Composition of Verification Assets for Software Product Lines of Cyber Physical Systems

Ethan T. McGee[1(✉)], Roselane S. Silva[2], and John D. McGregor[1]

[1] School of Computing, Clemson University, Clemson, SC, USA
{etmcgee,johnmc}@clemson.edu

[2] Department of Computer Science, Federal University of Bahia (UFBA), Salvador, BA 40170-110, Brazil
rosesilva@dcc.ufba.br

Abstract. The emerging Internet of Things (IoT) has facilitated an explosion of everyday items now augmented with networking and computational features. Some of these devices are developed using a Software Product Line (SPL) approach in which each device, or product, is instantiated with unique features while reusing a common core. The need to rapidly develop and deploy these systems in order to meet customer demand and reach niche markets first requires shortened development schedules. However, many of these systems perform roles requiring thorough verification, for example, securing homes. In these systems, the detection and correction of errors early in the development life cycle is essential to the success of such projects, with particular emphasis on the requirements and design phases where approximately 70% of faults are introduced. Tools such as the Architecture Analysis & Design Language (AADL) and its verification utilities aid in the development of an assured design for embedded systems. However, while AADL has excellent support for the specification of SPLs, current verification utilities for AADL do not fully support SPLs, particularly SPL models utilizing composition. We introduce an extended version of AGREE, a verification utility for AADL, with support for compositional verification of SPLs.

Keywords: Verification · AADL · AGREE

1 Introduction

Cyber-Physical Systems (CPS) are physical systems that are monitored, controlled, integrated and coordinated by a software layer. These systems bridge the gap between the discrete and continuous worlds [7] and are used in multiple domains: automotive, medicinal and aerospace among others. They also form the backbone of the emerging Internet of Things (IoT). Due to a need of being first to market, some manufacturers of IoT CPS have adopted a Software Product Line (SPL) strategy allowing them to reuse core functionality among products while tailoring the features of each product to the device's intended use. First to market also necessitates shortened development cycles imposing the need for

G. Botterweck and C. Werner (Eds.): ICSR 2017, LNCS 10221, pp. 123–138, 2017.
DOI: 10.1007/978-3-319-56856-0_9

faults to be discovered quickly. Research has shown that approximately 70% of all faults originate in the requirements and design phases of the Software Development Life Cycle; the majority, 80%, of these errors are not caught until later in the development life cycle [2]. The Architecture Analysis & Design Language (AADL), designed for modeling embedded systems, also has shown good success in modeling the intricacies of SPLs [1]. AADL has a strong set of verification tools that allow system designs to be tested for defects, thus allowing more defects to be caught early. However, none of the verification tools for AADL currently available fully support the verification of SPLs.

AADL supports compositional construction of design components, allowing it to natively represent the design of SPLs using design operators, such as substitution, to satisfy the variation needs of the implementation hierarchy within a component. AADL also provides variation mechanisms that support the incremental definition of component variants. The language provides facilities allowing one component to be defined by *extending* another component and permitting the inherited types to be *refined* into more contextually appropriate types, demonstrated with a model in Figs. 1 and 2. This allows the designer to specify, for example, a functional interface which each product of the SPL will implement and have each product inherit the interface rather than re-specify it for each product individually. AADL's primary behavioral verification mechanism, the Assume Guarantee REasoning Environment (AGREE), does not natively support component extension requiring more cumbersome verification conditions for verifying designs incorporating extension than should be necessary. If AGREE fully supported AADL's inheritance mechanisms it could be used to verify complex SPL designs more naturally, and it would enable the reuse of verification assets in SPLs just as AADL interfaces can be reused.

In this paper, we present an extension to the AGREE language that provides inheritance support. This is done via two accomplishments:

- detecting the AADL *extends* keyword (how AADL natively indicates inheritance) and overriding the behavior so that AGREE can utilize the connection, and
- introducing abstraction into the AGREE annex allowing children to override functionality inherited from their parent(s).

The remainder of this paper is structured as follows. In Sect. 2 we provide the background necessary for understanding the remainder of this work. In Sect. 3, we present the method used in modifying AGREE, and we present an extended example in Sect. 4. Finally, related work is over-viewed in Sect. 5.

2 Background

2.1 AADL

AADL is a language for the architectural modeling of embedded software [12]. It is a standard of the Society of Automotive Engineers (SAE) [13] and incorporates many features for the representation of both hardware (i.e. processors, memory, buses) and software (i.e. data, thread, subprograms). AADL supports a model-based architecture design through fine-grained modularity and separation of concerns. It's syntax also includes capabilities for querying the architectural model facilitating verification and validation of the models.

Our extension to AGREE utilizes the extensibility feature of the language. This is represented by the *extends* keyword and is how AADL designates inheritance. An extender receives all of the features, sub-components and connections of the component it extends. The extender is also permitted to *refine* components inherited from the parent. An AADL snippet using *extends* and *refines* is shown in Figs. 1 and 2. However, unlike features, properties and other native AADL elements which can be extended and refined, annexes are not inherited by extenders.

```
system parent
 features
  input: in data port;
  output: out data port;
 annex agree {**
  assume "input greater than 2":
   input > 2;
  guarantee "output input * 2":
   output = input * 2;
 **};
end parent;
```

Fig. 1. AADL parent example

```
system child extends parent
 --inherited from parent
 --features
  --output: out data port;
  input: refined to in data port
   Base_Types::Integer;
 annex agree {**
  assume "input greater than 2":
   input > 2;
  guarantee "output input * 2":
   output = input * 2;
 **};
end child;
```

Fig. 2. AADL child *extends* example

2.2 AGREE

AGREE is a compositional verification tool for AADL based on the widely-used assume-guarantee contract verification method [18]. Designers state their assumptions about input and specify guarantees concerning output provided the assumptions are met. Designers also specify the behavior of a system to ensure that the system can fulfill its guarantees. Analysis work in AGREE is performed by a Satisfiability-Modulo Theorem (SMT) prover that checks the behavior model for contradictions that would prevent the system from fulfilling its guarantees. Any found contradictions are then presented to the user as a case against the system's correctness.

AGREE is an AADL annex that encapsulates the definitions of contracts and specifications. A sample of AGREE's syntax, an assume-guarantee contract, is shown in Fig. 1. Note that AGREE splits the assume-guarantee contracts from their behavior specification. The assume-guarantee contract is placed in the functional interface along with the input / output specifications, and the behavior specification is placed in the implementation. In this way, the multiple implementations common in an SPL can use the same assumptions and guarantees while each has its own behavior specification.

2.3 Software Product Lines

A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment / mission and are developed from a common set of core assets in a prescribed way [22]. SPLs have achieved remarkable benefits including productivity gains, increased agility, increased product quality and mass customization [8].

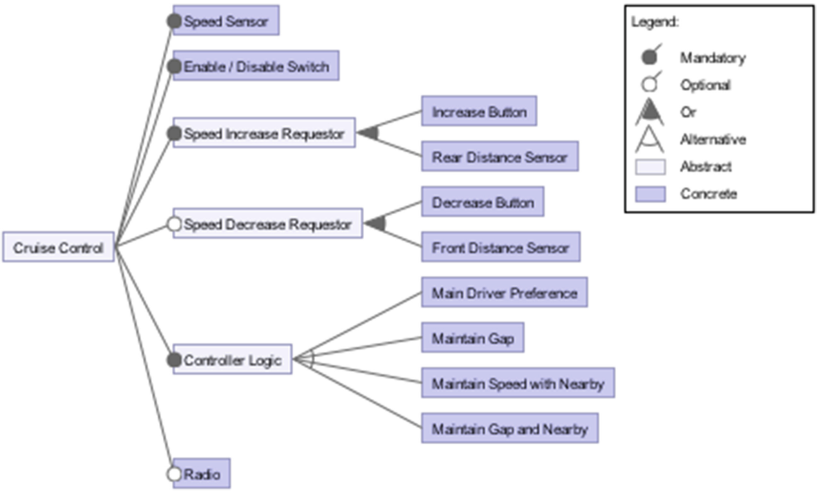

Fig. 3. SPL feature model

SPLs are of particular importance to the IoT, particularly for their cost / time savings and productivity gains. They enable IoT companies to maintain a common core of features which can be reused across several products through customization of the product instantiation. This reuse permits shortened development schedules and also allows companies to maintain a common set of applications, each targeted to a specific audience.

Figure 3 represents an example SPL feature model, a diagram of the configurations each product in the product line can choose. Some features of the cruise control are required, for example, the sensor which determines the current speed of the vehicle, a method of requesting the vehicle accelerate and a method of enabling / disabling the cruise control system. Other features, like the radio to facilitate communication between vehicles, are optional. Each product will make a selection of which features to include and, for the features, which have multiple variations, which variations to use.

3 Method

AGREE is packaged as a plug-in for the Open Source AADL Tool Environment (OSATE) development workbench, which is built on top of Eclipse [21]. AGREE adds several features to OSATE. The first addition is a right-click context menu for the model outline viewer, shown in Fig. 5. This context menu exposes the verification options supported by AGREE and allows the user to select which component(s) he wishes to verify. The second addition is that of an annex which exposes the AGREE language, its parser, and its semantic analyzer as well as the interface to the prover. An overview of the work-flow of the plug-in can be seen in Fig. 4.

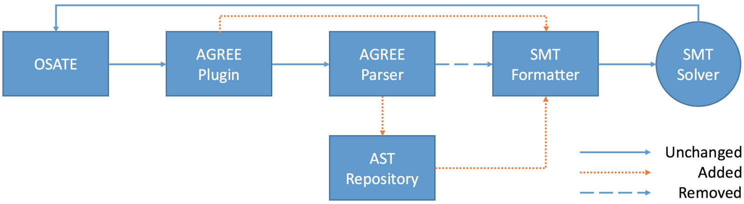

Fig. 4. Workflow for AGREE

The user accesses the context menu for a component and selects a verification task. The architectural description of the component, the AGREE annex contents and any sub-components are then provided to the plug-in. The AGREE contract statements are extracted from the component and parsed into an Abstract Syntax Tree (AST). The AST is provided to a formatter which transforms the AST into the syntax expected by the Satisfiability-Modulo Theorem (SMT) prover. The results of the SMT prover's execution are provided back to OSATE in a displayable format which OSATE renders. A view of the rendered results can be seen in Fig. 5. Note that OSATE displays successfully verified conditions of a component with green checks, and errors are displayed with a red X. Users can right-click the invalidated conditions for more detail.

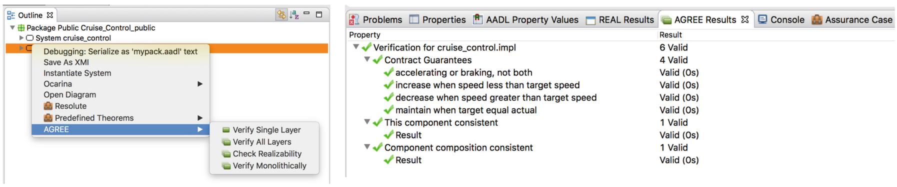

Fig. 5. Context menu & AGREE console

AGREE can be used for both architectural design and verification. When used for design, AGREE contracts are specified at a broad level first, then as the architecture matures, they become increasingly refined. Throughout this paper, we primarily focus on AGREE's verification functionality. Note, however, that our work is applicable to the design functionalities of AGREE as well.

Our extension to AGREE includes modifications to the architecture of the plugin facilitating inheritance support [1]. We also introduce new statements to the language which facilitate inheritance while also providing the ability to disable it for backwards compatibility. We first cover the modifications made to the architecture of the plugin.

In order to facilitate inheritance, we modified the parser of AGREE so that it no longer directly communicated ASTs to the SMT Formatter. We also added a repository which serves two purposes. It first functions as a temporary bank which holds all ASTs of the architectural model. Secondly, it functions as a composer that is capable of stitching together parent and child ASTs into a single, unified AST. The composer functionality is invoked only when a component's AST is requested by the Formatter. These modifications along with the original architecture are visualized in Fig. 4.

From the perspective of the composer, there are three types of statements that an annex can contain. The first are original, or normal, statements. These are statements that are introduced in the current specification and do not exist at any higher level of the inheritance hierarchy. The second are inherited statements, statements that are introduced at a higher level of the inheritance hierarchy which are copied down into the behavior of the child. And finally, override statements are statements which amend the behavior of inherited statements.

When an AST is requested from the Formatter, the inheritance hierarchy of the requested component is gathered, and then a composed AST is generated starting at the highest level of the hierarchy. As the composer moves down each level of the hierarchy, it invokes a merging formula

$$C = (I + O) + N$$

where C is the composed behavior of the current level and all higher levels. I represents inherited behavior, N represents normal behavior and O is the override behavior. As the order of statements in the AGREE annex is important, inherited behavior is always included first, taking care to account for any overrides. Finally, new behavior introduced in the current level is appended. The composed AST is then passed down the inheritance hierarchy until all levels have been evaluated. Once the hierarchy is completely traversed, the final composed behavior is returned to the Formatter.

We now cover the modified / additional statements added to the AGREE language in order to facilitate inheritance. A short overview of the statements that have been added or modified is presented in Table 1. Each statement will be discussed and an example of its use provided.

Guarantee / Assume Statements. The *guarantee* and *assume* statements of the AGREE language are analogous to the pre-condition / post-condition concepts

[1] We will refer to the AGREE language provided in the standard OSATE distribution as "traditional" and our version as "extended".

Table 1. AGREE syntax overview

Keyword	Description
assume	declare that the system expects input to conform to the following statement
do not inherit	explicitly disable inheritance
eq	declare a concrete variable or override an abstract variable
eq abstract	declare an abstract variable
guarantee	declare that output of the system will conform to the following statement
inherit	explicitly state that inheritance from a parent occurs

of other verification tools. With traditional AGREE, the assumptions and guarantees of parent components are not inherited by their children despite the fact that many times the children will use the same inputs, outputs, assumptions and guarantees as their parents. Our extended version of AGREE allows for such inheritance. We also recognize that it is sometimes necessary to tweak the assumptions or guarantees of your parent, particularly if the child introduces new inputs or outputs that the parent does not have.

An example of AGREE's assume and guarantees are shown in Figs. 6 and 7. Also shown in these figures is a demonstration of how our extended version of AGREE permits verification assets to be reused across different components of the model hierarchy as well as how assumptions and guarantees can be overridden by children if necessary. The parent component, introduced in Fig. 6 has two features, a single input and output, and the AGREE annex assumes that the input will be greater than or equal to 0 while guaranteeing that the output will be greater than or equal to 1. The behavior of the parent is simply to take the input value and set the output to the input plus 1. The child, shown in Fig. 7, adds an additional complication by adding a second output. Note that in Fig. 7 all inherited pieces are shown using comments (denoted by a double dash in AGREE and AADL). The guarantees of the child have to be modified or amended to account for this extra output. The override is driven by the descriptor string, or, children who have an assumption or guarantee with a descriptor that matches a parent assumption / guarantee's descriptor will override the parent's matching descriptor.

Eq / Eq Abstract Statements. In traditional AGREE, the *eq* statement allows for the declaration of a single variable. In introducing inheritance, we modified the *eq* statement to either introduce a new variable or to override an existing variable if the variable in the child has the same name as a variable in the parent. We also introduced an *eq abstract* statement that provides a way to define a variable without providing an implementation for that variable. Abstract variables in AGREE are much like abstract variables in Java or C++. They can

```
system parent
  features
    ip: in data port Base_Types::Integer;
    op: out data port Base_Types::Integer;
  annex agree {**
    assume "input req": ip >= 0;
    guarantee "output req": op >= 1;
  **};
end parent;

system implementation parent.impl
  annex agree {**
    assert op = ip + 1;
  **};
end parent.impl;
```

Fig. 6. AADL G / A example

```
system child extends parent
  features
    --ip: in data port Base_Types::Integer;
    --op: out data port Base_Types::Integer;
    op2: out data port Base_Types::Integer;
  annex agree {**
    --assume "input req": ip >= 0;
    guarantee "output req": op >= 1
      and op2 >= 1;
  **};
end child;

system implementation child.impl
  extends parent.impl
  annex agree {**
    --assert op = ip + 1;
    assert op2 = ip + 1;
  **};
end child.impl;
```

Fig. 7. AADL child G / A example

be used in calculations and statements just like any other variable but their implementation is left for children, or extenders, to provide. We also introduce the concept of an abstract implementation, an implementation specification that contains an AGREE annex which introduces or inherits an abstract variable. In order for an implementation specification to be non-abstract, or concrete, it must override and provide an implementation for all inherited abstract variables without introducing any new abstract variables.

An example of eq and eq abstract statements and how they are used in inheritance is shown in Figs. 8 and 9. Once again comment lines (those starting with a double dash) represent components that have been inherited. The parent figure, shown in Fig. 8, has one output, a string representing the type. In the parent figure, the type produced by the component is guaranteed to be null. This is reflected in the parent's implementation as *myType* has been declared abstract and not provided with an implementation. The child figure, shown in Fig. 9, overrides the parent's guarantee and asserts that the component will declare its type as "child". The child, however, does not have a full implementation, only a provision of a definition for the inherited abstract variable. The assert that ties the abstract variable to the output is inherited and does not require respecification.

Inherit / Do Not Inherit Statements. The *inherit* and *do not inherit* statements are unique to our extended implementation of AGREE. The *do not inherit* statement allows inheritance to be explicitly disabled allowing the traditional behavior of the plug-in to be used. This statement was introduced to provide a means of enabling backwards compatibility. When encountered, the composer

```
system parent
  features
   type: out data port Base_Types::String;
  annex agree {**
   guarantee "output req": type = null;
  **};
end parent;

system implementation parent.impl
  annex agree {**
   eq abstract myType: string;
   assert type = myType;
  **};
end parent.impl;
```

Fig. 8. AADL Eq example

```
system child extends parent
  --features
   --type: out data port Base_Types::String;
  annex agree {**
   guarantee "output req": type = "child";
  **};
end child;

system implementation child.impl
 extends parent.impl
  annex agree {**
   eq myType: string = "child";
   --assert type = myType;
  **};
end child.impl;
```

Fig. 9. AADL child Eq example

of the repository component halts and the current results are returned without including any statements from parent annexes. The *inherit* statement is similar to the *do not inherit* statement in that it allows a developer to explicitly state that inheritance does occur. The statement has no effect on the composer, however, it does allow developers to specify which hierarchies use inheritance and which hierarchies do not if a mixed model is being utilized.

Finally, we provide an example where inheritance is controlled using the *inherit* and *do not inherit* statements. This example is shown in Figs. 10 and 11 and can be seen in the implementation's AGREE annexes. Note that the child's annex does not inherit the assert of the parent due to the child specifying that inheritance should not be used. Note, however, that the *do not inherit* statement does not affect extends. The child will still inherit the features of the parent even though the AGREE annex will not inherit any attributes of the parent.

A video providing more detail and an example can be found online at https://goo.gl/VK6NKe. The source of the implementation is available at https://goo.gl/TG9A4r, and an Eclipse / OSATE compatible update site is provided at https://goo.gl/QZhSrv.

4 Example

We now provide an example of a SPL verified using our extended version of AGREE. First, an overview of the architecture and excerpts of AADL are provided for discussion. Also shown are examples of AGREE using the features introduced in our extended version. Second, we demonstrate that the extended version of AGREE is capable of working with models that use several layers of inheritance.

```
system parent
  features
    ip: in data port Base_Types::Integer;
    op: out data port Base_Types::Integer;
  annex agree {**
    assume "input req": ip >= 0;
    guarantee "output req": op >= 1;
  **};
end parent;

system implementation parent.impl
  annex agree {**
    assert op = ip + 1;
  **};
end parent.impl;
```

Fig. 10. AADL inherit example

```
system child extends parent
  features
    --ip: in data port Base_Types::Integer;
    --op: out data port Base_Types::Integer;
    op2: out data port Base_Types::Integer;
  annex agree {**
    do not inherit;
    assume "input req": ip < 1;
    guarantee "output req": op <= 0
      and op2 <= 0;
  **};
end child;

system implementation child.impl
 extends parent.impl
  annex agree {**
    do not inherit;
    assert op = ip - 1;
    assert op2 = ip - 1;
  **};
end child.impl;
```

Fig. 11. AADL child inherit example

4.1 Architecture Overview

The example SCSPL architecture, whose product hierarchy is diagrammed in Fig. 12 and whose feature model is shown in Fig. 3, has three levels. The top-most level is a collection of core assets shared by each of the different types of cruise controls, or products. The middle level includes a standard cruise control and an adaptive cruise control. The standard cruise control is the type common in many vehicles, particularly older vehicles. It uses the "Maintain Driver Preference" variant for the controller logic feature and does not have a radio or speed decrease detector, and its increase requestor feature variant is simply a button. It allows a user to manually set a speed for the car to maintain, and sensors in the engine determine how the throttle needs to be modified in order for the requested speed to be achieved. The adaptive cruise control, found in some vehicles, is the same as the standard cruise control except that it has extra sensors on the front of the vehicle that also feed into the throttle actuator as well as the braking system. If the cruise control is causing the vehicle to approach another vehicle too rapidly, the adaptive cruise control can use the brake actuators to match the speed of the vehicle in front. This product uses the "Rear Distance Sensor" variant and a button for the increase speed requestor feature and the "Front Distance Sensor" and a button for the decrease speed requestor feature. The "Maintain Gap" variant is chosen for the controller feature. Finally, the bottom of the hierarchy contains a collaborative-adaptive cruise control. This cruise control, in addition to front sensors, includes networking capabilities that allow vehicles to communicate amongst one another to determine the safest

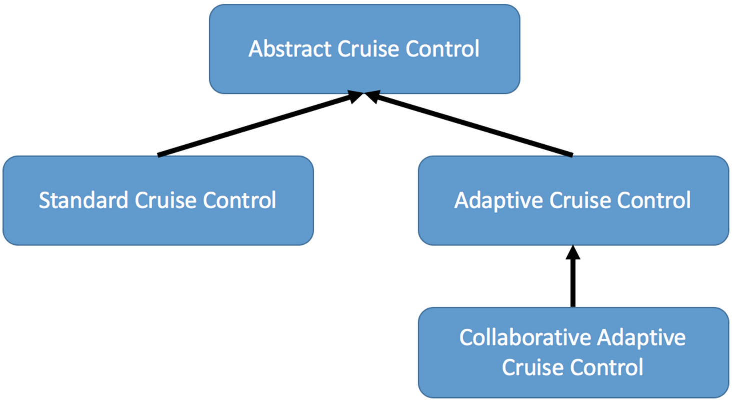

Fig. 12. Example SPL architecture

speed for all vehicles to be traveling considering the location and lane of the vehicle, so the optional radio feature is selected. This cruise control uses the "Maintain Gap and Nearby" variant for the controller logic feature. In addition to other vehicles, collaborative-adaptive cruise controls could communicate with Traffic Management Centers or roadside infrastructure, however, this is outside the scope of this architecture.

4.2 Verifying Multi-layered Architectures

We will now introduce several models which represent parts of the cruise control architecture. These models will be used to demonstrate, using a more extensive example, how the extended version of AGREE facilitates reuse within models utilizing the inheritance features of AGREE.

The first model used is the model of the abstract cruise control, shown in Fig. 13. This represents all of the shared features found in each cruise control present in the SPL of cruise controls. There are 3 inputs and 1 output. The inputs represent whether or not the cruise control is turned on (*enabled*), what the target speed of the cruise control should be (*targetSpeed*) and what the current speed of the vehicle is (*actualSpeed*). Note that many cruise controls will not operate below a minimum speed threshold, and for our purposes, we have set this threshold at 30 miles per hour.

The single output represents the throttle setting for the vehicle. A method of decreasing the speed is not included in the shared model as this is not a shared trait of the cruise controls in our product line. For example, the standard cruise control is not connected to the braking system of the vehicle. It can moderate the speed by letting off of the throttle, allowing the vehicle to slow down, but it cannot stop the vehicle; this task is left up to the driver. The adaptive cruise control, however, is connected to the braking system of the vehicle and it can issue a command to the braking system over the vehicle bus slowing the vehicle.

```
system abstract_cruise_control
  features
    enabled: in data port Base_Types::Boolean;
    targetSpeed: in data port Base_Types::Integer;
    actualSpeed: in data port Base_Types::Integer;
    increaseSpeed: out event data port Base_Types::Boolean;
  annex agree {**
    assume "target speed is greater than lower threshold when enabled":
      enabled => targetSpeed >= 30;
    assume "actual speed is greater than lower threshold when enabled":
      enabled => actualSpeed >= 30;
    guarantee "increase speed only when enabled":
      not increaseSpeed or (enabled and targetSpeed < actualSpeed and increaseSpeed);
  **};
end abstract_cruise_control;

system implementation abstract_cruise_control.impl
  annex agree {**
    eq abstract shouldIncreaseSpeed : bool;
    assert increaseSpeed = shouldIncreaseSpeed;
  **};
end abstract_cruise_control.impl;
```

Fig. 13. Shared core asset model

The AGREE annex of Fig. 13 focuses on the verification of a single property, assuring that the increase speed event fires only when the cruise control system is enabled and the target speed is less than the actual speed. In all other instances, the increase speed event should be disabled. The controls around whether or not the speed should be increased will depend largely on the components used by the instantiated product of the product line, so an abstract variable *shouldIncreaseSpeed* is introduced in the abstract cruise control system's implementation that children will override based on their requirements.

The second model provided is a representation of the adaptive cruise control. Recall that the adaptive cruise control is connected to various other sensors on the vehicle that allow it to maintain both speed and, in the presence of another vehicle, a gap between the vehicles.

The adaptive model is shown in Fig. 14. Note that 3 additional inputs are provided as well as 1 additional output. The additional output represents the connection to the braking system of the vehicle and can be used to slow the vehicle down when necessary. The additional inputs represent the upper limit of the gap between the vehicle and the vehicle in front (*upperGapLimit*) as well as the lower limit on that gap (*lowerGapLimit*). The final input is the current measured gap distance (*gap*).

Notice that the adaptive model has many more assumptions and guarantees than the shared model, including guarantees from the shared model that are overridden. The implementation is also much more detailed, and it provides an implementation for the abstract variable of the shared model (*shouldIncreaseSpeed*). The reason for the extra complexity, of course, is due to the need to factor a gap

```
system adaptive_cruise_control extends abstract_cruise_control
  features
   decreaseSpeed: out event data port Base_Types::Boolean;
   upperGapLimit: in data port Base_Types::Integer;
   lowerGapLimit: in data port Base_Types::Integer;
   gap: in data port Base_Types::Integer;
  annex agree {**
   guarantee "decrease when speed greater than target speed":
    enabled and actualSpeed > targetSpeed => decreaseSpeed;
   assume "upper gap limit is non-negative and non-zero": upperGapLimit > 0;
   assume "lower gap limit is non-negative and non-zero": lowerGapLimit > 0;
   assume "gap is non-negative and non-zero": gap > 0 ;
   assume "gap is above lower limit": lowerGapLimit <= gap;
   guarantee "increase speed when gap reaches lower limit":
    enabled and gap = upperGapLimit => increaseSpeed;
   guarantee "decrease speed when gap reaches lower limit":
    enabled and gap = lowerGapLimit => decreaseSpeed;
   --override
   guarantee "maintain when target equal actual":
    enabled and actualSpeed = targetSpeed and
    lowerGapLimit < gap and gap < upperGapLimit => not increaseSpeed;
  **};
end adaptive_cruise_control;

system implementation adaptive_cruise_control.impl extends abstract_cruise_control.impl
  annex agree {**
   eq shouldIncreaseSpeed: bool =
    if enabled and (actualSpeed < targetSpeed or gap = upperGapLimit) then
     true
    else
     false;
   eq shouldDecreaseSpeed: bool =
    if enabled and (actualSpeed > targetSpeed or gap = lowerGapLimit) then
     true
    else
     false;
   assert decreaseSpeed = shouldDecreaseSpeed;
  **};
end adaptive_cruise_control.impl;
```

Fig. 14. Adaptive cruise control model

calculation into whether or not the increase speed event should be fired as well as constraints on the decrease speed event. However, notice that, other than the over-ridden guarantee and abstract, none of the parent's restrictions or implementation details need to be copied down into the child. This allows that only verification assets unique to the adaptive cruise control are required to be attached to the adaptive cruise control. This reuse increases the maintainability of the model and reduces the workload / cognitive load on those developing the model.

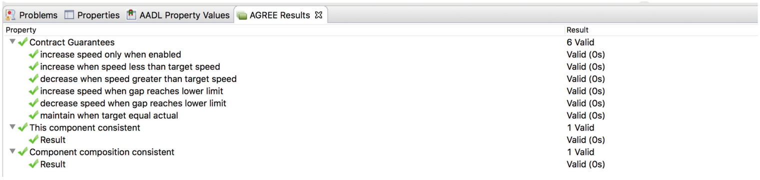

Fig. 15. Adaptive cruise control verification results

Finally, we provide the results of verifying the adaptive cruise control using the extended AGREE implementation in Fig. 15. Note in the figure that the assumptions / guarantees of both models are present despite the assumptions / guarantees of the parent not being specified in the child. This demonstrates that our inheritance mechanism works as expected, and the results of the composition can be validated by a SMT prover.

5 Related Work

Compositionally composed assume-guarantee verification is a popular verification technique, and it has been used successfully in many other ecosystems outside of AADL. Some examples of this are [15,16]. Our work differs from these groups in where verification is applied to the system. We apply compositional verification to the architecture during the design phase of the development life cycle, while these projects apply verification technique later.

Our work is most similar to the work performed by the following groups, particularly [18,26], both of which used AADL. Additional architecture-based techniques exist, such as [14,17,20]. Our work differs from these groups in that we are explicitly focused on allowing the verification assets to be reused in the same manner as SPL assets, exploiting the inheritance features of the AADL language.

6 Conclusion

We have introduced an extension to the AGREE language allowing it to support compositional verification of SPL models that utilize inheritance. Our extended version of AGREE facilitates the re-use of verification assets across multiple levels of inheritance hierarchies present in SPL. It also allows verification assets to incorporate abstraction and refinement into their definitions further simplifying the verification assets to be shared and ensuring they are more maintainable. In future work, we plan to incorporate abstraction in the other statements of traditional AGREE. We also plan to further validate our claims of verification asset reusability by utilizing the extended AGREE module to analyze more complex models, particularly dynamic SPL. The extended version of AGREE will be used to determine the correctness of such models and their appropriateness to an organization's goals.

Acknowledgements. The work of the authors was funded by the National Science Foundation (NSF) grant # 2008912.

References

1. Gonzalez-Huerta, J., Abrahão, S.M., Insfrán, E., Lewis, B.: Automatic derivation of AADL product architectures in software product line development. In: MODELS (2014)
2. Feiler, P., Goodenough, J., Gurfinkel, A., Weinstock, C., Wrage, L.: Four pillars for improving the quality of safety-critical software-reliant systems. DTIC Document (2013)
3. Klein, A., Goodenough, J., McGregor, J., Weinstock, C.: Increasing confidence by strengthening an inference in a single argument leg: An alternative to multi-legged arguments. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (2014)
4. McGee, E.T., McGregor, J.D.: Composition of proof-carrying architectures for cyber-physical systems. In: Proceedings of the 19th International Conference on Software Product Line, pp. 419–426 (2015)
5. Wheeler, D.: http://www.openproofs.org/wiki/Main_Page.OpenProofs (2010)
6. McGee, E.: http://dx.doi.org/10.5281/zenodo.33234 (2015)
7. Rajkumar, R.R., Lee, I., Sha, L., Stankovic, J.: Cyber-physical systems: The next computing revolution. In: Proceedings of the 47th Design Automation Conference, pp. 731–736 (2010)
8. Clements, P., McGregor, J.: Better, faster, cheaper: Pick any three. Bus. Horiz. **55**, 201–208 (2012)
9. Bishop, P., Bloomfield, R., Guerra, S.: The future of goal-based assurance cases. In: Proceedings of the Workshop on Assurance Cases, pp. 390–395 (2004)
10. Gacek, A., Backes, J., Whalen, M., Cofer, D.: AGREE User's Guide (2015). https://github.com/smaccm/smaccm/blob/master/documentation/agree/AGREE%20Users%20Guide.pdf
11. Feiler, P.H., Hansson, J., Niz, D.D., Wrage, L.: System architecture virtual integration: An industrial case study (2009)
12. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis & design language (AADL): An introduction (2006)
13. Feiler, H.P., Lewis, B., Vestal, S.: The SAE architecture analysis and design language (AADL) standard. In: IEEE RTAS Workshop (2003)
14. Goodloe, A.E., Muñoz, C.A.: Compositional verification of a communication protocol for a remotely operated aircraft. Sci. Comput. Program. **78**, 813–827 (2013)
15. Fong, P.W.L., Cameron, R.D.: Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. ACM Trans. Softw. Eng. Methodol. **9**, 379–409 (2000)
16. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Softw. Eng. **30**, 368–402 (2004)
17. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: NASA Formal Methods, pp. 126–140 (2012)
18. Murugesan, A., Whalen, M.W., Rayadurgam, S., Heimdahl, M.P.: Compositional verification of a medical device system. ACM SIGAda Ada Lett. **33**, 51–64 (2013)

19. White, J., Clarke, S., Groba, C., Dougherty, B., Thompson, C., Schmidt, D.C.: R&D challenges and solutions for mobile cyber-physical applications and supporting internet services. J. Internet Serv. Appl. **1**, 45–56 (2010)
20. Hsiung, P., Chen, Y., Lin, Y.: Model checking safety-critical systems using safecharts. IEEE Trans. Comput. **56**, 692–705 (2007)
21. Delange, J.: AADL Tools: Leveraging the Ecosystem. SEI Insights (2016)
22. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co, Inc., Boston (2002)
23. Nair, S., Vara, J.L., Sabetzadeh, M., Briand, L.: An extended systematic literature review on provision of evidence for safety certification. Inf. Softw. Technol. **56**(7), 689–717 (2014)
24. Braga, R.T.V., Junior, O.T., Castelo Branco, K.R., De Oliveira Neris, L., Lee, J.: Adapting a software product line engineering process for certifying safety critical embedded systems. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 352–363. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33678-2_30
25. Feiler, P., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, Boston (2012)
26. Yushtein, Y., Bozzano, M., Cimatti, A., Katoen, J., Nguyen, V., Noll, T., Olive, X., Roveri, M.: System-software co-engineering: Dependability and safety perspective. In: 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, pp. 18–25 (2011)
27. Agosta, G., Barenghi, A., Brandolese, C., Fornaciari, W., Pelosi, G., Delucchi, S., Massa, M., Mongelli, M., Ferrari, E., Napoletani, L., et al.: V2I Cooperation for traffic management with SafeCop. In: 2016 Euromicro Conference on Digital System Design, pp. 621–627 (2016)