Contract-based specification and analysis of AADL models*

Ernesto Posse Juergen Dingel
{eposse,dingel}@cs.queensu.ca

School of Computing – Queen's University
Kingston, Ontario, Canada

Abstract. We describe an approach to the specification, analysis and verification of AADL models using assume/guarantee behavioural contracts specified with the Property Specification Language (PSL). This approach aids the development process by 1) supporting the reuse and replacement of components based on their contracts rather than only their interface or their implementation and thus reducing the need for re-engineering; 2) providing early discovery of behavioural inconsistencies that may pose problems with integration; and 3) allowing an incremental and flexible application of specification and verification instead of requiring an all-or-nothing approach. It also helps improving the product itself by detecting safety and liveness problems via model-checking. We also briefly discuss a prototype plug-in for OSATE supporting an annex language which we call AGCL.

1 Introduction

The development of distributed, real-time embedded systems (DRE) presents multiple challenges born out of their inherent complexity. In order to address the complexity of these systems and their design, component-based and model-driven approaches are often used. Such approaches often rely on modelling and architecture description languages such as the Architecture Analysis and Design Language, AADL [6], which provides the means to describe systems in terms of interacting components and their composition.

Complex patterns of interaction between components pose a challenge to developers, making it difficult to understand how a system behaves and whether it satisfies its requirements and behaves correctly, *e.g.*, satisfying safety and liveness constraints. In order to provide some relief to the developer, automatic formal verification techniques such as model-checking can help to analyze a system's behaviour. Nevertheless, formal verification often faces the so-called *state-explosion problem*, whereby adding a component multiplies the number of states in a system, resulting in an exponential growth in the state-space which provides a challenge to verification techniques and tools.

* This work was financed in part by Edgewater Computer Systems Inc., Ontario Centres of Excellence and Connect Canada.

An approach to deal with the state-explosion problem is the use of *compositional analysis* which leverage the structure of the system. In these techniques, the analysis of a composite system is reduced to the analysis of its parts. A main advantage of such techniques is that if a single component changes, there is no need to reanalyze the whole system, only the portion directly affected. This provides the basis for *incremental analysis*, which aids development by focusing verification only on the components on which the developer is working.

A well-known compositional approach is based on *assume/guarantee contracts* where each component is annotated with a *contract* consisting of an *assumption* specifying how the component expects its environment to behave, and a *guarantee* specifying the behaviour guaranteed by the component if the assumptions hold. Contract-based specification facilitates integration not only by making expectations and assurances explicit, but also by ensuring preservation of correctness when a component with a given contract is replaced by another component whose contract conforms to or refines the first. Contract-based analysis uses the contracts to automatically establish whether a composition of components satisfy the contract of the composite component to which they belong.

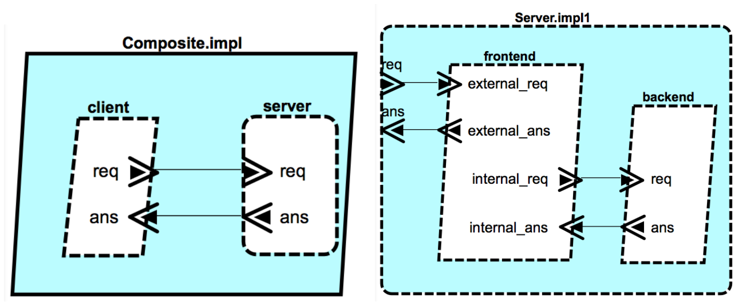
Most approaches to assume/guarantee analysis (*e.g.*, [3]) limit the scope of assumptions to component inputs and guarantees to component outputs. Furthermore, in many approaches the form of a contract is of the form "assuming these inputs, we guarantee these outputs". These are two big limitations. Assumptions and guarantees are supposed to capture *behaviour*, not just individual inputs and outputs. Furthermore, both assumptions and guarantees should describe "conversations" between a component and its environment, with assertions about information flowing both ways. For example, a component may assume that whenever it sends a particular output to its environment, the environment will send back some particular message as input to the component.

In this paper we address these shortcomings by we proposing an AADL annex sub-language for annotating components with assume/guarantee contracts and a prototype verifier that performs compositional analysis. In this sublanguage we use the Property Specification Language, PSL, an IEEE Standard [4] which allows the specification of behaviour combining the expressive power of ω-regular expressions and linear temporal logic (LTL), and in which both assumptions and guarantees can refer to inputs and outputs.

Another shortcoming of many formal approaches to analysis is that they usually require an all-or-nothing commitment on the part of the developer, for example requiring a full, formal account of all components' behaviours. We address this by supporting the notion of *viewpoints.* A viewpoint represents a particular set of requirements distributed across components. The designer may annotate any given component with several contracts. All contracts sharing the same name across different components form a *viewpoint.* For example, the designer can define some safety viewpoints separately from some liveness viewpoints. This allows the developer to add contracts and viewpoints as the design progresses. This notion of viewpoint is simpler than that found in [2] where the developer is required to explicitly use more complex operators to combine contracts.

2 AGCL: a sublanguage for assume/guarantee contracts

Consider the model shown in Figure 1 depicting a system consisting of a client and a server which itself consists of a front-end or mediator, and a back-end. In this common pattern, the client may issue requests via a channel `req` and expects an answer on channel `ans`. The front-end of the server receives these requests, may perform some preprocessing, and delegates the requests to the back-end server via the `internal_req` channel. When the back-end server responds on the `internal_ans` channel, the front-end may do some post-processing, and deliver the final answer to the client.



(a) composite (b) server

Fig. 1. A simple client-server architecture with a mediator process.

To annotate components with contracts we need to declare viewpoints, which is done at the package-level annex library as shown below (using the `viewpoint` keyword). The `enforce` keyword is used to inform the tool which viewpoints should be analyzed.[1]

```
package client_server_mediator
public

annex AGCL {**
    viewpoint normal_operation;
    viewpoint alternative_operation;
    enforce normal_operation;
**};

-- etc.

end client_server_mediator;
```

[1] To keep the presentation of our example simple we show only the annex for each classifier. We also ommit the specification of the top-level process, the client and focus on the server only, and we ommit the thread type declarations with ports which are visible in Figure 1.

2.1 Contracts for atomic components (threads)

Figure 2 shows the backend server. Its annex has a `behaviour` clause describing the behaviour of the actual implementation, and a `contract` clause defining a contract for this component within the `normal_operation` viewpoint. The behaviour states that whenever the server receives a request (an `in` event on the `req` port with some signal `s1`), then it will produce an output on the `ans` port in the next state or cycle. The contract in this case has no assumptions and therefore it is simply `true`. The guarantee is that whenever the backend receives a request, it will eventually produce an answer. In this case, it should be fairly trivial that the beahviour satisfies the contract.

```
thread implementation BackendServer.impl1
annex AGCL {**
    behaviour always (in req:s1 -> next out ans:s2);
    contract normal_operation
        assumption TRUE;
        guarantee always (in req:s1 -> eventually out ans:s2);
    end normal_operation;
**};
end BackendServer.impl1;
```

Fig. 2. Backend server.

Note that the guarantee can talk about both inputs and outputs. The same is true for assumptions. A guarantee represents an obligation on the component, whereas an assumption represents an obligation on its environment. Hence, when a guarantee states an atomic proposition labeled `in`, it is stating the component's obligation to accept or receive an input. When a `in` atomic proposition appears in an assumption, the input direction is stated from the point of view of the component but it actually represents an output obligation from the component's environment to the component. Similarly, an `out` in a guarantee is an obligation for the component to produce output, whereas an `out` in an assumption, while stated from the point of view of the component, actually represents an obligation on the environment to accept or receive input coming from the component.

Figure 3 shows the frontend. Its behaviour clause specifies that whenever an external request arrives (from the client), eventually it will reach a state where it will send a request to the backend (through the `internal_req` port) and from that point onwards, whenever it receives an answer from the backend, it will eventually forward the answer to the client on the `external_ans` port. The contract clause specifies as assumption that whenever it sends a request to the backend server, it will get an answer from it eventually. The guarantee states that whenever it receives an external request from the client, it will eventually send an internal request to the backend, and whenever it gets a response from the backend it will eventually send an answer back to the client. In this case it is less trivial that the behaviour satisfies the contract, but this follows from the formal semantics of PSL.

In general, for threads, a behaviour B satisfies a contract $C = (A, G)$ with assumption A and guarantee G, if the formula $B \wedge A \Rightarrow G$ is valid. Intuitively, the behaviour and the assumptions must be enough to imply the guarantee. A (linear) temporal logic formula (including PSL) is *valid* if it holds in all possible paths for every possible model. In our case, the premise of this implication captures the model: the guarantee will be required to be true only on those models with behaviour B, if the assumption A is true as well. The validity of PSL formulas can be established with a model-checker (see Section 4).

```
thread implementation Frontend.impl1
annex AGCL {**
    behaviour always (in external_req:s1
                -> eventually (out internal_req:s1
                    & always (in internal_ans:s2
                        -> eventually out external_ans:s2)));
    contract normal_operation
        assumption always (out internal_req:s1
                        -> eventually in internal_ans:s2);
        guarantee always (in external_req:s1
                        -> eventually out internal_req:s1)
                & always (in internal_ans:s2
                        -> eventually out external_ans:s2);
    end normal_operation;
**};
end Frontend.impl1;
```

Fig. 3. Server frontend (mediator).

An AGCL annex can contain multiple contracts, which can be verified independently. This allows the developer to add contracts as the design progresses, and define contracts which focus only on particular aspects of interest.

2.2 Contracts for composite components (thread groups)

Figure 4 shows the server combining frontend and backend. In this case, the thread group does not have a behaviour specification, but only a contract. It's contract doesn't make any assumptions, but it states the guarantee that whenever an external request comes from the client, eventually it will answer it.

The problem in this case is the following: if we already know that the subcomponents satisfy their respective contracts, how do we establish if the composition (the `Server.impl1`) satisfies its contract? This can be established as follows: let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts for the two subcomponents K_1 and K_2 of a composite component K with contract $C = (A, G)$. Assuming that K_1 satisfies C_1, and K_2 satisfies C_2, then K satisfies C if the following two PSL formulas are valid:

1. $G' \Rightarrow G$ where $G' \overset{def}{=} G_1 \wedge G_2$, and
2. $A \Rightarrow A'$ where $A' \overset{def}{=} (G_2 \Rightarrow A_1) \wedge (G_1 \Rightarrow A_2)$

Intuitively the first one states that the guarantees of the subcomponents together must imply the guarantee of the composite. The second one states that the assumption of the composite must be enough to ensure that 1) the guarantee of the second must imply the assumption of the first, and 2) the guarantee of the first component implies the assumption of the second. This is because the subcomponents may be connected and information may flow both ways between them, and they are part of each other's environments: the behaviour of K_1's environment is given by K_2's guarantees G_2 together with K's environment given by A. Hence, A and G_2 must imply A_1. Similarly for K_2. To be precise, there is a little processing that needs to be done on the formulas G_i and A_i, namely we need to replace port references ocurring in atomic propositions by connector references so that they refer to the same entity, and we need to flip the direction (in/out) of those atomic propositions in assumptions for the same reason. For composite components with n subcomponents, the formulas are generalized to $G' \overset{def}{=} G_1 \wedge G_2 \wedge \cdots \wedge G_n$ and $A' \overset{def}{=} \wedge_{i=1}^{n}((\wedge_{j \neq i} G_j) \Rightarrow A_i)$ respectively. In other words, the guarantees of all subcomponents must imply the guarantee of the composition, and the assumption of each subcomponent must be implied by the guarantees of all other subcomponents. This later requirement can be relaxed in that it is only needed that the assumption of each subcomponent must be implied by the guarantees of only those subcomponents connected to it.

In our example, K_1 and K_2 are `Backend.impl1` and `Frontend.impl1`, and K is `Server.impl1`. As before, we establish the validity of the formulas above with a model-checker (see Section 4), and in this case they happen to be true.

```
thread group implementation Server.impl1
subcomponents
    backend : thread BackendServer.impl1;
    frontend : thread Frontend.impl1;
connections
    client_req : port req -> frontend.external_req;
    client_ans : port frontend.external_ans -> ans;
    server_req : port frontend.internal_req -> backend.req;
    server_ans : port backend.ans -> frontend.internal_ans;
annex AGCL {**
    contract normal_operation
        assumption TRUE;
        guarantee always (in external_req:s1
                        -> eventually out external_ans:s2);
    end normal_operation;
**};
end Server.impl1;
```

Fig. 4. The server.

Incremental analysis is supported in the following way: if one component changes its behaviour, for example the frontend, we only need to check whether this behaviour satisfies its contract(s). If the result of this analysis is positive, then there is no need to check other components, or the validity of the composite formulas, as the contract has not changed and therefore the validity of formulas 1 and 2 is preserved. If the result of this analysis fails, then the developer needs to either modify the behaviour or the contract for the component in question. If the contract for a component changes then one must re-analyze that component (recursively if it is a composite component) and then re-evaluate the implications $G' \Rightarrow G$ and $A \Rightarrow A'$ as above, but there is no need to re-analyze components which have not changed or whose contract has not changed, as they would not change the validity of these formulas.

2.3 Conformance

Contracts can annotate not only implementations but also types. This opens a set of closely related problems that need be addressed. The first one is this: if we have a component implementation K of type T and K has a contract $C_K = (A_K, G_K)$ and T is annotated with contract $C_T = (A_T, G_T)$, how do we know that C_K conforms to C_T? This can be answered by checking two implications: $G_K \Rightarrow G_T$ and $A_T \Rightarrow A_K$. Note that the implication is covariant on guarantees and contravariant on assumptions. For guarantees, this is because the guarantee of the type must be a guarantee of any of its implementations: the set of possible observable behaviours described in G_K must be a subset if the set of behaviours defined by G_T, otherwise there would be at least one behaviour guaranteed by the implementation which does not conform to what the type prescribes. For assumptions the direction is contravariant because the set of behaviours specified by A_T must be a subset of the set of behaviours specified by A_K. If this wasn't required, there would be at least one environment behaviour acceptable by A_T but not by A_K which would entail that component K would not be able to be placed in some composite components expecting type T.

The other related problems occur when an implementation extends another implementation or a type extends a type and both have contracts in the same viewpoint. These cases can be handled as the above: if K' (or T') has contract $C' = (A', G')$ and it extends K (resp. T) with contract $C = (A, G)$, then conformance can be established by checking the validity of $G' \Rightarrow G$ and $A \Rightarrow A'$.

3 Relation between PSL sequences and AADL behaviours

A key issue in the use of a specification language or temporal logic such as PSL to describe behaviours and contracts of AADL models is the correspondance between the semantics of PSL expressions and the behaviour of the AADL model which they intend to describe. However, there is a fundamental obstacle: the core AADL standard doesn't define a unique way of specifying behaviour. It is up to annexes or external languages to provide the implementation of a component and therefore it is not possible to define a general correspondance, but only consider specific types of implementation. One such possibility is to use the behaviour annex where the implementation is defined as a kind of (hierarchical) state machine. In this paper we do not assume any particular formalism, annex or type of implementation. Nevertheless, if behaviour is specified with the behaviour annex or a similar state-based formalism, we can infer the PSL behaviour specification from such state machine using standard transformations (*e.g.*, automata to regular expression, [7]) and then apply the analysis algorithms as described. Alternatively, we could use the behaviour clause itself to infer an automaton that implements it, using well-known algorithms that can transform such expressions and formulas into automata (*e.g.*, [7,8]).

Another way of relating the PSL specifications with the behaviour of AADL components is to establish a correspondance with the thread semantics defined by the AADL standard ([6] Subsection 5.4).

A PSL expression is evaluated with respect to a path or sequence of states labelled with the atomic propositions which are true in such state. Given a sequence, a PSL expression may *hold strongly*, *hold*, *be pending* or *fail*. The expression holds strongly when it contains no bad states, all future obligations have been met, and the expression holds on all extensions to the sequence. The expression holds (but does not hold strongly) when it contains no bad states, all future obligations have been met, and the expression may or may not hold on any given extension of the path. The expression is pending when it contains no bad states, but future obligations have not been met, and the expression may or may not hold on any given extension of the path. Finally, the expression fails when there is some bad state in the path, future obligations may or may not have been met and the expression will not hold on any extension of the path. Additionally, a PSL expression is evaluated with respect to a *clock context*, a boolean expression that determines in which cycles the expression is to be evaluated. The PSL standard does not specify any particular time granularity or what counts as a cycle or clock tick. It is up to verification tools to decide. The default context is true so that the expression is evaluated at every cycle.

There are several alternative ways to establish a correspondance between these paths and cycles and the states of an AADL thread. One possibility, is to consider a cycle every time the thread is dispatched. This is the natural choice when the thread is periodic. For aperiodic threads it is also possible to consider a cycle when the thread is dispatched, but in this case the dispatch occurs only when an event arrives at a port. For sporadic, timed or hybrid threads, the cycle would occur either by an event or by the specified period. If one adopts such convention, then the designer must be aware that the meaning of the PSL expressions depend on the type of thread. For example, the formula $a \wedge \mathsf{X}\, b$ asserting that a holds in the current cycle and b holds in the next cycle means, for a periodic thread, that a holds at the current time t according to the clock, and b holds at time $t+p$ where p is the thread's period. On the other hand, for an aperiodic thread the formula would mean that when an event arrives to one of the thread's ports, a holds, and b will hold the next time an event arrives.

If we are using the behaviour annex to specify implementations, the choice of associating cycles with dispatches may lead to the traditional interpretation of temporal operators with respect to automata, where "next" does really mean the next state. Since the behaviour annex allows for hierarchical state machines, by "state" we would mean a state in the flattened state machine, with a particular assignment of variables to values.

Another possibility is to treat all kinds of threads in the same way, as periodic threads, *i.e.*, assuming that there is an underlying periodic clock, even for aperiodic threads. In this case, there must be a way for the verification tool to obtain the current state of a thread at any time t of this underlying clock.

Since there are several possibilities, none of which seems to be *a priori* any more fundamental than the others, it is up to the developer to decide which interpretation of PSL expressions is more suitable.

4 An AGCL analysis tool

We have implemented a prototype of the AGCL annex and the analyses outlined in the previous sections as a plugin for OSATE. The tool allows the user to apply the analyses outlined in this paper, providing results sorted by either viewpoint or by component. When the result of a particular analysis fails, a counterexample is generated by the model checker. Our plug-in uses the NuSMV model-checker to check the validity of the formulas in question, but the underlying architecture can easily be extended to support other model-checkers.

A model-checker receives as input a model and a specification (temporal logic formula) and decides whether the model satisfies the formula or not. A model-checker can be used to check validity by checking the formula against a *universal model* for the formula, this is, a model that contains all possible states and transitions about which the formula could talk. For example, if a formula contains three atomic propositions, the universal model has three boolean variables and therefore eight states, all of which are initial, and all possible transitions between them. Such universal model contains every possible model of the formula embedded in it, and therefore every possible path. A linear temporal formula is valid if it holds in every path of every model, hence, it is valid if it holds in every path of the universal model. On the other hand, if there is at least one path in the universal model for which the formula doesn't hold, then there exists at least one model for which the formula doesn't hold and therefore the formula is not valid.

In terms of complexity, dealing with universal models might appear untractable, but the size of such models depends only on the size of the formulas (the number of atomic propositions) and not on the size of the state space of the components themselves. This observation combined with the fact that contracts don't need to describe all aspects of behaviour, and can be specified in separate viewpoints and analyzed independently makes the technique feasable.

5 Final remarks

We have sketeched an approach to specify and verify assume/guarantee contracts for AADL components and briefly discussed the kinds of analyses that can be performed and discussed our prototype implementing these. Given the space limitations we are unable to provide here the actual algorithms and their proof of correctness, but these are available in detail as a technical report [5]. The theory behind this work is based on [1] which developed a generic theory of contract-based reasoning applicable to a wide range of specification formalisms. In our technical report we extended and specialized that theory to PSL, showing in particular that when using PSL we can compose contracts, the basis for the compositional analysis. Our approach differs from other compositional techniques such as [3] in that we do not restrict assumptions to inputs and guarantees to outputs. Furthermore, with viewpoints, we make it possible to divide requirements into sets of smaller contracts, providing the developer with flexibility as well as making automatic verification more feasible. While this work is preliminary and we have yet to test the plugin on large-scale models, we believe our early results show promise, and contract-based analysis can provide a fundamental support to the development process of DRE systems.

References

1. S. S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from specifications to contracts in component-based design. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2012.
2. A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *LNCS*, pages 200–225. Springer, 2007.
3. D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2012.
4. IEEE Computer Society. IEEE Standard for Property Specification Language (PSL). IEEE Standard 1850$^{\mathrm{TM}}$-2010, June 2012.
5. E. Posse. Contract-based compositional analysis for reactive systems in RTEdge$^{\mathrm{TM}}$, an **AADL**-based language. Tech. Rep. 2013-607, School of Computing – Queen's University, August 2013. http://research.cs.queensu.ca/TechReports/Reports/2013-607.pdf.
6. SAE International. Architecture Analysis & Design Language (AADL). SAE Standard AS5506b, 10 September 2012.
7. M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1997.
8. P. Wolper. The Tableau Method for Temporal Logic: An Overview. *Logique et Analyse*, 28(110–111):119–136, June–September 1985.