Compositional Verification of Architectural Models

Darren Cofer[1], Andrew Gacek[1], Steven Miller[1], Michael Whalen[2], Brian LaValley[3],
and Lui Sha[4]

[1] Rockwell Collins Advanced Technology Center
{ddcofer, ajgacek, spmiller}@rockwellcollins.com
[2] University of Minnesota, whalen@cs.umn.edu
[3] WW Technology Group, blavalley@wwtechgroup.com
[4] University of Illinois, lrs@uiuc.edu

Abstract. This paper describes a design flow and supporting tools to significantly improve the design and verification of complex cyber-physical systems. We focus on system architecture models composed from libraries of components and complexity-reducing design patterns having formally verified properties. This allows new system designs to be developed rapidly using patterns that have been shown to reduce unnecessary complexity and coupling between components. Components and patterns are annotated with formal contracts describing their guaranteed behaviors and the contextual assumptions that must be satisfied for their correct operation. We describe the compositional reasoning framework that we have developed for proving the correctness of a system design, and provide a proof of the soundness of our compositional reasoning approach. An example based on an aircraft flight control system is provided to illustrate the method and supporting analysis tools.

Keywords: Cyber-physical systems, design patterns, formal methods, model checking, compositional verification, SysML, AADL, META, DARPA

1 Introduction

Advanced capabilities being developed for the next generation of commercial and military aircraft will be based on complex new software. These aircraft will incorporate adaptive control algorithms and sophisticated mission software providing enhanced functionality and robustness in the presence of failures and adverse flight conditions. Unmanned aircraft have already displaced manned aircraft in most surveillance missions and are performing many combat missions with increasing levels of autonomy. Manned and unmanned aircraft will be required to coordinate their activities safely and efficiently in both military and commercial airspace.

The *cyber-physical systems* that provide these capabilities are so complex that software development and verification is one of the most costly development tasks and therefore poses the greatest risk to program schedule and budget. Without significant changes in current development processes, the cost and time of software development will become the primary barriers to the deployment of the advanced capabilities needed for the next generation of military aircraft.

DARPA's META program was undertaken to significantly improve the design, manufacture, and verification process for complex cyber-physical systems. The work described in this paper directly addresses this goal by allowing the system architecture to be composed from libraries of *complexity-reducing design patterns* with formally guaranteed properties. This allows new system designs to be developed rapidly using patterns that have been shown to reduce unnecessary complexity and coupling between components. This work also deeply embeds formal verification into the design process to enable correct-by-construction development of systems that work the first time. The use of components with formally specified contracts, design patterns that provide formally guaranteed properties, and an architectural modeling language with a well-defined semantics ensures that the system design is known to meet its requirements even before it is implemented. Further details can be found in [1].

In previous work, we have successfully applied model checking to software components that have been created using model-based development (MBD) tools such as Simulink [7]. Our objective in this project was to build on this success and extend the reach of model checking to system design models. Examples of previous work in this area include approaches that essentially flatten the system model by elaborating each component and including its implementation in the same language used for the system [12]. This approach permits accurate modeling of component behaviors and interactions, but suffers from limited scalability. An alternative approach replaces each component with a state machine description that is an abstraction of the component design [11]. This provides better scalability, but can result in the component descriptions that diverge from their implementations and can limit the expressiveness of the overall system model.

The compositional approach we advocate in this paper attempts to exploit the verification effort and artifacts that are already part of our software component verification work. We do this through the use of formal assume-guarantee contracts that correspond to the component requirements for each component. Each component in the system model is annotated with a contract that includes the requirements and constraints that were specified and verified as part of its development process. We then reason about the system-level behavior based on the interaction of the component contracts. The use of contracts is also extended to architectural design patterns that have been formally verified. This approach allows us to leverage our existing MBD process for software components and provides a scalable way to reason about the system as a whole.

Section 2 of this paper presents our architectural modeling framework and describes how we have used the AADL and SysML languages to formally specify system designs. We have developed a mapping between relevant portions of these languages, as well as an automated translation tool and support for contract annotations. Section 3 briefly describes our formalization of architectural design patterns. These patterns encapsulate several fault-tolerance and synchronization mechanisms, increasing the level of design abstraction and supporting verification reuse. Section 4 describes in more detail our compositional verification approach, and Section 5 presents the formulation of our method and a proof sketch of its soundness, the main technical contribution of the paper. Section 6 presents an example based on an aircraft flight control system, and Section 7 briefly describes our tool framework.

2 Architectural Modeling

Our domain of interest is distributed real-time embedded systems (including both hardware and software), such as comprise the critical functionality in commercial and military aircraft. MBD languages and tools are commonly used to implement the components of these systems, but the system-level descriptions of the interactions of distributed components, resource allocation decisions, and communication mechanisms are largely ad hoc. Application of formal analysis methods at the system level requires 1) an abstraction that defines how components will be represented in the system model, and 2) selection of an appropriate formal modeling language.

Assumptions and guarantees. Many aerospace companies have adopted MBD processes for production of software components. As a result of aircraft certification guidelines, these components must have detailed requirements. We have been successful applying formal methods to software component designs because of our decision to conform (as much as possible) to existing trends in industry. By formalizing the component requirements for verification using a combination of model checking and automated translation of the component models, we have made formal analysis accessible to embedded system developers. Therefore, one of our goals in this project was to create a system modeling methodology that would incorporate existing practices and artifacts and be compatible with tools being used in industry.

In this approach, the architectural model includes interface, interconnections, and specifications for components but not their implementation. It describes the interactions between components and their arrangement in the system, but the components themselves are black boxes. The component implementations are described separately by the existing MBD environment and artifacts (or by traditional programming languages, where applicable). They are represented in the system model by the subset of their specifications that is necessary to describe their system-level interactions.

Assume-guarantee contracts [4] provide an appropriate mechanism for capturing the information needed from other modeling domains to reason about system-level properties. In this formulation, guarantees correspond to the component requirements. These guarantees are verified separately as part of the component development process, either by formal or traditional means. Assumptions correspond to the environmental constraints that were used in verifying the component requirements. For formally verified components, they are the assertions or invariants on the component inputs that were used in the proof process.

A contract specifies precisely the information that is needed to reason about the component's interaction with other parts of the system. Furthermore, contract mechanism supports a hierarchical decomposition of verification process that follows the natural hierarchy in the system model.

SysML and AADL. The two modeling languages that we have worked with in this program are SysML and AADL. These languages were developed for different but related purposes. SysML was designed for modeling the full scope of a system, including its users and the physical world, while AADL was designed for modeling real-time embedded systems. While both SysML and AADL are extensible and can be tailored to support either domain, the fundamental constructs each provides reflect these differences. For example, AADL lacks many of the constructs for eliciting system requirements such as SysML requirement diagrams and use cases, and for specifying the behavior of systems such as SysML activity diagrams. On the other hand, SysML lacks many of the constructs needed to model embedded systems such as processes, threads, processors, buses, and memory.

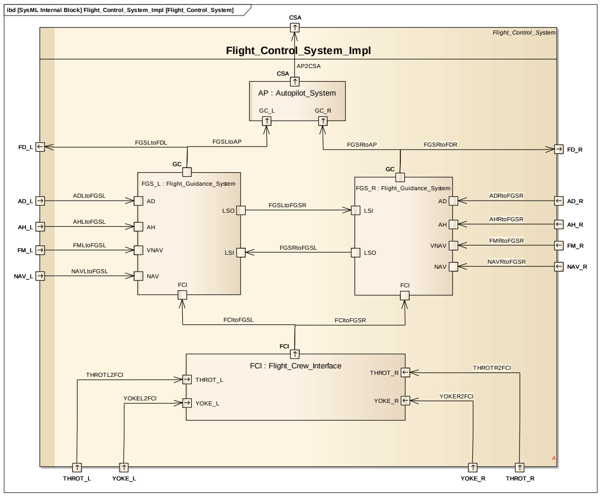


Fig. 1. Flight Control System modeled using SysML.

AADL is a good fit for our domain of interest and provides a sufficiently formal notation. However, AADL has yet to gain traction with many industrial users and its lack of a stable graphical environment (at least in the most popular available tool, the Eclipse-based OSATE) has been a barrier to adoption. Consequently, SysML has been adopted by many organizations for system design specification, even though it has no formal semantics and no common textual representation.

Our solution is to allow developers to do at least their initial system development in SysML, and provide support for automatic translation to AADL for analysis. We have built an Eclipse plugin that provides bidirectional translation between SysML and AADL for the domain in which they overlap. We have defined block stereotypes in SysML that correspond to AADL objects, thus effectively mapping the semantics of AADL onto a subset of SysML. The translation is based on the Enterprise

Architect SysML tool used by Rockwell Collins. An example system is shown in Fig. 1.

For SysML to be used to model embedded systems in the same way that AADL does, SysML blocks and ports need to be tagged with stereotypes corresponding to AADL constructs such as threads and processors. AADL components are represented using SysML Blocks with stereotypes. If a SysML block is not tagged with one of these stereotypes, the translator treats it as an AADL system. AADL features are represented using SysML flow ports with stereotypes. If a SysML flow port is not tagged with one of these stereotypes, the translator treats it as an AADL port.

The translator also translates the package structure from a SysML model to AADL and vice versa. When translating from AADL to SysML, the translator will create a single SysML block diagram for each AADL package with a SysML block drawn for each AADL component type and implementation. The translator will also create a single internal block diagram for each AADL implementation that has subcomponents showing that implementation, its subcomponents, their features, and connections.

3 Architectural Design Patterns

The second technical thrust in our META project was the use of *architectural design patterns*. An architectural design pattern is a transformation applied to a system model that implements some desired functionality in a verifiably correct way. Each pattern can be thought of as a partial function on the space of system models, mapping an initial model to a transformed model with new behaviors. We refer to the transformed system as the instantiation of a pattern.

We have three main objectives in creating architectural design patterns. The first is the encapsulation and standardization of good solutions to recurring design problems. The synchronization and coordination of distributed computing platforms in avionics system is a common source of problems that are often challenging to implement correctly. By codifying verified solutions to these problems and making them available to developers, we raise level of abstraction and avoid “reinventing the wheel.”

Reuse of verification is the second objective. The architectural design patterns are developed in a generic way so that they can be formally verified once, and then reused in many different development projects by changing parameters. Each pattern has a contract associated with it that specifies constraints on the systems to which it can be applied, and specifies the behaviors that can be guaranteed in the transformed system. In this way we are able to amortize the verification effort over many systems.

The final objective is reduction or management of system complexity. An architecture pattern can be said to reduce system complexity if it provides an abstraction that effectively eliminates a type of component interaction in a way that can be syntactically enforced. The PALS (physically asynchronous logically synchronous) architecture pattern is an example in which real time tasks are executed with bounded asynchrony physically but the asynchronous execution is logically equivalent to synchronous execution. This greatly reduces the verification state space.

Four architectural patterns were implemented in this project: PALS, Replication, Leader Selection, and Fusion.

The purpose of the PALS pattern is to make portions of a distributed asynchronous system operate in virtual synchrony. This allows portions of the system logic to be designed and verified as though they will be executed on a synchronous platform, and then deployed in the asynchronous system with the same guaranteed behavior. The pattern relies on certain timing constraints on the delivery and processing of messages that must be enforced by the underlying execution platform. To use the pattern, a group of nodes (systems) is selected that are to execute at approximately the same time at period *T*. The outputs (ports) of these nodes are to be received by other nodes in the group such that all nodes will receive the same values at each execution step. The pattern does not add any new data connections to the model, but assumes that the required connections already exist.

The purpose of the Replication pattern is to create identical copies of portions of the system. This is typically used to implement fault tolerance by assigning the copies to execute on separate hardware platforms with independent failure modes. To use the pattern, one or more nodes (systems) are selected and the number of copies to create is specified. Optional arguments for each input and output port on the selected systems determine how these ports and their connections are handled in the replication process. Each new system and port created is given a unique name. When multiple outputs are created they may be merged by the addition of a new system block to select, average, or vote the outputs.

The purpose of the Leader Selection pattern is to coordinate a group of nodes so that a single node is agreed upon as the ‘leader’ at any given time. The nodes typically correspond to replicated computations hosted on distributed computing resources, and are used as part of a fault-tolerance mechanism. If a replicated node fails, this allows a non-failed node to be selected as the one which will interact with the rest of the system. To use the pattern, a group of *N* nodes (systems or processes) is identified that are to select a leader from among themselves. The leader selection pattern will insert new leader selection threads into each of the systems/processes which are to participate in leader selection. Each thread will have a unique identifier (an integer) to determine its priority in selecting a leader. Connections will be added so that all leader selection threads are able to communicate with each other (*N*-1 input ports, 1 output port). In addition, each leader selection thread will have an input port from which it determines (from other local systems) if it is failed, and an output port which will say if it is the leader. These two ports are initially left unconnected.

The purpose of the Fusion pattern is to insert a component into the architecture that combines several component interfaces into a single interface. The component supplies properties that define the validation/selection algorithm that is used and its impact on the fault tolerance or performance properties of the interfaces. The fusion algorithm could provide voting through exact or approximate agreement or by mid-value selection. The output could correspond to one of the selected inputs or it could be a computing average. To use the pattern, the user will select from a predefined set of fusion algorithms that are presented in a list. Each option will describe the properties and allow the user to browse these as part of the selection process. The user will select the type of component to be inserted in the model to perform the fusion algorithm. There are three initial choices: System (for abstract system designs), Thread (for software implemented voting), and Device for hardware implementations. Finally, the user will select the insertion point for the voter by first selecting an existing architecture component that is the current destination of the interfaces to be voted. After component selection the user will be presented with a list of input interfaces that match the constraints required for the voter that was selected. The user can then select the set of interfaces to which the voter will be applied.

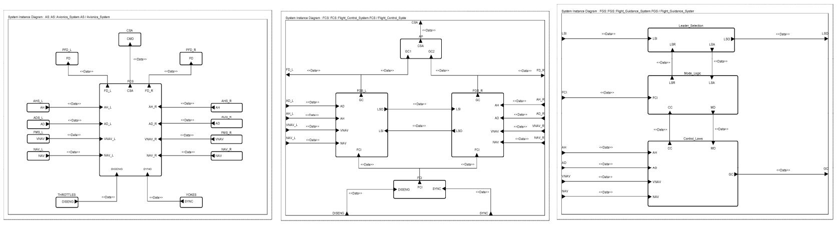

System Instance Diagram : AS::AS::Avionics_System.AS / Avionics_System
CSA
CMD
PFD_L
FD
PFD_R
FD
FCS
FD_L
CSA
FD_R
AHS_L
AH
ADS_L
AD
FMS_L
VNAV
NAV_L
NAV
AH_L
AD_L
VNAV_L
NAV_L
AH_R
AD_R
VNAV_R
NAV_R
AHS_R
ADS_R
FMS_R
NAV_R
DISENG
SYNC
THROTTLES
YOKES
<<Data>>
System Instance Diagram : FCS::FCS::Flight_Control_System.FCS / Flight_Control_Syste
AP
GC1
GC2
FGS_L
FGS_R
GC
LSO
LSI
FCI
VNAV
NAV
System Instance Diagram : FGS::FGS::Flight_Guidance_System.FGS / Flight_Guidance_Syster
Leader_Selection
LSR
LSA
Mode_Logic
CC
MD
Control_Laws

Fig. 2. Avionics System, Flight Control System, and Flight Guidance System models that were used to demonstrate the use of architectural design patterns.

We have applied these patterns in an Avionics System modeled in AADL. Three levels of the system architecture are shown in Fig. 2: the Avionics System, the Flight Control System (FCS), and the Flight Guidance System (FGS). The initial system model to which we apply the patterns captures the functionality of the system under the assumption that nothing ever fails. It only has one set of inputs and outputs and has no redundancy in its implementation. We first apply the replication pattern to the FGS component to create two redundant copies. This pattern automatically replicates ports as necessary and applies a property requiring the copies not be hosted on the same hardware. We next apply the Leader Selection pattern to manage the redundant copies of the FGS. This pattern inserts pre-verified leader selection functionality as new threads inside each FGS to determines the current leader. The Leader Selection protocol that we have used requires that the nodes communicate synchronously. To satisfy this assumption, we apply the PALS synchronization pattern. The constraints of the PALS pattern will be verified during implementation to ensure they can actually be satisfied. Finally, the Fusion pattern is used inside the Autopilot component to combine the two outputs produced by the active and standby FGS copies into a single command input.

4 System Verification

The system-level properties that we wish to verify fall into a number of different categories requiring different verification approaches and tools. This is also true for the contracts that are attached to the components and design patterns used in the system model.

- They may be behavioral properties that describe the state of the system as it changes over time. Behavioral properties may be used to describe protocols governing component interactions in the system, or the system response to combinations of triggering events. We will use the Property Specification Language (PSL) [5] to specify most behavioral properties. An example of a behavioral property associated with the Leader Selection pattern is: *A failed node will not be leader in next step*, or `G(!device_ok[j]- X(leader[i]!= j))`.

- They may be structural properties of the system model to which the pattern is applied (pre-conditions), or of the transformed system model after pattern instantiation (post-conditions). Relationships among timing properties in the model or constraints on the numbers of various objects in the model are in this category.
- Some design patterns rely explicitly on resource allocation properties of the system, including real-time schedulability, memory allocation, and bandwidth allocation. Even after we annotate the model with deadlines and execution times, we must still demonstrate that threads can be scheduled to meet their deadlines. There are many tools available to support verification of these properties, including the ASIIST tool developed by UIUC and Rockwell Collins [8].
- Failure analysis of the system often requires the use of probabilistic methods to demonstrate that the sufficiency of the proposed fault handling mechanisms. The AADL error annex can be used to attach fault behavior models to the system design. As part of the META program we have participated in some demonstrations based on our example AADL model using the PRISM probabilistic model checker [9].

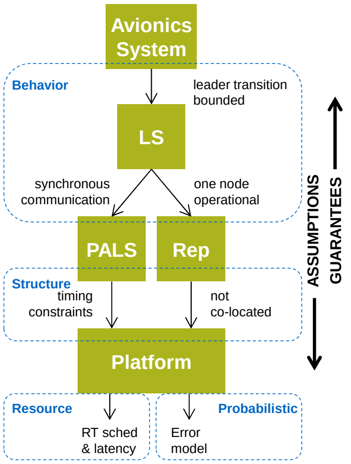


Fig. 3. Contracts between patterns used in the Avionics System example.

At the system level, assumptions and guarantees associated with the system components and patterns interact and are composed to achieve desired system properties. For example, the behavior of the avionics system in our example depends upon guarantees provided by the Leader Selection pattern (Fig. 3). Leader Selection includes an assumption of synchronous data exchange which is satisfied by the PALS pattern guarantees. It also includes an assumption that there will be at least one working node, which is satisfied by the Replication pattern. The PALS pattern, in turn, makes assumptions about the timing properties of the underlying execution platform and the Replication pattern requires that copies are not co-located. Finally, the platform can only guarantee these properties if it verified to satisfy its resource allocation constraints, and its probability of failure is sufficiently low.

The focus of our work is the first two categories: behavioral and structural properties. The next section describes our approach to compositional reasoning.

5 Compositional Reasoning

Our idea is to partition the formal analysis of a complex system architecture into a series of verification tasks that correspond to the decomposition of the architecture. By partitioning the verification effort into proofs about each subsystem within the architecture, the analysis will scale to handle large system designs. Additionally, the approach naturally supports an architecture-based notion of requirements refinement: the properties of components necessary to prove a system-level property in effect define the requirements for those components. We call the tool that we have created for managing these proof obligations *AGREE*: Assume Guarantee Reasoning Environment.

There were two goals in creating this verification approach. The first goal was to reuse the verification already performed on components and design patterns. The second goal was to enable distributed development by establishing the formal requirements of subcomponents that are used to assemble a system architecture. If we are able to establish a system property of interest using the contracts of its components, then we have a means for performing *virtual integration* of components. We can use the contract of each of the components as a specification for suppliers and have a great deal of confidence that if all the suppliers meet the specifications, the integrated system will work properly.

In our composition formulation, we use past-time LTL [2]. This logic supports a uniform formulation of composition obligations that can be used for both *liveness* properties and *safety* properties. For the reasoning framework, we use the LTL operator G (globally) supplemented by the past time operators H (historically) and Z (in the previous instant) [3]. They are defined formally over paths σ and time instants *t* as follows:

$$\sigma, t \models G(f) \;\equiv\; \forall\, (u, t \leq u) : \sigma, u \models f$$

$$\sigma, t \models H(f) \;\equiv\; \forall\, (u, 0 \leq u \leq t) : \sigma, u \models f$$

$$\sigma, t \models Z(f) \;\equiv\; (t = 0) \vee (\sigma, (t\text{-}1) \models f)$$

Verification conditions. Formally, in our framework a component contract is an assume-guarantee pair (A, P), where each element of the pair is a PSL formula. Informally, the meaning of a pair is “if the assumption is true, then the component will ensure that the guarantee is true.” To be precise, we need to require a component to meet its guarantee only if its assumptions have been true up to the current instant. We can state this succinctly as a past-time LTL formula G(H(A) ⇒ P) [1].

Components are organized hierarchically into systems as shown in Fig. 1. We want to be able to compose proofs starting from the leaf components (those whose implementation is specified outside of the architecture model) through several layers of the architecture. Each layer of the architecture is considered to be a system with inputs and outputs and containing a collection of components. A system S can be described by its own contract (A_s, P_s) plus the contracts of its components C_S, so we have $S = (A_S, P_S, C_S)$. Components "communicate" in the sense that their formulas may refer to the same variables. For a given layer, the proof obligation is to demonstrate that the system guarantee P_s is provable given the behavior of its subcomponents C_S and the system assumption A_s.

Our goal is therefore to prove the formula G(H(A_s) ⇒ P_s) given the contracted behavior G(H(A_c) ⇒ P_c) for each component c within the system. It is conceivable that for a given system instance a sufficiently powerful model checker could prove this goal directly from the system and component assumptions. However, we take a more general approach: we establish generic *verification conditions* which together are sufficient to establish the goal formula. Moreover, we provide verification conditions which have the form of safety properties whenever all the assumptions and guarantees do not contain the G or F LTL operators. In such cases, this allows the verification conditions to be proved even by model checkers which are limited to safety properties, such as k-induction model checkers.

Handling cycles in the model. It is often the case that architectural models contain cyclic communication paths among components (e.g., the FGS_L and FGS_R in Fig. 1), and that these components mutually depend on the correctness of one another. Therefore, we need to consider circular reasoning among components. To accomplish this, we use a framework similar to the one from Ken McMillan in [4]. We break these cycles using *induction over time*.

Suppose that we have components A and B that mutually refer to each other's guarantees. When trying to establish the assumptions of A *at time t*, we will assume that the guarantees of B are true *only up to time t*-1. Therefore, at time instant t there is no circularity. To accomplish this reasoning, we define a well-founded order (<) between component contracts. If $C_A < C_B$, then B can refer to A's assumptions and guarantees at the current instant, while A can refer to *B's* assumptions and guarantees only at the previous instant.

Following McMillan, for a contract $c \in C$, we define Θ_c to be the contracts whose assumptions and guarantees are true up to and including time t. We define c^ for a contract c to be (A_c ^ P_c) and C^ to be {c^ | c ∈ C}. Every element in Θ_c must be less than c according to the order <, so $\Theta_c \subseteq C$^. Since we are only considering cycles inside system S, its contracts are handled separately and do not need to be included. Therefore, the system assumptions are taken to hold up through the current time, and the system guarantees are proven separately (as shown below):

[1] We use "Promises" P in the place of G for guarantees for presentation because G is an LTL operator. Also, in the informal presentation, we represent each of A,P as *sets* of formulas. The formulas here are formed by simple conjunction of the elements of the set.

Theorem 1. Let the following be given:

- $S = (A_S, P_S, C_S)$ with assumption A_s, guarantee P_s and component contracts C_S, with a well-founded order < on C
- Sets $\Theta_c \subseteq C\wedge$, such that $q \in \Theta_c$ implies $q<c$
- For all $c \in C$, $\models G(H(A_c) \Rightarrow P_c)$

Then if for all $c \in C$

$$\models G((H(A_s) \wedge Z(H(C\wedge)) \wedge \Theta_c) \Rightarrow A_c)$$

and

$$\models G((H(A_s) \wedge H(C\wedge)) \Rightarrow P_s)$$

then

$$\models G(H(A_s) \Rightarrow P_s).$$

In other words, for a system with n components there are $n+1$ verification conditions: one for each component and one for the system as a whole. The component verification conditions establish that the assumptions of each component are implied by the system level assumptions and the properties of its sibling components. These verification conditions are naturally cyclic, but the cycle is broken using the well-founded ordering < and the one-step delay operator Z. The system level verification condition shows that the system guarantees follow from the system assumptions and the properties of each subcomponent. This is essentially an expansion of the original goal, $\models G(H(A_s) \Rightarrow P_s)$, with additional information obtained from each component.

Proof Sketch. It is possible to prove Theorem 1 directly using induction over time. The idea is, at each step, to go through each component (from largest to smallest based on the < ordering) and show that its assumptions hold in the current step. Then we can use the assumption $\models G(H(A_c) \Rightarrow P_c)$ to show that P_c also holds in the current step. Once we have done this for each component we can use the system level verification condition to show that the system level guarantees hold in the current step. Formally, the proof is by induction over time using the strengthened goal formula

$$\models G(H(A_s) \Rightarrow (H(P_s) \wedge H(C\wedge)))$$

The desired goal formula then follows directly.

Another approach to proving Theorem 1 is to encoding it using McMillan's circular reasoning framework. This is fairly straightforward to do. In fact, the two approaches show many similarities which provides a strong argument for the quality of approach. The details of this equivalence are presented in a companion technical report [6].

6 Flight Control System Example

We have applied our compositional verification approach to the avionics system model. While there is not space to present the entire example, this section provides a summary of the assumptions and guarantees on the flight control system and describes one level of reasoning.

One of the typical requirements levied on a flight control system has to do with transients in the actuator commands. For passenger comfort and safety, a limit is placed on the forces that would be experienced by the passengers during normal operation. For example, the automation should not command a sharp change in the pitch of the aircraft, even in the presence of component failures.

In our system architecture, this property becomes a constraint on the control surface actuator (CSA) output of the system. We would like the commanded pitch to be bounded both in terms of the both the actuator angle and its rate of change. In our notation, we can write these properties as follows:

```
transient_response_1 : assert
   true -> abs(CSA.CSA_Pitch_Delta) < MAX_PITCH_DELTA;
transient_response_2 : assert
   true -> abs(CSA.CSA_Pitch_Delta -
      prev(CSA.CSA_Pitch_Delta, 0.0)) < MAX_PITCH_DELTA_STEP ;
```

The "`true ->`" portion of each property states the property is initially true. The remainder of the first property states that the absolute value of the commanded pitch (CSA_Pitch_Delta) is less than some constant (MAX_PITCH_DELTA). The second property is similar, but states that the difference between the current pitch and the previously commanded pitch is less than some constant (MAX_PITCH_DELTA_STEP).

Similarly, we have system-level assumptions related to independence of failures:

```
active_assumption: assume (FD_L.mds.active or
                           FD_R.mds.active) ;
```

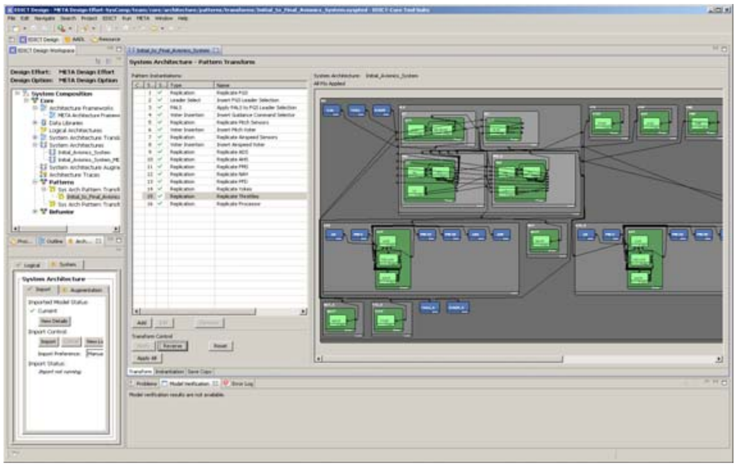
In our model we make assumptions about at least one FGS being active at all times (shown), as well as assumptions about maximum discrepancies between left and right side pitch sensors, and a handful of other assumptions. These assumptions state maximum discrepancies in the pitch inputs in time and between the left and right sides. In order to prove the guarantees for the system, we need to pull in assumptions from the left and right FGSs and the autopilot. In the absence of circularity, the tools automatically compute the dependency order for reasoning: FCI < {FGS_L, FGS_R}, {FGS_L, FGS_R} < AP[2]. For circular dependencies, the user must decide an order (if it is required). In this instance, our proof did not require "same instant" assumptions between FGS_L and FGS_R, so the cycles can be broken and verification conditions produced automatically by our AGREE tool.

The proof relies on the guarantees for the components {FGS_L, FGS_R, AP}, and uses the system level assumptions to discharge the component level assumptions. In addition, the Leader Selection pattern guarantees are brought in as additional *facts* which function as assumptions in the proof. Each set of assumptions and guarantees per component is between ½ page and 1 page of text, and the proof for this layer of the architecture can be discharged in ~5 seconds by the Kind model checker.

[2] The set notation is a shorthand: X < {Y, Z} is the same as X <Y and X < Z.

7 Tool Environment

We have produced a prototype implementation of all the tools described in this paper in a single Eclipse environment, shown in Fig. 4 . They have been designed to work with the open source OSATE AADL tool developed by the Software Engineering Institute.



#	Type	Name
1	Replication	Replicate FGS
2	Leader Select	Insert FGS Leader Selection
3	PALS	Apply PALS to FGS Leader Selection
4	Voter Insertion	Insert Guidance Command Selector
5	Replication	Replicate Pitch Sensors
6	Voter Insertion	Insert Pitch Voter
7	Replication	Replicate Airspeed Sensors
8	Voter Insertion	Insert Airspeed Voter
9	Replication	Replicate ADS
10	Replication	Replicate AHS
11	Replication	Replicate FMS
12	Replication	Replicate NAV
13	Replication	Replicate PFD
14	Replication	Replicate Yokes
15	Replication	Replicate Throttles
16	Replication	Replicate Processor

Fig. 4. Eclipse environment for our translation, pattern instantiation, and verification tools.

The SysML-AADL translator is implemented as an Eclipse plug-in. It provides a convenient way to import an initial block diagram model created in SysML into OSATE for further development.

The pattern instantiation tool is implemented as an extension to the EDICT tool developed by WW Technology Group. EDICT provides a wide variety analysis capabilities for building dependable systems, and now includes the ability to modify the system design through application of the architectural patterns described above.

We have developed two additional Eclipse plugins to implement the compositional verification approach described in this paper (AGREE), and a static analysis tool called Lute for verifying structural properties of AADL models. These tools are available for download through the AADL wiki page at http://wiki.sei.cmu.edu/aadl/index.php/RC_META.

Complex structural assumptions and guarantees can be verified using the Lute checker. While Lute is similar to the to the REAL verification system [10], it provides several enhancements needed for the META project for specifying and checking complex structural properties. A Lute specification is made up of Lute theorems, which are computational checks over the structure of the model. A typical Lute theorem iterates over a select group of components and aggregates information about each before checking a Boolean condition. For example, a Lute theorem may iterate over each process and verify that the maximum deadline for all threads in the process is less than or equal to the process deadline. The Lute code for this theorem is shown below:

```
theorem Process_Deadline_Greater_Or_Equal_Thread_Deadline
  foreach p in Process_Set do
    Thread_Deadlines := {Property(t, “Deadline”)
      for t in Thread_Set | Owner(t) = p};
  check Max(Thread_Deadlines) <= Property(p, “Deadline”);
  end;
```

Since Lute theorems are purely computational, they can be executed without user interaction. Thus it is feasible to re-verify the Lute specification every time a structural change is made to the model. This enables instant feedback during model development.

The AGREE tool uses the custom AADL property set PSL_Properties to add support for compositional reasoning to AADL. The PSL_Properties property set is currently implemented simply as an AADL string applied as follows:

```
property set PSL_Properties is
  Contract: aadlstring applies to (system, process, thread);
  Facts: aadlstring applies to (system, process, thread);
end PSL_Properties;
```

That is, it supports contracts and facts on systems, processes, and threads specified as AADL strings. Verification of AADL models is performed through the translation of the AADL structure and subcomponent assumptions and guarantees into a form suitable for model checking. Currently the KIND model checker is supported, but it would be straightforward to add support for additional model checkers and theorem provers.

In our initial implementation, subcomponents are assumed to operate synchronously with a one-step communication delay between connected subcomponents. This makes the analysis tractable and creates a sound approximation of the behavior of the system. Any error found during verification corresponds to an error in the actual system. The approximation is complete in the case of synchronous systems (e.g. systems using the PALS pattern), and incomplete in the general case. Incompleteness means that the absence of verification errors does not ensure that the system is correct.

8 Conclusion

The work described here was accomplished under the META program which had a period of performance of only 12 months. Consequently, what we have presented here is just a start in what we consider to be a very important and very interesting research area. There is much important work ahead of us.

First, we plan to extend our compositional verification approach to include more complex models of computation. Synchronous computation platforms are found in many avionics systems, but we also need to provide support for multiple execution rates, variable delays, and asynchronous computation.

We have implemented four architectural design patterns to demonstrate the concept, but there are many more that we have encountered. In particular, there is a great deal of work on standard fault tolerance mechanisms with existing verification artifacts that would fit very well into our design pattern scheme.

The technique we have used to embed contracts in AADL models is expedient but semantically shallow. An improved method for annotation of architecture models with formal contracts would allow much better integration with the system design and more robust tooling. A new AADL annex seems the best way to accomplish this. We would also to provide support for some of the features in SysML that are well-suited for capturing dynamic requirements, such as activity and sequence diagrams.

Acknowledgements. This work was sponsored in part by AFRL under contract FA8650-10-C-7081 in the DARPA META program and NSF grant CNS-1035715.

References

[1] Darren D. Cofer, Steven P. Miller, Andrew J. Gacek, Michael W. Whalen, Brian LaValley, Lui Sha, Abdullah Al-Nayeem: Complexity-Reducing Design Patterns for Cyber-Physical Systems. Air Force Research Laboratory Technical Report AFRL-RZ-WP-TR-2011-2098 (2011)

[2] J.A.W. Kamp: Tense Logic and the Theory of Order. Ph.D. Thesis, UCLA (1968)

[3] The NuSMV Toolset Users Manual. Available at: http://nusmv.irst.itc.it/ (2005)

[4] Ken L. McMillan: Circular Compositional Reasoning about Liveness. Technical Report 1999-02, Cadence Berkeley Labs, Berkeley CA (1999)

[5] IEEE Standard for Property Specification Language (PSL). IEEE Std 1850-2005, 2005.

[6] Michael Whalen, Andrew Gacek, and Darren Cofer: Circular Hierarchical Reasoning using Past Time LTL. Technical Report 2011-1, University of Minnesota Software Engineering Center. Available at: http://www.umsec.umn.edu/publications (2011)

[7] The Mathworks Inc. Simulink Product Web Site. Available at: http://www.mathworks.com/products/simulink

[8] Min-Young Nam; Pellizzoni, R.; Lui Sha; Bradford, R.M., ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs, 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, June 2009.

[9] M. Kwiatkowska, G. Norman and D. Parker: Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In Proc. TACAS'02, volume 2280 of Lecture Notes in Computer Science, pages 52-66, Springer.

[10] O. Gilles, J. Hugues, Expressing and Enforcing User-Defined Constraints of AADL Models, Engineering of Complex Computer Systems (ICECCS), pp.337-342 (2010).

[11] Peter Csaba, Olveczky, Artur Boronat, and Jose Meseguer: Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. FMOODS/FORTE 2010, Lecture Notes in Computer Science, Vol. 6117 (2010) 47–62.

[12] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, David Lesens, Virtual Integration of AADL models by a translation into synchronous programs, EMSOFT 2007, ACM (2007).