Regression Verification of AADL Models through Slicing of System Dependence Graphs

Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, Kaj Hänninen
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
{andreas.johnsen,kristina.lundqvist,paul.pettersson,kaj.hanninen}@mdh.se

ABSTRACT

Design artifacts of embedded systems are subjected to a number of modifications during the development process. Verified artifacts that subsequently are modified must necessarily be re-verified to ensure that no faults have been introduced in response to the modification. We collectively call this type of verification as regression verification. In this paper, we contribute with a technique for selective regression verification of embedded systems modeled in the Architecture Analysis and Design Language (AADL). The technique can be used with any AADL-based verification technique to efficiently perform regression verification by only selecting verification sequences that cover parts that are affected by the modification for re-execution. This allows for the avoidance of unnecessary re-verification, and thereby unnecessary costs. The selection is based on the concept of specification slicing through system dependence graphs (SDGs) such that the effect of a modification can be identified.

Keywords

software architectures, AADL, regression verification, specification slicing, system dependence graph

1. INTRODUCTION

Software verification of embedded systems consumes a majority of the development cost [6]. Numerous research efforts have been devoted to the development of more efficient regression testing techniques as studies show that regression testing consumes up to one-third of the total development cost of a software system [5]. Although the efficiency of regression testing is highly important, empirical studies show that the majority of development faults are introduced by incorrect, incomplete, and inconsistent specifications and models (we use the terms "specification" and "model" interchangeably) [10]. Such artifacts are, in addition to a significant source of fault introduction, also often subjected to a large number of modifications. Hence, they are also subjected to regression verification activities in form of reviewing, inspection, simulation, model-checking, etc., which efficiency may be even more important than the efficiency of regression testing.

The techniques used to perform regression verification of specifications and models do seldom identify the effect the modification has on the artifact under analysis. A modification does not only affect the behavior of the part that explicitly has been modified, but also the behavior of any other part that is dependent on it. By contrast, a modification does typically not affect the complete artifact. As the effect of a modification is not analyzed, there is little understanding of which parts of the artifact that, directly or indirectly, are affected by the modification and must be re-verified. This type of problem is essential for selective (efficient) regression testing [5] and, logically, for selective regression verification of specifications and models as well. A rerun of all already existing and still valid verification sequences, in addition to possibly newly created sequences to cover possibly added parts, is in this case necessary to ensure that no new faults follow from a modification. A rerun-all approach may be significantly more time-consuming and costly, depending on the extent of unnecessary exercised parts, compared to a selective approach where only the necessary verification sequences are executed, i.e., where only parts that are affected by the modification are exercised.

The contribution of this paper is a technique for selective regression verification of embedded systems modeled in the Architecture Analysis and Design Language (AADL) [2] – an overview of AADL can be found in [8]. The technique uses the concepts of *specification slicing* [9] and *system dependence graphs* [7] to identify the parts of a modified AADL model that are directly or indirectly affected by the modification and must be covered by verification sequences in the regression verification process. It can therefor be used with any AADL-based verification technique to efficiently perform regression verification.

The approach originates from the regression testing technique proposed by Bates and Horwitz in [3] who used *program slicing* [14] algorithms on so called *program dependence graphs* [12] to determine which parts of a software program that are affected by a modification and need to be retested. Both specification slicing and system dependence graphs, which backgrounds are presented in Section 2, are extensions of these concepts and can be used for the same purpose as of the approach proposed by Bates and Horwitz. The concept of program slicing is to remove statements, instructions, and variables that do not have an effect on and are not af-

QoSA'14, June 30–July 4, 2014, Marcq-en-Baroeul, France

http://dx.doi.org/10.1145/2602576.2602589.

fected by the value of a variable at some program point (statement or instruction) referred to as the *slicing criterion*. Causal relationships to the value of a variable at some point are determined by the control and data dependencies of the system. These are commonly expressed through a directed graph [13], such as a program dependence graph or, the more expressive, system dependence graph. The parts that have causal relationships to the value of the variable are thus determined by computing the transitive closure of the dependence graph on the slicing criterion. The parts that do not have an effect on the value of the variable are removed by computing the backward (with respect to the directions of the arrows of the graph) transitive closure, known as backward-slicing. The parts that are not affected by the point are removed by computing the forward transitive closure, known as forward-slicing [13].

The technique we propose for selective regression verification comprises four main steps. The first step is to use the algorithms and rules presented in Section 4 to generate the system dependence graphs of an AADL model and its modified version. These algorithms and rules are defined upon an abstract syntax of AADL, presented in Section 3, that has been tailored for the purpose of slicing. The two graphs are then compared according to the criteria presented in Section 5 to precisely identify the modification in the graph of the modified AADL model. With the modification constituting the slicing criterion, backward- and forward-slicing techniques presented in Section 5 are used to identify the elements of the model that have an effect on and are affected by the modification. These must be covered by verification sequences in the regression verification process. Note that the behavior of elements included in the backward-slice but not in the forward-slice is not dependent on the slicing criterion and does not need to be re-verified. However, they are necessary to be executed in the regression verification process (for full coverage) since the behavior of the modified part is dependent on those elements. A running example of applying the technique is initiated by an AADL model presented in Section 3.

The proposed slicing technique is also a contribution to the AADL community as it can be used for comprehension, analysis, verification and validation, maintenance, vectorization and parallelization, integration, removal of unreachable/dead software, worst case execution time analysis, compilation and code generation, reuse, etc., as most other forward- and backward-capable slicing algorithms [13, 7]. There exist to our knowledge no such contribution. In Section 6, we elaborate on the limitations of our work and possibilities for future work and present some concluding remarks.

2. BACKGROUND

The idea of slicing specifications was first introduced by Oda and Araki [11]. The idea is based on the concept of *program slicing* through control and data flow analysis, originally defined by Weiser [14]. Ottenstein and Ottenstein [12] later showed how program slicing algorithms could be defined in terms of operations on so called *program dependence graphs* (PDG) – a method also used for effectively slicing specifications [9]. A PDG is a directed graph of different types of vertices (nodes) and edges (arcs), where vertices represent the statements and predicate expressions of a single monolithic program, and where edges represent control and data dependencies among those vertices. Each PDG consists of a distinguished *entry* vertex representing the entry into the program. Essentially, a PDG is the union of a *control dependence graph* (CDG), where edges describe the control conditions on which the execution of vertices depends, and a *flow dependence graph* (FDG), where edges describe the data variables on which the operations of vertices depend [12]. Both graphs can be generated by analyzing the control flow graph (CFG) [1] of a program.

To be able to perform program slicing in the more general case where a program consists of multiple procedures, Horwitz et al. [7] introduced the so called *system dependence graph* (SDG). SDGs extend the expressiveness of PDGs such that procedure calls and parameter passing (by value) can be integrated.

3. PRELIMINARIES

In this section we present an example of an AADL model and an abstract syntax for AADL. Rules and algorithms for slicing AADL models through SDGs are then defined upon the abstract syntax in Section 4 and Section 5 while they are applied to the AADL example.

The AADL example, shown in Table 1 and Table 2, comprises an embedded system partly consisting of a process component *process.impl* that reads data produced by a dual modular redundant sensor and presents it to the operator through a display (sensors and display are not shown). *process.impl* has two periodically dispatched thread subcomponents, *thread_A* and *thread_B*, which are instances of *thread_1.impl* and *thread_2.impl* respectively. These threads provide the functionality together with three static (shared) data subcomponents, *sensorData_1*, *sensorData_2*, and *displayData_1*, whereby interactions with the sensors and the display are performed. The function of *thread_1.impl*, as described by its behavior specification, is simply to read the sensor data and to send it to *thread_2.impl* through the connected ports. The function of *thread_2.impl* is to compare the two received values and, if they are unequal, display the mean value or, if they are equal, display the value.

The abstract syntax for AADL is defined in terms of a tuple including constructs that determine the control and data dependencies of an AADL model. Let $part_1.part_2$ denote that $part_2$ is a set, sequence, or element of the set, sequence, or element $part_1$. An AADL model is a tuple:

$$\mathcal{AADLMDL} = \langle \mathcal{COMP}, \mathcal{THR}, \mathcal{DATA}, \mathcal{SUB}, \mathcal{C}, \mathcal{CALL} \rangle$$

$\mathcal{COMP} = \{comp_1, comp_2, \ldots, comp_n\}$ denotes the set of software components in the architecture, where $\mathcal{THR}$ denotes the subset of *thread components*, $\mathcal{DATA}$ denotes the subset of *data components*, and $\mathcal{SUB}$ denotes the subset of *subprogram components*. Let *thr*, *data*, and *sub* range over $\mathcal{THR}$, $\mathcal{DATA}$, and $\mathcal{SUB}$ respectively.

A thread $thr = \langle \mathcal{DATA\_S}, \mathcal{SUB\_S}, \mathcal{DP}, \mathcal{EP}, \mathcal{EDP}, \mathcal{DA}, \mathcal{SA}, \mathcal{MSM}, \mathcal{BM} \rangle$ has a set of *data subcomponents* $\mathcal{DATA\_S} \subseteq \mathcal{DATA}$; a set of *subprogram subcomponents* $\mathcal{SUB\_S} \subseteq \mathcal{SUB}$; a set of data ports $\mathcal{DP} = \{dp(data) \mid dp(data)$ is an in/out/in out *data port* of data type $data \in \mathcal{DATA}$ and of the form *port* (see Table 3)$\}$; a set of event ports $\mathcal{EP} = \{ep \mid ep$ is an in/out/in out *event port* and of the form *port*$\}$; a set of event data ports $\mathcal{EDP} = \{edp(data) \mid edp(data)$ is an in/out/in out *event data port* of data type $data \in \mathcal{DATA}$ and of the form *port*$\}$; a set of data accesses $\mathcal{DA} = \{da(data) \mid da(data)$ is a *data access* to shared data

data $\in \mathcal{DATA}$ and of the form *component_access*}; a set of subprogram accesses $\mathcal{SA} = \{sa(sub) \mid sa(sub)$ is a *subprogram access* to subprogram $sub \in \mathcal{SUB}$ and of the form *component_access*}; a *Mode State Machine* $\mathcal{MSM}$; and a *Behavioral Model* $\mathcal{BM}$.

Table 1: Running AADL example.

```
...
process implementation process.impl
  subcomponents
  thread_A: thread thread_1.impl;
  thread_B: thread thread_2.impl;
  sensorData_1 : data Base_Types::Integer;
  sensorData_2 : data Base_Types::Integer;
  displayData_1 : data Base_Types::Integer;
  connections
  Connection_1: data port thread_A.output_1 ->
  thread_B.input_1;
  Connection_2: data port thread_A.output_2 ->
  thread_B.input_2;
  Connection_3: access sensorData_1 ->
  thread_A.sensor_data_1;
  Connection_4: access sensorData_2 ->
  thread_A.sensor_data_2;
end process.impl;

thread thread_1
  features
  output_1: out data port Base_Types::Integer;
  output_2: out data port Base_Types::Integer;
  sensor_data_1 : requires data access sensorData_1;
  sensor_data_2 : requires data access sensorData_2;
end thread_1;

thread implementation thread_1.impl
  properties
  Dispatch_Protocol => Periodic; Period => 50ms;
  annex behavior_specification {**
  states
  s0: initial complete final state;
  transitions
  s0 -[ on dispatch ]->s0 { output_1 := sensor_data_1;
  output_2 := sensor_data_2 };
  **};
end thread_1.impl;

thread thread_2
  features
  input_1: in data port Base_Types::Integer;
  input_2: in data port Base_Types::Integer;
  display_data_1 : requires data access displayData_1;
end thread_2;

thread implementation thread_2.impl
  properties
  Dispatch_Protocol => Periodic; Period => 50ms;
  annex behavior_specification {**
  variables
  sensor_1, sensor_2, tmp: Base_Types::Integer;
  states
  s0: initial complete final state;
  s1: state;
  transitions
  s0 -[ on dispatch ]->s1 { sensor_1 := input_1;
  sensor_2 := input_2 };
  s1 [2] -[ sensor_1 != sensor_2 ]->s0 { mean!(sensor_1,
  sensor_2,tmp); display_data := tmp };
  s1 [1] -[ sensor_1 = sensor_2]->s0 { display_data
   := sensor_1 };
  **};
end thread_2.impl;
...
```

Table 2: Continuation of running example.

```
...
subprogram mean
  features
  x : in parameter Base_Types::Integer;
  y : in parameter Base_Types::Integer;
  z : out parameter Base_Types::Integer;
end mean;

subprogram implementation mean.impl
  annex behavior_specification {**
  states
  s0: initial state;
  s1: final state;
  transitions
  s0 -[ ]->s1 { z := (x + y)/2 };
  **};
end mean.impl;
...
```

A subprogram $sub = \langle \mathcal{DATA\_S}, \mathcal{SP}, \mathcal{EP}, \mathcal{EDP}, \mathcal{DA}, \mathcal{SA}, \mathcal{MSM}, \mathcal{BM} \rangle$ has a set of *data subcomponents* $\mathcal{DATA\_S} \subseteq \mathcal{DATA}$; a set of *subprogram parameters* $\mathcal{SP} = \{sp(data) \mid sp$ is an in/out/in out *parameter* of data type $data \in \mathcal{DATA}$ and of the form *parameter* }; a set of event ports $\mathcal{EP} = \{ep \mid ep$ is an out *event port* of data type $d \in \mathcal{DATA}\}$; a set of event data ports $\mathcal{EDP} = \{edp(data) \mid edp(data)$ is an out *event data port* of data type $data \in \mathcal{DATA}\}$; a set of data accesses $\mathcal{DA} = \{da(data) \mid da(data)$ is a *data access* to shared data $data \in \mathcal{DATA}\}$; a set of subprogram accesses $\mathcal{SA} = \{sa(sub) \mid sa(sub)$ is a *subprogram access* to subprogram $sub \in \mathcal{SUB}\}$; a *Mode State Machine* $\mathcal{MSM}$; and a *Behavioral Model* $\mathcal{BM}$.

Let $\mathcal{DP\_U}$, $\mathcal{EP\_U}$, $\mathcal{EDP\_U}$, $\mathcal{SP\_U}$, $\mathcal{DA\_U}$ and $\mathcal{SA\_U}$ denote the union of all sets of component data ports, event ports, event data ports, parameters, data accesses, and subprogam accesses respectively. $\mathcal{C}$ denotes the set of connections in the architecture, $\mathcal{C} = \{c(source, destination) \mid c$ is a *port connection* from $source \in \mathcal{DP\_U} \cup \mathcal{EP\_U} \cup \mathcal{EDP\_U}$ to $destination \in \mathcal{DP\_U} \cup \mathcal{EP\_U} \cup \mathcal{EDP\_U}$ of the form *port_connection*; or a *data access connection* (access to shared data) from $source \in \mathcal{DATA}$ to $destination \in \mathcal{DA\_U}$ of the form *data_access_connection*; or a *subprogram access connection* from $source \in \mathcal{SUB}$ to $destination \in \mathcal{SA\_U}$ of the form *subp_access_connection*; or a *parameter connection* from $source \in \mathcal{SP\_U} \cup \mathcal{DP\_U} \cup \mathcal{EDP\_U}$ to $destination \in \mathcal{SP\_U} \cup \mathcal{DP\_U} \cup \mathcal{EDP\_U}$ and $\langle source, destination \rangle \notin \mathcal{DP\_U} \times \mathcal{DP\_U} \cup \mathcal{DP\_U} \times \mathcal{EDP\_U} \cup \mathcal{EDP}_{\mathcal{U}} \times \mathcal{DP\_U}$ of the form *parameter_connection*}.

$\mathcal{CALL}$ denotes the set of subprogram calls in the architecture, $\mathcal{CALL} = \{call(sub) \mid call$ is a *subprogram call* to $sub \in \mathcal{SUB}$ of the form *subprogram_call*}.

A Behavioral Model $comp_i.\mathcal{BM} = \langle \mathcal{S}, s_o, \mathcal{CPL}, \mathcal{FIL}, \mathcal{VAR}, \mathcal{TR} \rangle$ has a set of *states* $\mathcal{S}$ of the form *state*; an *initial state* $s_0 \in \mathcal{S}$; a set of *complete states* $\mathcal{CPL} \subseteq \mathcal{S}$; a set of *final states* $\mathcal{FIL} \subseteq \mathcal{S}$; a set of typed variables $\mathcal{VAR}$ of the form *variable*; and a set of *state transitions* $\mathcal{TR} \subseteq \mathcal{S} \times \mathcal{PRI} \times \mathcal{G} \times \mathcal{ACT} \times \mathcal{S}$ of the form *state_transition*. A state $s \notin \mathcal{CPL} \cup \mathcal{FIL} \cup s_0$ is called an *execution state*. We shall use the denotation $s \xrightarrow{pri,g,act} s'$ iff $\langle s, pri, g, act, s' \rangle \in \mathcal{TR}$. $pri \in \mathbb{N}$ is the *priority* of the transition. g is a (possibly empty) set of *guards*, which are predicates (also known as *execute conditions*) over local variables, component ($comp_i$) in ports, component in parameters, subcomponent ($comp_i.sub\_s_j$) out ports, subcomponent out param-

eters, data subcomponents, or accesses to shared data components iff $s \notin \mathcal{CPL} \cup \mathcal{FIL}$; or predicates (also known as dispatch conditions) over (dispatch triggered by) event ports or event data ports (including receipt of a call) iff $s \in \mathcal{CPL}$. *act* is a (possibly empty) set of *actions* which are sequences (elements of a sequence are separated by ";" and executes in that order) and sets (separated by "&" and executes non-deterministically) of: subprogram calls with arguments of the form $sub!(list)$ where $sub \in \mathcal{SUB}$ and $list \in ARG \times ARG^*$ where ARG is the union of local variables, component ($comp_i$) in ports and parameters, subcomponent ($comp_i.sub\_s_j$) out ports and parameters, data subcomponents, and accesses to shared data components; of *assignments* of the form $target := expr$ where $target \in VAR \cup comp_i.DATA \cup comp_i.DA \cup comp_i.DP \cup comp_i.EP \cup comp_i.EDP$ where *expr* is an arithmetic expression over local variables, component in ports and parameters, subcomponent out ports and parameters, data subcomponents, and accesses to shared data components; and of timed actions of the form **computation(***min* **..** *max***)** which represent the use of the bounded CPU in terms of a duration between $min \in \mathbb{N}$ and $max \in \mathbb{N}$ time units.

A Mode State Machine $comp_i.\mathcal{MSM} = \langle \mathcal{M}, m_o, \mathcal{MTR} \rangle$ has a set of operational states (runtime configurations) called *modes* $\mathcal{M}$ of the form *mode*; an *initial mode* $m_0 \in \mathcal{S}$; and a set of *mode transitions* $\mathcal{MTR} \subseteq \mathcal{M} \times \mathcal{TRI} \times \mathcal{M}$ of the form *mode_transition*. We shall use the denotation $m \xrightarrow{tri} m'$ iff $\langle s, tri, s' \rangle \in \mathcal{MTR}$. $\mathcal{TRI}$ is a set of *triggers* which is the union of component ($comp_i$) in event and event data ports, and subcomponent ($comp_i.sub\_s_j$) out event and event data ports.

The complete semantics of the above abstract syntax is available in the AADL standard [2] and the Behavioral Annex [4].

4. SLICING THROUGH SYSTEM DEPENDENCE GRAPHS

Let $\mathcal{EXPR}$ be the set of possible expressions described by the abstract syntax. The slicing algorithm we propose builds on the general definition of program slicing originally discussed by Weiser [14]:

Definition 1. A backward slice of an AADL model with respect to slicing criterion $CRI = \langle comp, expr, var \rangle$, where $expr \in \mathcal{EXPR}$ is an expression within *comp*, and *var* is a variable or data component defined/assigned or used/read at *expr*, consists of all control flow and data flow determining expressions of the model that the value of *var* at *expr* in *comp* possibly depend on. A forward AADL slice with respect to slicing criterion $CRI = \langle comp, expr, var \rangle$ consists of all control flow and data flow determining expressions of the model that possible are dependent on the value of *var* at *expr* in *comp*.

As usual, there exist two types of dependencies: control dependence and data dependence.

Definition 2. An AADL expression $expr_1 \in \mathcal{EXPR}$ is *control-dependent* on an AADL expression $expr_2 \in \mathcal{EXPR}$ if $expr_2$ possibly decides whether $expr_1$ will be executed or not. $expr_1$ is *data-dependent* on $expr_2$ if $expr_2$ defines a data variable possibly used by an execution of $expr_1$.

Control and data dependencies of an AADL model are represented by its SDG. A SDG is generated through a process of algorithms starting with the generation of the CFGs of the components representing concurrent units of sequential execution: thread and subprogram components. The control flow through such a component is determined by its behavioral model that represents its logical execution, i.e. the CFG of a component is generated based on its behavioral model. Algorithms can then be applied to the CFGs to determine the internal control and data dependencies of each component. The set of (internal) control dependencies and the set of (internal) data dependencies of a component yield the CDG and the FDG of a component, respectively. The PDG of each component is then formed by merging their corresponding CDG and FDG. PDGs do not, however, have the ability to describe interdependencies among components. AADL models express control and data flow interactions among components throughout the architecture, from sensors to actuators through software components. The possible interactions among software components are represented by four different types of connections: *port connections*, *data access connections*, *subprogram calls*, and *parameter connections*. These expressions explicitly yield control and data dependencies and are similar to the dependencies that can be represented in a SDG, but not in a PDG, i.e., calls and parameter (data) passing. In order to generate the SDG of an AADL model, the set of PDGs of an AADL model must be integrated with these interdependencies.

In Section 4.1, we describe how CFGs of an AADL model are generated. In Section 4.2, we describe how PDGs are generated by performing operations on the CFGs, and in Section 4.3, we describe how to generate the SDG of an AADL model from the set of PDGs, which can be sliced for the purpose of selective regression verification as described in Section 5.

4.1 Generating Control Flow Graphs

The set of CFGs of an AADL model is generated by mapping the behavioral model ($comp_i.\mathcal{BM}$) of each thread component and subprogram component to its corresponding CFG as described in Algorithm 1.

Definition 3. A control flow graph $CFG(comp_i.\mathcal{BM}) = \langle V, A \rangle$ of a (possibly concurrent) component of sequential execution $comp_i$ is a directed graph of a set of vertices $V = \{v \mid v \in \mathcal{EXPR} \cup \langle \text{"}ENTRY\text{"}, comp \rangle \cup \langle \text{"}REENTRY\text{"}, comp \rangle \cup \langle \text{"}EXIT\text{"}, comp \rangle\}$ representing AADL expressions, and a set of arcs $A \subseteq V \times V$ describing how control flows through the vertices. Vertex v_1 of an arc $\langle v_1, v_2 \rangle \in A$ is called a **predecessor** of v_2 whereas vertex v_2 is called a **successor** of v_1. A vertex can have zero, one, or two successors. Let $outdegree(v)$ be a function mapping the number of successors to a vertex v and $indegree(v)$ the number of predecessors. A vertex v with $outdegree(v) = 2$ represents a so called **control expression** including a Boolean condition. The two outgoing arcs of v are attributed with $\langle v, v_x \rangle_T$ ($TRUE$) and $\langle v, v_y \rangle_F$ ($FALSE$) and correspond to the control flow in response to the evaluation of the condition. The $\langle \text{"}ENTRY\text{"}, comp \rangle$ vertex represents the point of the component *comp* through which control enters and $outdegree(\langle \text{"}ENTRY\text{"}, comp \rangle) = 1$. A $\langle \text{"}REENTRY\text{"}, comp \rangle$ vertex represents a point of the component *comp* through which control suspends, and reenters when the component has been reactivated/dispatched after the suspension

Table 3: AADL Grammar in Backus-Naur Form (BNF)

port_connection	::=	*identifier* **:** (**data port** | **event port** | **event data port**) *source_port_reference* (**− >** | **− >>**) *destination_port_reference*
data_access_connection	::=	*identifier* **: data access** *data_component_reference* (**− >** | **< − >**) *access_require_reference*
subp_access_connection	::=	*identifier* **: subprogram access** *subprogram_component_reference* **< − >** *access_require_reference*
parameter_connection	::=	*identifier* **: parameter** *source_parameter_reference* **− >** *destination_parameter_reference*
subprogram_call	::=	*identifier* **: subprogram** *subprogram_reference*
port	::=	*identifier* **:** (**in** | **out** | **inout**) (**data port** | **event data port** | **event**) *data_component_reference*
component_access	::=	*identifier* **: requires** (**data access** | **subprogram access**) *component_reference*
parameter	::=	*identifier* **:** (**in** | **out** | **in out**) **parameter** [*data_component_reference*]
state	::=	*state_identifier* **:** [**initial**][**complete**][**final**] **state**
variable	::=	*variable_declarator* **:** *data_component_reference*
state_transition	::=	[*identifier* [*priority*] **:**] *source_state_identifier* −[*guard*]− > *destination_state_identifier* [*action*]
mode	::=	*identifier* **:** [**initial**] **mode**
mode_transition	::=	[*identifier* **:**] *source_mode_identifier* −[*trigger*]− > *destination_mode_identifier*

and $outdegree(\langle \text{“}ENTRY\text{”}, comp\rangle) = 1$. The $\langle \text{“}EXIT\text{”}, comp\rangle$ vertex represents the point of the component *comp* through which control exits/stops and $outdegree(EXIT) = 0$. A **path** $P = v_1 v_2 \cdots v_n$ of CFG is a sequence of vertices such that $n \geq 2$ and for $i = 1, 2, \ldots, n-1$, $\langle v_i, v_{i+1}\rangle \in A$. A path $P = v_1 v_2 \cdots v_n$ is called a **basic block** if $v_1 \neq ENTRY \cup REENTRY$, $outdegree(v_1) = 1$, for $n > 2$ and $i = 2, 3 \ldots, n-1$, $indegree(v_i) = 1$ and $outdegree(v_i) = 1$, and $indegree(v_n) = 1$ and $outdegree(v_n) \geq 2$.

For simplicity, we assume that a $comp_i.\mathcal{BM}$ only includes deterministic behavior when defining the transformation to a $CFG(comp_i.\mathcal{BM})$. The assumption restricts the behavioral model (BM) such that actions of transitions cannot be of sets (i.e. actions must be of sequences), multiple outgoing edges from the same state must not have equivalent priorities (which otherwise execute non-deterministically), and there can only be one final state.

In a BM, the atomic expressions which define executable operations are guards and actions of state transitions. Hence, the vertices of $CFG(comp_i.\mathcal{BM})$ represent guards and actions of state transitions in $comp_i.\mathcal{BM}$. Each state transition yields a fixed execution order of operations: the guard of the transitions is first computed, and if evaluated to the Boolean value $TRUE$, the sequence of actions of the transition is executed according to the order of the sequence. Consequently, each transition $s \xrightarrow{pri,g,act} s'$, where $act = action_1; action_2; ...; action_n$ is a sequence of n actions (executes deterministically according to the sequence), maps to a CFG construct of one vertex $v_1 = g$ representing the guard of the state transition, a basic block of n vertices $v_2 = action_1, v_3 = action_2, \ldots, v_{n+1} = action_n$ representing the actions of the state transition, and n arcs $\langle v_1, v_2\rangle_T, \langle v_2, v_3\rangle, \ldots, \langle v_n, v_{n+1}\rangle$ representing the control flow through the executable operations. Note that the arc from the guard to the first action is attributed with a “T”. Let $stateTrToV : \mathcal{TR} \to \mathcal{P}(V)$ be a function mapping a state transition to a set of vertices, and $stateTrToA : \mathcal{TR} \to \mathcal{P}(A)$ to a set of arcs such that $stateTrToV(s \xrightarrow{pri,g,act} s') = \{v_1, v_2, v_3, \ldots, v_{n+1}\}$ and $stateTrToA(s \xrightarrow{pri,g,act} s') = \{\langle v_1, v_2\rangle_T, \langle v_2, v_3\rangle, \ldots, \langle v_n, v_{n+1}\rangle\}$ where $v_1 = g, v_2 = action_1, v_3 = action_2, \ldots, v_{n+1} = action_n$. The fixed execution order of operations is repeated throughout the BM until a final state is reached, regardless of the evaluation of the guard – and under the assumptions that the model is free from deadlocks. If evaluated to the Boolean value $TRUE$, the actions are executed, resulting in the arrival of a new state s', whereupon the transition going out from s' with the highest priority is executed according to the fixed order. If the guard is evaluated to the Boolean value $FALSE$, another state transition going out from s with the (next) highest priority is executed in the fixed order. Let $guardVertex : \mathcal{TR} \to \mathcal{V}$ be a function mapping a state transition $s \xrightarrow{pri,g,act} s'$ to the vertex v_x representing the guard of the state transition. Let $lastActionVertex : \mathcal{TR} \to \mathcal{V}$ be a function mapping a state transition $s \xrightarrow{pri,g,act} s'$ to the vertex v_x representing the last action of the state transition. Let $guardVertexPrio : \mathcal{S} \times \mathbb{N} \to \mathcal{V} \cup \{FALSE\}$ be a function mapping a state s to the vertex v_x representing the guard of the state transition going out from v and with the highest priority, or with the highest priority but less than n if a natural number is given as argument, or $FALSE$ if there exist no such vertex. In the case of an evaluation of a guard to $TRUE$, the control flow from the last action of the transition to the guard of the second transition with the highest priority is simply represented by an arc $\langle lastActionVertex(s \xrightarrow{pri,g,act} s'), guardVertexPrio(s')\rangle$. In case of an evaluation to $FALSE$, the control flow to the guard with the (next) highest priority is represented by an ($FALSE$-)arc $\langle guardVertex(s \xrightarrow{pri,g,act} s'), guardVertexPrio(s, pri)\rangle_F$.

It should be mentioned that actions may be of **if**, **while** and **for** constructs. Such an action comprises multiple vertices where control can leave the construct (action) from several vertices rather than a single one. In such constructs are predicates and nested actions also represented through distinguished vertices. Assume that v_x represents a control predicate expression of a loop or conditional, and v_y represents an action expression immediately nested within the loop or condition. If v_x is the predicate of a conditional expression the arc $\langle v_x, v_y\rangle$ is labeled with “T” or “F” according to weather v_y exists in the **then** branch, **elsif** or **else** branch. If v_x is the predicate of a while- or for-loop, the arc $\langle v_x, v_y\rangle$ is labeled with “T”. In case a state transition consists of an action sequence where the last action consists of an **if** construct, each (nested) action ending the control flow of the construct, including the current state transition, must be connected to the subsequent transition guard vertex, $REENTRY$ vertex, or $EXIT$ vertex by an arc.

Algorithm 1 Algorithm for generating control flow graphs

Input: $comp_i.\mathcal{BM} = \langle \mathcal{S}, s_o, \mathcal{CPL}, \mathcal{FIL}, \mathcal{VAR}, \mathcal{TR} \rangle$ and $\mathcal{TR}_{rel} \subseteq \mathcal{TR}$
Output: $\mathcal{CFG}(comp_i) = \langle V, A \rangle$

$V \leftarrow \emptyset \cup \{\langle$"$ENTRY$", $comp_i\rangle, \langle$"$EXIT$", $comp_i\rangle\}$
$A \leftarrow \emptyset$
for all $s \xrightarrow{pri,g,act} s' \in \mathcal{TR}_{rel}$ **do** ▷ generate vertices and arcs for each relevant transition
  $V \leftarrow V \cup stateTrToV(s \xrightarrow{pri,g,act} s')$
  $A \leftarrow A \cup stateTrToA(s \xrightarrow{pri,g,act} s')$
end for
$A \leftarrow A \cup \{\langle\langle$"$ENTRY$", $comp_i\rangle, guardVertexPrio(firstState($ $comp_i.\mathcal{BM}))\rangle$ ▷ Generate the arc representing control flow from the $ENTRY$ vertex to the guard vertex with highest priority
for all $cpl_j \in \mathcal{CPL}$ **do** ▷ generate possible $REENTRY$ vertices
  if $\exists s \xrightarrow{pri,g,act} s' \in \mathcal{TR}_{rel}[s' = cpl_j]$ **then**
    $V \leftarrow V \cup \{$"$REENTRY_j$"$\}$
    $A \leftarrow A \cup \{\langle$"$REENTRY_j$", $guardVertexPrio(s')\rangle\}$ ▷ Any control flow to "$REENTRY_j$" will successively flow to $guardVertexPrio(s')$
  end if
end for
for all $s \xrightarrow{pri,g,act} s' \in \mathcal{TR}_{rel}$ **do** ▷ Generate arcs to connect each transition representation to the subsequent guard, complete state (reentry) or final state representation
  if $s' \in \mathcal{CPL}$) **then**
    $A \leftarrow A \cup \{\langle lastActionVertex(s \xrightarrow{pri,g,act} s'), CPLState\ Vertex(s')\rangle\}$
  else if $s' \in \mathcal{FIL}$ **then**
    $A \leftarrow A \cup \{\langle lastActionVertex(s \xrightarrow{pri,g,act} s'), \langle$"$EXIT$", $comp_i\rangle\rangle\}$
  else$A \leftarrow A \cup \{\langle lastActionVertex(s \xrightarrow{pri,g,act} s'), guard\ VertexPrio(s')\rangle\}$
  end if
  if $guardVertexPrio(s, pri)$ **then** ▷ generate a false arc if a subsequent guard exists
    $A \leftarrow A \cup \{\langle guardVertex(s \xrightarrow{pri,g,act} s'), guard\ VertexPrio(s, pri)\rangle_F\}$
  end if
end for
return $\langle V, A \rangle$

The complete flow of control, i.e. the possible orders in which transitions are executed, is determined by the possible orders states can be visited through state transitions (the possible *paths* in the BM) and by the priorities of the state transitions. A BM of a subprogram component has: one initial state representing the starting point of a call; zero or more intermediate execution states representing the logical execution between start and completion of a call; and one final state representing the completion of a call. Thus, the initial state of a subprogram maps to an $ENTRY$ vertex whereas the final state maps to an $EXIT$ vertex. A BM of a thread component, on the other hand, has: one initial state representing the state (halted) of the thread before it is initialized; zero or more intermediate execution states representing the initialization steps (such as checking correctness of initial values of input and output ports) of the thread between the initial state and one, first, complete state (any path from the initial state will reach the same complete state before any other complete state); one or more complete states representing that the thread has suspended itself and is awaiting dispatch/reactivation (the first complete state reached from an initial state does also represent completion of initialization the first time it is reached); zero or more intermediate execution states representing logical execution between dispatches, that is, from and back to a complete state or between complete states; and one final state representing completion of finalization.

Execution of a subprogram component is triggered by incoming calls where the transition out from the initial state with the highest priority (if several) and with valid execute conditions is executed. A thread component, on the other hand, must first be initialized by an initialize action triggered when the process containing the thread is completely loaded into its virtual address space before it can be executed. An initialize action triggers the transition out from the initial state eventually leading to one, first, complete state. A state transition to a complete state means that the thread is calling an "await dispatch" run-time service, whereupon the thread is suspended after the action of the state transition has been executed. A dispatch of the thread component is triggered according to the dispatch conditions of the transitions out from the current complete state and the specified scheduling protocol of the thread. Dispatches of a periodic thread are solely triggered by a clock and the time interval (period) specified with the thread. In this case, dispatch conditions (guards of transitions out from complete states) are left empty. Dispatches of aperiodic, sporadic, timed and hybrid threads are essentially triggered by the arrival of an event or an event data at an event or event data port of the thread, or a remote subprogram call arriving to a provides subprogram access feature of the thread. By default, any arrival of event, event data or subprogram call triggers a dispatch where dispatch conditions restrict the number of triggers if modeled. In either case, an input-

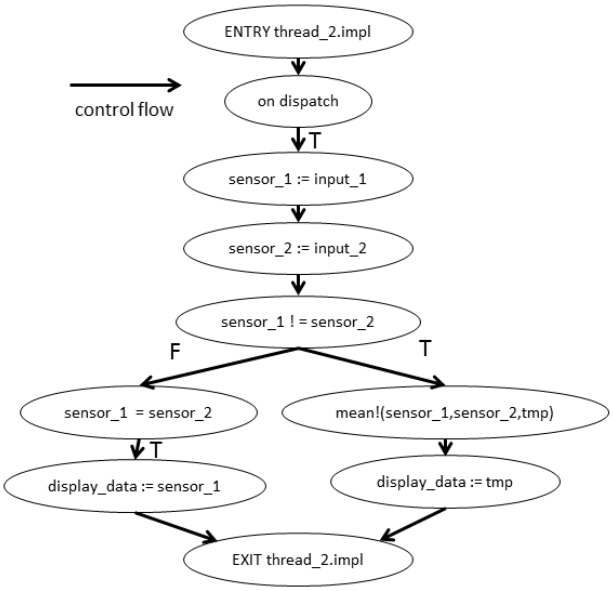

Figure 1: The CFG of thread_2.

compute-output model of execution is triggered. Input on in ports is frozen at the time of dispatch, where input from each port connection is read and assigned to a corresponding port variable which value (by default) is not affected by new arrivals for the remainder of the current dispatch. Output on out ports is transmitted through the connections at the time of completion, deadline or at specific output times according to an *Output_Time* property. For simplicity, we assume that the output is transmitted at completion. A

state transition to a final state means that the thread completes and is calling a "finalize" run-time service, whereupon the thread terminates after the action of the state transition has been executed. Consequently, a BM of a thread component, in contrast to a BM of a subprogram component, expresses state transitions which are not relevant to the logical execution (such as initialization transitions).

The relevant set of state transitions includes each transition that exists on every path from every complete state in the BM. Each of these are either from a complete state to a complete state, execution state, or a final state; or from an execution state to a complete state, an execution state, or a final state. Thus, the first complete state reached from an initial state maps to an $ENTRY$ vertex, any subsequently reachable complete states, including the one first reached from an initial state, maps to a $REENTRY$ vertex. The final state maps to an $EXIT$ vertex. Let $firstState : \mathcal{P}(\mathcal{S}) \times S \times \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{VAR}) \times \mathcal{P}(\mathcal{TR}) \rightarrow \mathcal{CPL}$ be a function mapping a BM to the initial state if the BM is of a subprogram component, or the first complete state reachable from the initial state if the BM is of a thread component. Let $CPLStateVertex : \mathcal{CPL} \rightarrow \mathcal{V}$ be a function mapping a complete state to its corresponding $REENTRY$ vertex. Note that the first complete state is mapped to both an $ENTRY$ and an $REENTRY$ vertex if there exist a transition back to the state. However, there exist only one distinguished $ENTRY$ vertex, so there is no need to define a function to retrieve it. Let $\mathcal{TR}_{rel} \subseteq \mathcal{TR}$ be the relevant set of state transitions of a BM. If the BM is of a subprogram component, then $\mathcal{TR}_{rel} = \mathcal{TR}$. The transformation from a $comp_i.\mathcal{BM}$ to the corresponding $CFG(comp_i.\mathcal{BM})$ can then be calculated as shown in Algorithm 1. The result of applying the algorithm on *thread_2.impl* of the running example (Table 1) is shown in Figure 1.

Each state transition out from or to a complete state comprises interactions (of control, data, or both) with other components if the thread has in ports or out ports connected to them, respectively. These are relevant to the logical execution but not considered in the CFG, however, they are considered when constructing the SDG from PDGs as described in Section 4.3.

4.2 Generating Program Dependence Graphs

A $CDG(comp_i)$ or a $FDG(comp_i)$ of a component is a directed graph $\langle V, A \rangle$ of a set of CFG vertices V and arcs $A \subseteq V \times V$ of the form $\langle v, v' \rangle_c$ or $\langle v, v' \rangle_d$. An arc $\langle v_1, v_2 \rangle_c$ labeled with "c" represents that v_2 is control-dependent on v_1. An arc $\langle v_1, v_2 \rangle_d$ labeled with "d" represents that v_2 is data (flow) dependent on v_1.

A $PDG(comp_i)$ of a component is simply the union of $CDG(comp_i)$ and $FDG(comp_i)$. Control dependencies of a CFG are calculated by so called dominator vertices [12]. Assume that v_x, v_y and v_z are vertices of $CFG(comp_i.\mathcal{BM})$. A vertex v_x is *post-dominated* by a vertex v_y if every path from v_x to the EXIT vertex includes v_y. Control dependency is then defined as:

Definition 4. A vertex v_y is *control-dependent* $\langle v_x, v_y \rangle_c$ on a vertex v_x iff **1)** v_x is an $ENTRY$ or an $REENTRY$ vertex and v_y is not nested within any loop or conditional vertex, or **2)** there exists a path P from v_x to v_y such that any vertex v_z in P is post-dominated by v_y, and v_x is not post-dominated by v_y (v_x must be a control expression).

An algorithm to generate the corresponding CDG of a CFG, based on this definition of control dependency, can be found in [12]. The corresponding CDG of the CFG in Figure 1 is shown in Figure 2.

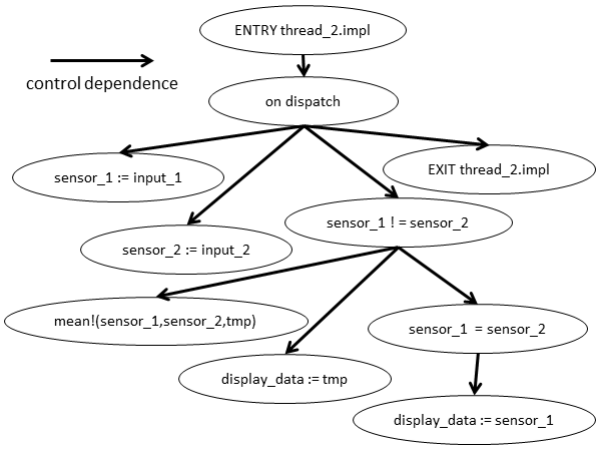

Figure 2: The CDG of thread_2.

Data dependencies of a CFG are calculated by so called *def-use pairs*. Assume that v_x is a vertex that defines/assigns variable var, and v_y is a vertex that uses/reads var. Flow-dependency is then defined as:

Definition 5. A vertex v_y is *data-dependent* $\langle v_x, v_y \rangle_d$ on a vertex v_x iff v_x defines/assigns a variable var that is used/read by v_y, and there exists a path P from v_x to v_y such that any vertex v_z in P does not define/assign var.

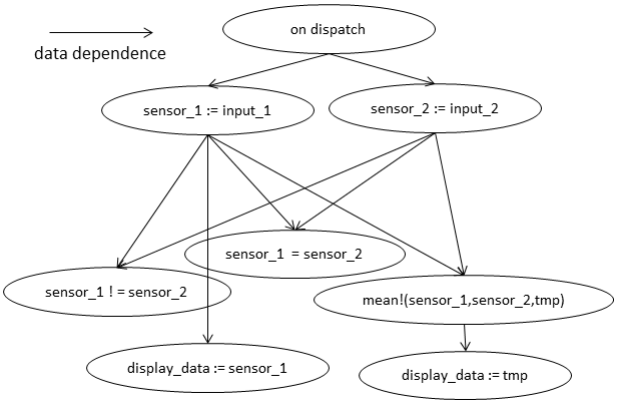

Figure 3: The FDG of thread_2.

Since each thread dispatch and subprogram activation includes assignments to input port and parameter variables if the component has such connections, a control flow from an $ENTRY$ and $REENTRY$ vertex includes assignment which are not represented in a CFG. Interactions between components are considered when constructing the SDG, however, to calculate all possible data dependencies it is assumed that all necessary initial assignments occur in the $ENTRY$ vertex of a subprogram, and in the dispatch condition vertices of $ENTRY$ and $REENTRY$ vertices of a thread. In

addition, a vertex including a subprogram call includes assignment of return values to in data ports connected with the out parameters of the called subprogram. Such assignments are assumed to occur in the vertex including the call. The corresponding FDG of the CFG in Figure 1 is shown in Figure 3.

4.3 Generating the System Dependence Graph

SDGs, as originally defined in [7], extend the expressiveness of PDGs such that procedure calls and parameter (data) passing (by value) can be integrated. The extension consists of five distinguished types of vertices to accurately represent the semantics of procedure calls and parameter passing: *call* vertices ($subprogram(arg_0, arg_1, \ldots, arg_m, var_0, var_1, \ldots, var_n)$ where $m, n \in \mathbb{N}$), representing call sites; *actual-in* vertices ($\{temp_i\_in = arg_i \mid i \in \mathbb{N} \text{ and } 0 \leq i \leq m\}$), representing assignments that copy the values of the actual arguments of call sites to temporary "in" variables; *formal-in* vertices ($\{parameter_i = temp_i\_in \mid i \in \mathbb{N} \text{ and } 0 \leq i \leq m\}$), representing assignments that copy the values of temporary "in" variables to the formal parameters of procedures; *formal-out* vertices ($\{temp_i\_out = return_i \mid i \in \mathbb{N} \text{ and } 0 \leq i \leq n\}$), representing assignments that copy the values of return variables of procedures to temporary "out" variables; and *actual-out* vertices ($\{var_i = temp_i\_out \mid i \in \mathbb{N} \text{ and } 0 \leq i \leq n\}$), representing assignments that copy the values of temporary "out" variables to the variables assigned by the calls. Actual-in and actual-out vertices are control-dependent on the call vertex, whereas formal-in and formal-out vertices are control-dependent on the entry vertex (of the called subprogram). In addition, the extension includes three distinguished types of dependence edges, which also maintain a clear structure of the system: *call* edges, representing the control dependence between a call site and the entry of the called procedure; *parameter-in edges*, representing the data dependence between an actual-in vertex and the corresponding formal-in vertex; *parameter-out* edges, representing the data dependence between a formal-out vertex and the corresponding actual-out vertex. A SDG is basically formed by: 1) representing each procedure of a program by a PDG extended with vertices for procedure calls and parameter passing; 2) connecting call vertices of PDGs to the entries of the corresponding called PDGs through call edges; and connecting actual-in vertices to formal-in vertices and formal-out vertices to actual-out vertices through parameter-in and parameter-out edges respectively. In AADL, however, there are additional types of interdependencies than procedure calls and parameter passing. The original definition of a SDG must therefore be extended to be applicable to AADL models.

A $SDG(\mathcal{AADLMDL})$ of an AADL model is a directed graph $\langle V, A \rangle$ of a set of CFG vertices V and arcs $A \subseteq V \times V$ of the form $\langle v, v' \rangle_c$, $\langle v, v' \rangle_d$, $\langle v, v' \rangle_{call}$, $\langle v, v' \rangle_{p-in}$, $\langle v, v' \rangle_{p-out}$, and $\langle v, v' \rangle_{mode}$ representing control, data, call and event, data passing by value and reference, and mode dependencies. A SDG of an AADL model is formed by generating the set of PDGs, and annotating them to include interdependencies between the set of PDGs, such as shown in Figure 4 where the SDG of our running example is presented. The possible interactions among components are represented by four different types of connections: *port connections*, *data access connections*, *subprogram calls*, and

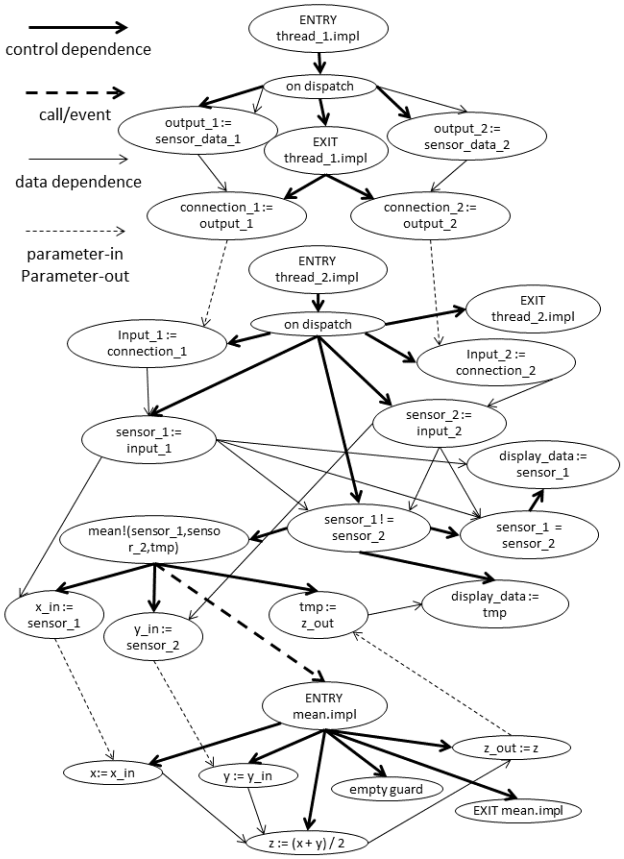

Figure 4: The SDG of the AADL example.

parameter connections. Ports, parameters, and data components are accessible as typed data variables.

Port connections represent transfers of data (by value), control, or both, depending on the type of interconnected interfaces: data ports, event ports, or event data ports. A data port connection can be accurately represented by an unidirectional variant of SDG parameter passing including an actual-in vertex $connection_i\_temp = outdataport\_var$ and a formal-in vertex $indataport\_var = connection_i\_temp$ interconnected through a parameter-in (data dependence) arc denoted $\langle connection_i\_temp = outdataport\_var, indataport\_var = connection_i\_temp \rangle_{p-in}$. The actual-in vertex is control dependent on each $REENTRY$ vertex and the $EXIT$ vertex of the sending thread (since we assume transfer of output at completion). The formal-in vertex is control-dependent on the dispatch condition vertices of the receiving thread. E.g., data port connection *Connection_1* in the AADL example (Table 1) maps to the actual-in vertex "$connection\_1 := output\_1$" and the formal-in vertex "$Input\_1 := connection\_1$" in Figure 4. An event port connection can be accurately represented by an event port "call" vertex *outeventport_var*! (an exclamation mark denotes the triggering of an event in AADL) and a target event port variable vertex *ineventport_var* interconnected through a call (control dependence) arc denoted $\langle outeventport\_var!, ineventport\_var \rangle_{call}$. The dispatch condition vertices of the receiving thread is control-dependent on the event port variable vertex. Event data port connections can be represented through an actual-in vertex $connection_i\_temp = outevent$

dataport_var and a formal-in vertex *ineventdataport_var* = $connection_i$*_temp* interconnected through both a parameter in arc and a call arc, where actual-in and formal-in vertices are control-dependent on the sending and receiving thread according to the union of data and event port connections.

Subprogram calls represent transfers of control whereas *parameter connections* represent transfers of data (by value). Control and data dependencies of subprogram calls with parameter connections can be represented as originally defined in a SDG [7], however, where actual-in, formal-in, formal-out, and actual-out vertices operates on port and parameter variables. E.g., the subprogram call *mean!(sensor_1,sensor_2,tmp)* in the AADL example (Table 1) maps to actual-in vertices "*x_in* := *sensor_1*" and "*y_in* := *sensor_2*", formal-in vertices "*x* := *x_in*" and "*y* := *y_in*", formal-out vertex "*z_out* := *z*", and actual-out vertex "*tmp* := *z_out*" in Figure 4.

Data access connections to a common data component may represent transfers of data (by reference) if there exist both write-right (a component is able to write data) and read-right (a component is able to read data) access connections. In case this condition holds, data dependence between a write-right ($connection_i$) and a read-right ($connection_j$) data access connection can be represented through a variant of SDG parameter passing including an actual-in vertex $connection_{i-j}\_temp = datacomp\_var$, representing the write-right connection, and an inverting formal-in vertex $datacomp\_var = connection_{i-j}\_temp$, representing the read-right connection, interconnected through a parameter-in arc. We assume that a thread or a subprogram gets the data source upon dispatch, and releases it upon a completion. There are thereby no data dependencies among components on the data component during the execution of a thread or subprogram. Consequently, the actual-in vertex is control-dependent on each $REENTRY$ vertex and the $EXIT$ vertex of the sending thread, or the $EXIT$ vertex of the sending subprogram. The formal-in vertex is control-dependent on the dispatch condition vertices of the receiving thread, or the $ENTRY$ vertex of the receiving subprogram.

The runtime configuration of subcomponents and their interactions within a component may change if it is specified with *modes*. For each mode, it is possible to set the active components and connections, mode-specific subprogram calls, and mode-specific properties. Modes essentially determine if complete components and interactions between components will be executed or not and therefore express control dependencies. Hence, entry vertices of components and vertices involved with dependencies between components may be control-dependent on modes. To accurately represent the SDG of an AADL model, we extend the expressiveness of a SDG with a *mode* and a *mode trigger* vertex type representing modes and mode triggers, and a *mode dependence* arc type representing control dependence due to modes. Each mode, except for the initial mode, in a mode state machine is control-dependent on the previous mode and the mode transition triggers of the mode transitions to the mode. Hence, for each mode transition $m \xrightarrow{tri} m'$ of each $comp_i.\mathcal{MSM}$, the following vertices and dependence arcs are generated:$\langle$"$MODE$", $m\rangle \in V$, $\langle$"$MODETRI$", $tri\rangle \in V$, $\langle$"$MODE$", $m'\rangle \in V$, $\langle\langle$"$MODE$", $m\rangle$, $\langle$"$MODE$", $m'\rangle\rangle_{mode} \in A$, and $\langle\langle$"$MODETRI$", $tri\rangle$, $\langle$"$MODE$", $m'\rangle\rangle_{mode} \in A$.

To complete the interdependencies between a set of PDGs to form the SDG, a dependence construct according to above description must be generated for each connection and mode state machine. The constructs and PDGs are completely integrated by adding a control, data, and/or a call dependence edge for each definition, use, event call, and event call retrieval of a port, parameter, or data component variable of each PDG that yields a data or control dependency with the added set of vertices. E.g., vertex "*output_1* := *sensor_data_1*" yields a data dependency with the added actual-in vertex "*connection_1* := *output_1*" in Figure 4 and must be connected with a data dependence edge as shown. Since we assume that output is sent on the time of completion, an actual-in or a formal-out vertex is only data-dependent on the corresponding final definitions of the out data port, out event data port, out parameter, or data component variable of each thread dispatch or subprogram activation. Note that a CFG with multiple paths between dispatches may include multiple final definitions. An actual-in vertex, of an event data port connection, or an event port call vertex is control-dependent on the corresponding final definitions and event calls. In addition, entry, actual-in, formal-in, formal-out, and actual-out vertices that are dependent on modes are connected to the corresponding mode vertices through mode-dependence arcs.

Note that in the construction of a PDG, it is assumed that initial assignments of in data or event data ports, in parameters, and accessed data components variables, and of return values of subprogram calls to in data ports occur in $ENTRY$ vertices of subprograms, dispatch vertices of threads, and vertices including subprogram calls. In a SDG are these assignments explicitly represented in formal-in and actual-out vertices. Any data dependence on such vertices is substituted with a data dependence on the corresponding formal-in or actual-out vertex. E.g., vertex "*sensor_1* := *input_1*" is data-dependent on formal-in vertex "*Input_1* := *connection_1*" in Figure 4, rather than on the dispatch vertex as in Figure 3. The corresponding SDG of the running example according to the rules defined in this section is presented in Figure 4.

5. SLICING AND SELECTION

Through comparison of SDGs generated from the architecture model and its modified version, the modifications can be identified and their effects on the system architecture can be traced by slicing the modified model with respect to the variables assigned or used in the modifications. A modification is defined as an added or changed vertex, or a vertex which dependency on another vertex has been removed, added or changed. Verification sequences which do not cover modified or affected vertices, i.e., the sliced model, can then be disregarded to generate a more efficient subset for regression verification.

A backward-slice of an AADL model with respect to slicing criterion $CRI = \langle comp, expr, var\rangle$ and a $SDG = \langle V, A\rangle$ is simply the subset of vertices $V_{b-slice} \subseteq V$ that are backwards reachable (through arcs) from vertex $expr \in V$ (including $expr \in V$). A forward-slice, on the other hand, is simply the subset of vertices $V_{f-slice} \subseteq V$ that are forward reachable (through arcs) from vertex $expr \in V$ (including $expr \in V$).

As an example, consider the SDG of our running example and assume that there exist a prior SDG of a prior version

of the AADL model in Table 1 and Table 2. Further assume that according to a comparison between the two SDGs, the vertex "$z := (x + y)/2$" is a changed vertex and constitutes the modification and therefore also the slicing criterion. The forward-slice $V_{f-slice} = \{$"$z := (x + y)/2$", "$z\_out := z$", "$tmp := z\_out$", "$display\_data := tmp$"$\}$ includes the set of elements of the AADL model that may exhibit a different behavior with respect to the prior version and thus must be re-verified. Any previous verification sequence covering any of these vertices should therefore be re-executed since it may generate a different result. The backward slice includes all vertices that may affect the behavior of the modification and, depending on the verification technique (e.g. unit- or system-level verification) and the coverage criteria (e.g. statement, condition, or path coverage), can be used to guide the selection process. There are two extremes: 1) if the technique has the ability to directly execute the modification (e.g. directly call a modified subprogram rather than stimulating the system with input that eventually causes a call to the modified subprogram) and the coverage criteria allow it, none of the vertices in the backward slice except for the modification and vertices also included in the forward-slice may be covered, and 2) if the coverage criteria require all possible scenarios in which the modification may be executed to be covered, all vertices of the backward slice must be covered (in fact, all paths of the backward slice must be covered for full coverage). Despite verification technique and coverage criteria, vertices that are not included in the forward nor backward transitive closure {"$sensor\_1 = sensor\_2$", "$display\_data := sensor\_1$"} can, with respect to the modification, be confidentially disregarded in the regression verification process. With auxiliary vertices (ENTRY and EXIT) excluded, this constitutes a $2/23 \approx 9\%$ to $19/23 \approx 83\%$ reduction of exercised elements.

6. CONCLUSIONS

In this paper we presented a technique for selective regression verification of AADL models through slicing. We showed how it can be applied to reduce the scope of re-verification due to modifications. The technique allows for a more efficient regression verification process and could therefore result in significant cost and time savings.

In the current form, the technique may however introduce a slight overestimation of the data and control dependencies between concurrently executed tasks, by not taking into account dynamic properties such as patterns of (timed) scheduling and concurrency protocols for access to shared memory (critical regions). These properties may, in a complex manner, constrain the possible flows of control and data between concurrent tasks and, thus, the possible number of dependencies. A future area of improvement is an inclusion of such dynamic properties to generate more precise dependencies. The overestimation however makes interdependences between tasks easy and fast to compute, which advocate the overhead a selective approach yields compared to a rerun-all approach. The downside is that some elements may still be unnecessarily exercised in the regression verification process, which delimits the return on the investment. The possible return in terms of time-efficiency with respect to a non-selective approach is a subject for future research. In the running example of this paper, the technique resulted in a 9% to 83% reduction of exercised elements. As slicing of SDGs can be performed in linear time [7], such results indicate a beneficial use.

7. REFERENCES

[1] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[2] As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506A, 2009.

[3] S. Bates and S. Horwitz. Incremental Program Testing Using Program Dependence Graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 384–396, New York, NY, USA, 1993. ACM.

[4] R. B. Franca, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas. The AADL behaviour annex – experiments and roadmap. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 377–382, Washington, DC, USA, 2007. IEEE Computer Society.

[5] R. Gupta, M. Jean, M. J. Harrold, and M. L. Soffa. An Approach to Regression Testing using Slicing. In *In Proceedings of the Conference on Software Maintenance*, pages 299–308. IEEE Computer Society Press, 1992.

[6] M. J. Harrold. Testing: A Roadmap. In *In The Future of Software Engineering*, pages 61–72. ACM Press, 2000.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[8] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat. Automated Verification of AADL-Specifications Using UPPAAL. *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*, 0:130–138, 2012.

[9] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. In *In Proceedings of the Fourth Irvine Software Symposium*, 1994.

[10] N. G. Leveson. *Safeware: system safety and computers*. ACM, New York, NY, USA, 1995.

[11] T. Oda and K. Araki. Specification slicing in formal methods of software development. In *Proceedings of the 17th annual International computer software and applications conference (COMPSAC'93)*, pages 313–319, 1993.

[12] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, Apr. 1984.

[13] J. Silva. A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.

[14] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.