Slicing AADL Specifications for Model Checking*

Maximilian Odenbrett Viet Yen Nguyen Thomas Noll

Software Modeling and Verification Group
RWTH Aachen University, Germany

Abstract

To combat the state-space explosion problem in model checking larger systems, abstraction techniques can be employed. Here, methods that operate on the system specification before constructing its state space are preferable to those that try to minimize the resulting transition system as they generally reduce peak memory requirements.

We sketch a slicing algorithm for system specifications written in (a variant of) the Architecture Analysis and Design Language (AADL). Given a specification and a property to be verified, it automatically removes those parts of the specification that are irrelevant for model checking the property, thus reducing the size of the corresponding transition system. The applicability and effectiveness of our approach is demonstrated by analyzing the state-space reduction for an example, employing a translator from AADL to Promela, the input language of the SPIN model checker.

1 The Specification Language

The work that is described in this paper emanates from the European Space Agency COMPASS Project[1] (Correctness, Modeling, and Performance of Aerospace Systems). Within this project, a specification language entitled SLIM (System-Level Integrated Modeling Language) is developed which is inspired by AADL and thus follows the component-based paradigm. Each component is given by its type and its implementation. The *component type* describes the interface features: in and out data and event *ports* for exchanging data (instantaneous) and event messages (synchronously) with other components. The behavior is defined in the *component implementation* by *transitions* between *modes*, like in a finite automaton. Transitions can have an event port associated to them as a *trigger*. A transition with an out event port as trigger is enabled only if in at least one other component a transition with a corresponding in event port as trigger can synchronously be taken and thereby react to the trigger (and correspondingly the other way round). Furthermore, transitions can be taken only when their *guard* expression evaluates to true. Transitions can be equipped with *effects*, i.e., a list of assignments to data elements. Within a component implementation *data subcomponents*, comparable to local variables, can be declared. Together with in and out data ports we refer to them as *data elements*. All of them are typed and can have a *default value* which is used as long as it is not overwritten. The availability of each data subcomponent can be restricted with respect to modes of its supercomponent. In other than these modes the data subcomponent may not be used and on (re)activation it will be reset to its default value.

In addition to data subcomponents, components can be composed of other *non-data subcomponents*, possibly using multiple instances of the same component implementation. In the resulting nested component hierarchy, components can be connected to their direct subcomponents, to their neighbor components in the same supercomponent and to their direct supercomponent by *data flows* and *event port connections* between their ports. The connections can again be mode dependent. If a data flow becomes deactivated then its target port is reset to its default value. Fan-out is always possible whereas fan-in is not allowed for data flows and must be used carefully with event port connections. Cyclic dependencies are disallowed, too. A component can send an in event to or receive an out event from any of its subcomponents directly by using *subcomponent.eventport* as transition trigger.

*Partially funded by ESA/ESTEC under Contract 21171/07/NL/JD

[1] `http://compass.informatik.rwth-aachen.de/`

Listing 1 gives an example SLIM specification modeling an adder component provided with random input $x, y \in [0, 30]$. We refer to [2, 3] for a more detailed description of the language including a discussion of the similarities and extensions with respect to AADL. In particular, [2] presents a formal semantics for all language constructs, based on networks of event-data automata (NEDA).

```
system Main
end Main;

system implementation Main.Impl
  subcomponents
    random1: system Random.Impl
                    accesses aBus;
    random2: system Random.Impl
                    accesses aBus;
    adder: system IntAdder.Impl
                  accesses aBus;
    aBus: bus Bus.Impl;
  flows
    adder.x := random1.value;
    adder.y := random2.value;
  modes
    pick: initial mode;
  transitions
    pick -[random1.update]-> pick;
    pick -[random2.update]-> pick;
end Main.Impl;

bus Bus
end Bus;
bus implementation Bus.Impl
end Bus.Impl;
```

```
system IntAdder
  features
    x: in data port int;
    y: in data port int;
    sum: out data port int;
end IntAdder;

system implementation IntAdder.Impl
  flows
    sum := x + y;
end IntAdder.Impl;

system Random
  features
    value: out data port int default 2;
    update: in event port;
end RandomIntValue;

system implementation Random.Impl
  modes
    loop: initial mode;
  transitions
    loop -[update then value := 0]-> loop;
    ...
    loop -[update then value := 30]-> loop;
end RandomIntValue.Impl;
```

Listing 1: Integer Adder in SLIM

2 Slicing

The term "slicing" has been coined by Weiser [9], initially for sequential programs, and the approach was extended later on in several ways by many authors (cf. [8]). Directly related to our work is the extension of slicing to concurrent programs by Cheng [4] and, most important, the application of slicing to software model checking including formal notions of correctness by Hatcliff, Dwyer and Zheng [5].

The principal idea of slicing is to remove all parts of a program, typically variables and statements, that do not influence the behavior of interest, typically the values of some variables at some statements, described by a slicing criterion. To determine which parts are relevant, the transitive backward closure of the slicing criterion along different kinds of dependences, typically data and control dependences, is computed. However, finding a minimal sliced program is in general unsolvable since the halting problem can be reduced to it (cf. [9]).

2.1 Slicing of SLIM Specifications

Given a specification S and a CTL* property φ (without next operator), slicing should yield a smaller specification S^{φ}_{sliced} that is equivalent to S with respect to φ, i.e., $S \models \varphi$ iff $S^{\varphi}_{sliced} \models \varphi$ (cf. [5]). Consequently, comparable to a slicing criterion, the property defines the initially interesting parts that must not be sliced away: data elements and modes used in φ (events are not allowed in our properties but could

be added). Subsequently, the closure of the set of interesting parts, i.e., all other aspects that have an (indirect) influence on them and thus on the property, is calculated in a fixpoint iteration. Obviously this iteration always terminates but in the worst case all parts of the specification become interesting.

In the following three paragraphs we describe in more detail the closure rules for adding data elements, events and modes to the set of interesting parts before the actual slicing algorithm is presented in pseudo-code.

Identifying Interesting Data Elements Like with data flow dependence for classic program slicing, all data elements used to calculate a new value for an interesting data element are interesting, too. Here, this affects the right hand sides of assignments to an interesting data element, either in transition effects or by data flows. Furthermore, comparable to control flow dependence, all data elements used in guards on interesting transitions (see below) must be kept in the sliced specification as the evaluation of the guard at runtime determines whether the transition can indeed be taken.

Identifying Interesting Events The main difference of SLIM specifications compared to sequential programs is that the components can synchronously communicate by sending and receiving events. Comparable to synchronization and communication dependences (cf. [4]), all events used as triggers on interesting transitions are important. As events can be forwarded by event port connections all events connected to an interesting event in any direction are interesting as well.

Identifying Interesting Modes Similarly to the program location in classical slicing, our algorithm does *not* treat the mode information as a data element which is either interesting or not but tries to eliminate uninteresting modes. The difficulty is that the questions whether a mode, a data element or an event is interesting are related to each other since all those elements can be combined in the transition relation: On the one hand, transitions are (partially) interesting when they change an interesting data element, have an interesting trigger or their source or target mode is interesting. On the other hand, triggers, guards, and source modes of those transitions are interesting. However, transitions themselves are not considered as elements of interest in the fixpoint iteration. Instead, modes are made interesting and with them implicitly all incoming and outgoing transitions. More concretely, besides the modes used in the property the following modes are interesting as well:

- Source modes of transitions changing an interesting data element. This obviously applies to transitions with assignments to interesting data elements in their effects but also to transitions reactivating an interesting data element, i.e., it is active in the target mode but not in the source mode, since it will be reset to its default value.
- All modes in which a data flow to an interesting data element or an event port connection to/from an interesting event is active. This guarantees that all transitions that deactivate a data flow to an interesting data element and thus reset it to its default value are included in the sliced specification.
- Source modes of transitions with interesting events as triggers because of their relevance for synchronous event communication.

Moreover, the reachability of interesting modes from the initial mode matters. Thus, every predecessor of an interesting mode, that is, the source modes of transitions to interesting target modes, is interesting as well.

Finally, for liveness properties it is additionally necessary to preserve the possibility of divergence since no fairness assumptions are made. For example, whether a component is guaranteed to reach a certain mode can depend on the fact whether another component can loop ad infinitum. To handle this, all modes on “syntactical cycles”, i.e., cycles of the transition relation without considering triggers and guards, are interesting as well. For safety properties this can be omitted.

2.2 The Slicing Algorithm

For the pseudo-code description given in Listing 2 we use the following notations: *Dat*, *Evt* and *Mod* are the sets of data elements, events and modes occurring in the specification, respectively. The relation *Trn* contains transitions of the form $m \xrightarrow{e,g,f} m'$ with source and target mode $m, m' \in Mod$, trigger $e \in Evt$, guard expression g over data elements and f a list of assignments. Data flows $d := a$ instantly assigning the value of an expression a to a data element $d \in Dat$ are collected in the set *Flw*. Finally, *Con* contains connections $e \rightsquigarrow e'$ between event ports $e, e' \in Evt$. On this basis, the algorithm can compute the sets of interesting data elements (D), events (E) and modes (M).

Note that, in analogy to distinguishing calling environments of procedures, slicing is done for component instances and not for their implementation. This is more effective since different instances of the same implementation might be sliced differently. Therefore, we have to distinguish identical parts of different component instances. For example, the set *Dat* for the specification in Listing 1 does not simply contain the name `Random.value` but `random1.value` and `random2.value` to differentiate between the different instances of the `Random` component.

/* Initialization */
$D := \{d \in Dat \mid d \text{ occurs in } \varphi\}$;
$E := \emptyset$;
$M := \{m \in Mod \mid m \text{ occurs in } \varphi\}$;
/* Fixpoint Iteration */
repeat
  /* Transitions that update/reactivate interesting data elements or have interesting triggers */
  for all $m \xrightarrow{e,g,f} m' \in Trn$ with $\exists d \in D : f$ updates d or $\exists d \in D : d$ inactive in m but active in m' or $e \in E$ **do**
    $M := M \cup \{m\}$;
  /* Transitions from/to interesting modes */
  for all $m \xrightarrow{e,g,f} m' \in Trn$ with $m \in M$ or $m' \in M$ **do**
    $D := D \cup \{d \in Dat \mid g \text{ reads } d\}$
      $\cup \{d \in Dat \mid f \text{ updates some } d' \in D \text{ reading } d\}$;
    $E := E \cup \{e\}$;
    $M := M \cup \{m\}$;
  /* Data flows to interesting data ports */
  for all $d := a \in Flw$ with $d \in D$ **do**
    $D := D \cup \{d' \in Dat \mid a \text{ reads } d'\}$;
    $M := M \cup \{m \in Mod \mid d := a \text{ active in } m\}$;
  /* Connections involving interesting event ports */
  for all $e \rightsquigarrow e' \in Con$ with $e \in E$ or $e' \in E$ **do**
    $E := E \cup \{e, e'\}$;
    $M := M \cup \{m \in Mod \mid e \rightsquigarrow e' \text{ active in } m\}$;
until nothing changes;

Listing 2: The Slicing Algorithm

2.3 The Sliced Specification

After calculating the fixpoint, the sliced specification S^{φ}_{sliced} can be generated. Essentially, it contains only the interesting data elements (D), events (E) and modes (M) plus data flows and connections to them, i.e., $d := a \in Flw$ with $d \in D$ and $e \rightsquigarrow e' \in Con$ where $e \in E$, respectively. Their default values and the lists of modes in which they are active stay the same. The sliced transition relation contains all transitions $m \xrightarrow{e,g,f} m' \in Trn$ leaving an interesting mode $m \in M$ with slight modifications: If the target mode is not interesting ($m' \notin M$), it is replaced by a "sink mode" $m_o \notin Mod$ which is added to every component that had uninteresting modes. For each data subcomponent, this sink mode is also added to the list of modes in which the component is active. Furthermore, only those transition effects $d := a$ in f are retained that assign to an interesting data element, i.e., $d \in D$. Finally, all "empty" components, i.e., those that neither have interesting data elements, interesting modes nor non-empty subcomponents, are completely removed in a bottom-up manner.

The resulting specification S^{φ}_{sliced} is again a valid SLIM specification. In particular, every object referenced in it is indeed declared as it was included in the fixpoint iteration, e.g., the data elements used in the guards of interesting transitions. Beyond that, sink modes indeed do not need outgoing transitions as it is impossible to change an interesting data element, to send or receive an interesting event or to reach an interesting mode as soon as an uninteresting mode has been entered by the original component.

3 Results and Conclusions

For model checking SLIM specifications we developed a translator [7] to Promela, the input language of SPIN [6]: Every component instance is transformed to a process whose program labels reflect the modes. Data elements are stored in global variables and communication is implemented using channels. Due to dependencies introduced by translation details, SPIN's slicing algorithm could not effectively reduce the resulting Promela code.

Comparing the model checking results of sliced and unsliced specifications served as a first sanity check for our algorithm while a general correctness proof based on the formal semantics of SLIM is to be developed. The idea is to show that the corresponding transition systems are related via a divergence-sensitive stuttering bisimulation, which is known to preserve the validity of CTL* properties without next [1, p. 560]. Furthermore, the differences in resource demands demonstrate the effectiveness of our approach, e.g., for the introductory example from Listing 1 as shown in the following table:

Specification	Mem/State (bytes)	#States	Memory (MBs)	Time (seconds)
Unsliced (identical for $\varphi_1, \ldots, \varphi_3$)	136	1,676,026	272	6.0 - 7.3
Sliced for $\varphi_1 \equiv \Box\,(0 \leq \mathit{adder.sum} \leq 60)$	116	1,437,691	211	5.4
Sliced for $\varphi_2 \equiv \Box\left(\bigwedge_{\beta=1}^{2} 0 \leq \mathit{random}\beta.\mathit{value} \leq 30\right)$	84	553,553	84	1.4
Sliced for $\varphi_3 \equiv \Box\,(0 \leq \mathit{random}1.\mathit{value} \leq 30)$	76	9,379	33	0.1

All three properties are valid invariants and thus require a full state space search. The difference is in the resulting set of interesting data elements: While for φ_1 every data port is needed, for φ_2 the whole `adder` component can be removed and for φ_3 only `random1.value` is interesting. The removal of the empty bus component accounts for the reduction from the unsliced specification to the one sliced for φ_1.

We conclude that our slicing algorithm can considerably reduce the state space, especially when whole components can be removed. We end with the remark that beyond the scope of this paper the algorithm has been extended to more involved language constructs (such as de- and reactivation of non-data subcomponents or hybridity) and that a refining distinction between weak and strong interesting data elements was made.

References

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Codesign of Dependable Systems: A Component-Based Modelling Language. In *7th Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 121–130. IEEE CS Press, 2009.

[3] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *28th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2009)*, volume 5775, pages 173–186. Springer, 2009.

[4] J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In P. Fritzson, editor, *AADEBUG*, volume 749 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 1993.

[5] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[6] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[7] M. Odenbrett. Explicit-State Model Checking of an Architectural Design Language using SPIN. Master's thesis, RWTH Aachen University, Germany, Mar. 2010. `http://www-i2.informatik.rwth-aachen.de/dl/noll/theses/odenbrett.pdf`.

[8] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[9] M. Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.