety verification for automated platoon maneuvers: A case study," in *International Workshop on Hybrid and Real-Time Systems*, pp. 154–170, 1997.

[7] C. Livadas, J. Lygeros, and N. Lynch, "High-level modeling and analysis of TCAS," in *Proceedings of the 20th IEEE Real-Time Systems Symposium*, (Phoenix, Arizona), pp. 115–125, December 1999.

[8] T. Wongpiromsarn and R. M. Murray, "Distributed mission and contingency management for the DARPA Urban Challenge," in *International Workshop on Intelligent Vehicle Control Systems (IVCS 2008)*, (Madeira, Portugal), May 2008. submitted.

[9] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pp. 46–57, IEEE, 1977.

[10] M. Abadi and L. Lamport, "An old-fashioned recipe for real time," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1543–1571, September 1994.

[11] L. Lamport, "Specifying concurrent program modules," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 190–222, April 1983.

[12] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 872–923, May 1994.

[13] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *Conference on Correct Hardware Design and Verification Methods*, pp. 54–66, 1999.

[14] G. J. Holzmann, *The Spin Model Checker*. Boston: Addison-Wesley, 2004.

[15] R. Alur, T. A. Henzinger, and P.-H. Ho, "Automatic symbolic verification of embedded systems," in *IEEE Transactions on Software Engineering*, pp. 181–201, 1996.

[16] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 110–122, 1997.

[17] G. Frehse, "Phaver: Algorithmic verification of hybrid systems past hytech," in *HSCC*, pp. 258–273, 2005.

[18] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 507–535, May 1995.

[19] M. Charpentier and K. M. Chandy, "Examples of program composition illustrating the use of universal properties," in *IPPS/SPDP Workshops*, pp. 1215–1227, 1999.

[20] M. Charpentier and K. M. Chandy, "Towards a compositional approach to the design and verification of distributed systems," in *World Congress on Formal Methods (1)*, pp. 570–589, 1999.

[21] M. Charpentier and K. M. Chandy, "Theorems about composition," in *Mathematics of Program Construction*, pp. 167–186, 2000.

[22] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*. No. 5 in Harper & Row Computer Science and Technology Series, New York: Harper & Row, 1986.

[23] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)* (D. Kapur, ed.), vol. 607 of *Lecture Notes in Artificial Intelligence*, (Saratoga, NY), pp. 748–752, Springer-Verlag, jun 1992.