Reachability of System Operation Modes in AADL

Lutz Wrage

May 2024

TECHNICAL REPORT
CMU/SEI-2024-TR-003
DOI: 10.1184/R1/24764256

Software Solutions Division

[Distribution Statement A] Approved for public release and unlimited distribution.

http://www.sei.cmu.edu
Table of Contents

Abstract ii

1 Introduction 1
   1.1 Modes in AADL 1
   1.2 System Operation Modes 2

2 SOM Reachability Analysis 5
   2.1 Overview 5
   2.2 Adding Modes to a System Operation Mode 5
   2.3 Creating SOM Transitions 6
      2.3.1 Transitions for Non-Modal Connections 8
      2.3.2 Transitions for Modal Connections 13
   2.4 Testing SOM Reachability 16
   2.5 The Full Algorithm 16

3 Example Models 19
   3.1 Non-Modal Connections 19
   3.2 Modal Connections 26

4 Conclusion and Future Work 29

A Appendix 30
   A.1 Data Model 30
   A.2 Java Implementation 33

References 47
Abstract

Components in an Architecture Analysis and Design Language (AADL) model can have modes that determine which subcomponents and connections are active. Transitions between modes are triggered by events originating from the modeled system’s environment or other components in the model. Modes and transitions can occur on any level of the component hierarchy. The combinations of component modes (called system operation modes or SOMs) define the system’s configurations. It is important to know which SOMs can actually occur in the system, especially in the area of system safety, because a system may contain components that should not be active simultaneously, for example, a car’s brake and accelerator. This report presents an algorithm that constructs the set of reachable SOMs for a given AADL model and the transitions between them.
1 Introduction

1.1 Modes in AADL

AADL (the Architecture Analysis and Design Language) [3] defines components, their interconnectivity, and their relationship to each other. AADL software components communicate via ports that are connected to ports of other components. Execution platform components are linked to each other via access features that are connected to other access features. An AADL model forms a hierarchical structure by allowing components to be nested inside other components as subcomponents. Software components are bound to execution platform components. Each AADL (instance) model has a single top-level root element, the system instance. AADL elements can have properties that provide additional information for use when analyzing the model or generating code from it.

Each AADL component can have one or more modes that define a configuration of the component, that is, which of its subcomponents are active, together with the values of its properties in this mode. If a component has several modes, one of them is marked as the initial mode. A component transitions from one mode to another in response to one of the following events:

• one received on a port
• one generated internally by the component itself
• one generated by one of its subcomponents

AADL elements such as subcomponents or connections are called modal if they are active in only some, but not all, modes of their containing component. In this case, the element’s declaration lists the modes in which it is active. In terms of model analysis, an inactive component may be treated as not present, and the details are dependent on the analysis. For example, a scheduling analysis can ignore inactive threads because they do not consume processor time. In contrast, an analysis that sums the weights of physical components must still consider the weight of inactive processor or memory components. Software bindings to execution platform components can also be modal, for example, to indicate that a thread runs on different processors, depending on the system’s operating mode.

The model in Listing 1.1 shows a system with two modes: $m_0$ and $m_1$. Transitions between these modes can be triggered by an external event that enters the system via port $e_0$. The system has three subcomponents: $a$, $b$, and $c$. Components $a$ and $b$ are always active, but $c$ is active only if the system is in mode $m_1$.

Subcomponent $b$ has modes itself ($m_{10}$ and $m_{11}$), and transition between them can be triggered by event $e_1$ originating in component $a$. From port $a.e_1$, the event travels along connection $c_1$ to component $b$. Note that the connection is modal and exists in mode $m_0$ only. Whenever the system is in mode $m_1$, no event from $a$ can trigger a transition in $b$. 

CMU/SEI-2024-TR-003 | SOFTWARE ENGINEERING INSTITUTE | CARNEGIE MELLON UNIVERSITY

[Distribution Statement A] Approved for public release and unlimited distribution.
package ModesIntro
public
  system S
  features
e0: in event port;
  modes
  m0: initial mode;
m1: mode;
m0 --> e0 --> m1;
m1 --> e0 --> m0;
end S;
system implementation S.i0
subcomponents
  a: system A;
b: system B;
c: system C in modes (m1);
connections
c1: port a.e1 --> b.e1 in modes (m0);
end S.i0;
system A
features
e1: out event port;
end A;
system B
features
e1: in event port;
modes
  m10: initial mode;
m11: mode;
m10 --> e1 --> m11;
m11 --> e1 --> m10;
end B;
system C
end C;
end ModesIntro;

Listing 1.1: Introductory Mode Example

1.2 System Operation Modes

The set of all component modes in an AADL instance model is called the *system operation mode* (SOM). The initial SOM is defined as the model components’ set of initial modes. Each mode transition in a component is related to a transition between SOMs for the complete system.

The possible SOMs of a system are defined as all combinations of component modes. In the example above, the possible SOMs are

- \((m_0, m_{10})\)
- \((m_0, m_{11})\)
- \((m_{1}, m_{10})\)
- \((m_{1}, m_{11})\)

In general, some SOMs may not be reachable.

This is the first set of rules that influence which SOMs are possible:

- The modes of an inactive component do not contribute to the set of SOMs.
The modes of its subcomponents do not contribute either, since they are all inactive too. Following these rules, we refine the SOM definition: The set of SOMs is defined as the combination of the reachable modes of all active components.\textsuperscript{1} For the purposes of this report, we include inactive components in an SOM: An SOM defines the following for each component $C$ in a model:

- whether it is active
- its mode, if it has one

When considering events and connections, possible SOMs are not reachable when:

- an event that triggers a mode transition is not connected to an event’s source, that is, either
  - a component’s out port for an event generated inside the system
  - a system instance’s in port for an event generated in the system’s environment
- a component that can generate a trigger event is inactive
- a connection that can transport a trigger event is inactive
- a component mode is not reachable via a sequence of mode transitions from the initial mode

The possible SOM transitions are further limited by a rule in the AADL standard demanding that all component mode transitions triggered by the same event happen simultaneously.

AADL version 2 introduced modes that are derived from the modes of their containing component. AADL does not allow direct transitions between derived modes. These derived modes are added to the system’s SOMs but do not influence the number of SOMs or which SOM transitions are possible.

In this report, we describe an algorithm that uses all the above rules to determine the set of reachable SOMs in a system instance and the possible transitions between them. Our algorithm is a significant extension of the first step of an algorithm to translate AADL mode state machines into timed Petri nets\cite{1}. That algorithm, documented by Bertrand et al., has a few shortcomings compared to ours. It does the following:

- only handles mode transitions triggered by external events
- assumes that all connections are non-modal
- does not address the resumption policy for reactivating components

In addition, Bertrand et al.’s paper addresses transitions triggered simultaneously in several components only in the special case of a component and its subcomponents, but not for unrelated components.

We have implemented the reachability analysis algorithm as a plug-in to the Open Source AADL Tool Environment (OSATE\cite{5}), which enables new verifications that could be run on an AADL model.

Knowing which SOMs are reachable helps to answer questions such as the following about correctness of a system design:

- Is it possible for the autopilot subsystem to be active when the system is in takeoff or landing mode?
- Does the system contain components that are never active?

\textsuperscript{1}Model instantiation in OSATE generates SOMs according to this definition.
• Are all modes of a component reachable?

• Is it possible for components $C_1$ and $C_2$ to be active simultaneously? Or is that always the case?

• Which modes are possible for component $C_1$ when component $C_2$ is in mode $m$?

Knowing the possible transitions allows exporting the SOMs and transitions for further analysis with a model checker to answer additional questions such as

• Can the system get into a state where a component $C$ is inactive and unable to be reactivated?

• Which modes are possible for component $C_1$ before component $C_2$ is in mode $m$?

Our OSATE plug-in includes an export to the NuSMV [2] input language. A user can add LTL or CTL formulae that represent the above questions and verify them by running NuSMV.2

The rest of this report is organized as follows:

• Chapter 2 describes our reachability analysis algorithm and gives the pseudocode for two variants.

• Chapter 3 shows the analysis output for a few small non-modal and modal AADL models.

• Chapter 4 (the conclusion) describes outstanding extensions to all AADL models and future work.

• Appendix A describes the data model used in the analysis implementation and includes the main Java code of our prototype implementation.
2 SOM Reachability Analysis

In this section, we describe the algorithm for determining which SOMs are reachable for a given AADL instance model. We make the simplifying assumption that the system is synchronized, that is, all components use the same globally synchronized reference time. In such a system, it makes sense to talk about mode transitions occurring simultaneously if they are triggered by the same event. We only consider events that originate at a component feature and ignore internal features. We also assume that the model does not contain derived modes. In Chapter 4, we discuss how the algorithm can handle internal features and derived modes.

2.1 Overview

The basic idea of our SOM reachability analysis is to create a state machine for a subset of a model’s components. This state machine contains the SOMs and transitions between them for this subset of components. We start with the root system instance component $R$. This component is always active. The corresponding SOMs and SOM transitions are just the modes and mode transitions of $R$, restricted to the modes that are reachable from the initial mode. If the root component has no modes, a single SOM represents this active component.

We then incrementally add components until all components in the analyzed model are handled. For each new component $C$, we extend the set of SOMs to include the modes of the new component and merge $C$’s mode transitions into the set of SOM transitions. Once all components have been added, we have the complete set of reachable SOMs and SOM transitions for the analyzed model.

For the incremental processing, a subcomponent should be added only after the parent component that contains it has been added. Following this order constraint, we know if the newly added component is active or inactive when it is added for each existing SOM, because subcomponents cannot influence whether a parent component is active. The simplest processing sequence is to traverse the component containment hierarchy and add components in pre-order, although other traversals (e.g., level-order) would lead to the same result.

We can extend an existing SOM by creating a new SOM for each mode of $C$, together with an indication of whether $C$ is active in the new SOM. After that, merging transitions is the crucial step that takes into account if transitions are triggered by the same event, if the source component of the trigger event is active, and so on. Finally, we remove any SOMs that are not reachable from the initial SOM by following the new set of SOM transitions.

In the next sections, we describe in detail how to create extended SOMs and transitions sets when processing a component.

2.2 Adding Modes to a System Operation Mode

An SOM consists of a sequence of modes for each component that has been processed and an indication of whether the component is active or inactive in the SOM. We use the following symbols to represent component $C_k$ in an SOM:

- $\top^k$ – $C_k$ is active and has no modes.
- $\top^k_m$ – $C_k$ is active, and its mode is $m$.
- $\bot^k$ – $C_k$ is inactive and has no modes.
- $\bot^k_m$ – $C_k$ is inactive and resumes in mode $m$.
If a component that is active with mode \( m \) is deactivated during an SOM transition and later reactivated, the component continues in either mode \( m_0 \) (the initial mode) or \( m \). In the AADL model, the reactivation behavior can be selected using property `Thread.Properties::Resumption_Policy` with values `restart` (to continue in \( m_0 \)) and `resume` (to continue in \( m \)).

When the component is obvious from the context, we leave out the superscript \( k \). We generally use \( m_0 \) to represent the initial mode of a component and \( m_1, m_2, \ldots \) for its remaining modes. To simplify the discussion, we treat a component without modes as if it has a single mode \( m_0 \) so that we do not need to treat a component without modes as a special case. With this notation, we define the *mode states* of component \( C_k \) with modes \( m_0, m_1, \ldots, m_{j_k} \) as this set:

\[
MS_k = \begin{cases} 
\{T_{m_0}, T_{m_1}, \ldots, T_{m_{j_k}}\} & \text{if } C_k \text{ is active} \\
\{\perp_{m_0}\} & \text{if } C_k \text{ is inactive with resumption policy } \text{restart} \\
\{\perp_{m_0}, \perp_{m_1}, \ldots, \perp_{m_{j_k}}\} & \text{if } C_k \text{ is inactive with resumption policy } \text{resume}
\end{cases}
\]

We write SOM\(_k\) for the set of SOMs in the model that contains components \( C_0, C_1, \ldots, C_k \), where \( C_0 \) is the top-level system instance.

When processing \( C_0 \), we initialize the set SOM\(_0\) as follows. Note that \( C_0 \) is always active.

\[
SOM^0_0 = MS_0 = \begin{cases} 
\{T_{m_0}\} & \text{if } C_0 \text{ has no modes} \\
\{T_{m_0}, T_{m_1}, \ldots, T_{m_{j_k}}\} & \text{if } C_0 \text{ has one or more modes}
\end{cases}
\]

SOM\(_0\) is the set of reachable SOMs in SOM\(_0\).

Whether a component \( C_k, k > 0 \) is active is determined by the mode state of its parent component. This mode state is part of the SOMs in SOM\(_{k-1}\) where MS\(_k\) can be interpreted as a function from SOM\(_{k-1}\) to the set of mode states for \( C_k \).

When processing component \( C_k \), the set SOM\(_k\) is constructed based on SOM\(_{k-1}\):

\[
\forall k > 0 : SOM^k_k = \bigcup_{s \in SOM_{k-1}} s \times MS_k(s)
\]

SOM\(_k\) is the set of reachable SOMs in SOM\(_k\).

### 2.3 Creating SOM Transitions

Before describing the algorithm to determine SOM transitions, we must discuss how connection instances relate to connections in a declarative AADL model. A *connection declaration* describes either a mapping between a component port and a port of one of its subcomponents, or the connection of ports that belong to its sibling component—that is, direct subcomponents of the same parent component. The AADL standard describes how a sequence of declarative connections defines a *semantic connection* between two components.

In the context of an AADL instance model, we use the more general notion of *connection instance*, which is a sequence of declarative connections that is complete in the sense that it cannot be extended by another declarative connection. A connection instance is allowed to start and end at components with any category, whereas a semantic connection is defined only

---

1 The AADL standard defines no default value for this property. In our implementation we use `restart` behavior for all components that have no value for this property.
for port connections between threads. Mode transitions are triggered by events that are transported via port (or feature) connections. These events can be internal or external to the modeled system:

- an internal trigger event
  - is generated by a component \( C \) in the model
  - starts at one of \( C \)'s ports
  - follows a sequence of declarative connections that map to outgoing ports of parent components
  - goes to an incoming port of a sibling component
  - follows a sequence of mappings to subcomponent ports

- an external trigger event
  - is generated outside the modeled system
  - starts at an incoming port of the top-level component \( C_0 \)
  - follows a sequence of mappings to subcomponent ports

In the rest of this report, we use connection for a connection instance and segment for the declarative connections that make up the connection instance.

An event can trigger multiple mode transitions via several connections but also via a single connection: Each component the connection passes through may contain a mode transition that names a port in this connection as a trigger.

We define \( TN_k \) as the set of SOM transitions between the SOMs in \( SOM_k \). A mode transition is a tuple \( tn = (s, tg, d) \) with \( s, d \in SOM_k \), and \( tg \) as the source of the event triggering the transition. The trigger source is generally the feature at the source of the connection that transports the trigger event. In contrast, the mode transition in the declarative model typically lists the destination features of such connections as triggers. Multiple connections can go through the same feature, so a single trigger for a mode transition in the declarative model can result in multiple triggers for a corresponding SOM transition. In addition, AADL allows multiple connections between the same features, so a trigger event could follow two or more paths to the same transition. For modal connections, these paths may be active in different SOMs.

In a modal model, connections can be active or inactive in an SOM just like components. A connection is modal if at least one of its segments is modal. A modal connection is

- active if all of its segments are active
- inactive if at least one of its segments is inactive

A segment is inactive if one of the following conditions is true:

- Its containing component is inactive.
- The segment is modal, and the component’s current mode is not listed in the segment’s \texttt{in modes} clause.

Creating the correct set of SOM transitions is the core step of our reachability algorithm. We present this step in two parts:

1. We give the algorithm for models where all connections that transport trigger events are non-modal.
2. We extend it for the case of modal connections.
2.3.1 Transitions for Non-Modal Connections

A non-modal connection is active in an SOM if all of these elements are active:

- the source component
- all components through which the connection passes
- the destination component

To determine whether a non-modal connection is active, look at the source and destination components. If they are both active, the non-modal connection is too, since all their parent components must be active.

The initial set \( TN^i_0 \) is the set of mode transitions in the system instance component \( C_0 \):

\[
TN^i_0 = \begin{cases} 
\emptyset & \text{if } C_0 \text{ has no mode transitions} \\
\{t_{n_0}, \ldots, t_{n_j}\} & \text{if } C_0 \text{ has } j > 0 \text{ mode transitions}
\end{cases}
\]

The individual transitions \( t_n \) are constructed from the mode transition in \( C_k \). For each combination of triggering port and mode transition from mode \( m_s \) to mode \( m_d \), determine the trigger sources \( TG \). Then, add all transitions: \( \{ (T_{m_s}) \} \times TG \times \{ (T_{m_d}) \} \).

\( TN_0 \) is the set of transitions where the source SOM is reachable from the initial SOM.

Adding SOM transitions during processing component \( C_k \) requires careful consideration of the possible cases. At this point, we have created these sets:

- \( SOM_{k-1} \)
- \( TN_{k-1} \)
- \( SOM_k^i \)

First, consider all SOMs \( som_i \in SOM_{k-1} \) for which \( C_k \) is inactive and transitions \( t_n = (som_i, tg_l, som_j) \in TN_{k-1} \). We disregard transitions where the trigger \( tg_l \) is a port of \( C_k \), because it is inactive and thus cannot emit the trigger event. If \( C_k \) is inactive in the target SOM \( som_j \) too, we add the following mode transition to \( TN_k^i \) for each mode \( m \) in \( C_k \):

\[
(som_i^{k-1} \times \{ \bot^k_m \}, tg_l, som_j^{k-1} \times \{ \bot^k_m \})
\]

Figure 2.1 depicts this situation.
If the state of $C_k$ changes to active in $som_j$ as shown in Figure 2.2 and Figure 2.3, we add the following transitions to $TN_k^i$:

$$
\left\{
\begin{aligned}
(som_k^{i-1} \times \{\bot_k^m\}, tg_i, som_j^{k-1} \times \{\top_k^m\}) & \forall m \in C_k \\
(som_l^{i-1} \times \{\bot_k^m\}, tg_i, som_j^{k-1} \times \{\top_k^m\}) & \forall m \in C_k
\end{aligned}
\right.
$$

for policy resume

for policy restart

The AADL standard does not fully define the mode behavior in case of component activation and deactivation. We adopt the interpretation that a component activated during an SOM transition with trigger $tg$ does not simultaneously perform any internal mode transition, even if there is a transition triggered by $tg$. Similarly, we assume that a component deactivated in an SOM transition does not simultaneously perform an internal mode transition.
Now we consider the cases where $C_k$ is active in $som_i$. If $C_k$ is inactive in $som_j$, the component is deactivated in any SOM transition $(som_i, tg_l, som_j)$. If the component has only a single target mode state $\perp_m$ (i.e., it has a single mode only or restart activation semantics), we add this transition to $TN^k_i$ for each mode $m$ in $C_k$ (see Figure 2.4):

$$(som_i^{k-1} \times \{\top_m\}, tg_l, som_j^{k-1} \times \{\perp_m\})$$

When there are several target mode states, that is, $C_k$ has resume activation semantics, we add a transition for each mode $m$ in $C_k$ (see Figure 2.5):

$$(som_i^{k-1} \times \{\top_m\}, tg_l, som_j^{k-1} \times \{\perp_m\})$$

In this case, the deactivated mode state stores the last active mode so it can be used in a SOM transition that reactivates $C_k$ (see Figure 2.2).

The remaining cases involve transitions where $C_k$ remains active in the target SOM.
Figure 2.6: Active Component $C_k$ Without Internal Transition

If $C_k$ does not contain any mode transition triggered by $tg_l$, $C_k$ does not change its mode in the SOM transition. For each mode $m$ of $C_k$, we add the following mode transition to $TN^k$ (see Figure 2.6):

$$(som_i^{k-1} \times \{\top_m^k\}, tg_l, som_j^{k-1} \times \{\top_m^k\})$$

Trigger $tg_l$ cannot be inactive, because it is active in $som_i$ and $C_k$ is active.

If $C_k$ contains a mode transition $(m, tg_l, m')$, we add the following transition to $TN^k$:

$$(som_i^{k-1} \times \{\top_m^k\}, tg_l, som_j^{k-1} \times \{\top_m^{k'}\})$$

For all modes $m$ of $C_k$ that have no outgoing transition triggered by $tg_l$, we add the following transition to $TX^k$:

$$(som_i^{k-1} \times \{\top_m^k\}, tg_l, som_j^{k-1} \times \{\top_m^k\})$$

Figure 2.7 illustrates this case for an internal mode transition between $m_0$ and $m_1$ in component $C_k$.

In the last remaining case, $C_k$ contains a mode transition $(m, tg, m')$ with a trigger that is not part of any SOM transition between $som_i$ and $som_j$. For each such mode transition, we add the following SOM transition to $TN^k$, if the trigger $tg$ is active in $som_i$ (see Figure 2.8):

$$(som_i^{k-1} \times \{\top_m^k\}, tg, som_j^{k-1} \times \{\top_{m'}^k\})$$

Note that each set $TN_k$ (except the last one) may contain transitions that reference a trigger source in a component $C_j$, $j > k$ that has not yet been processed. We consider such a transition as active when evaluating SOM reachability. When processing $C_j$, such a transition will be ignored if it starts in an SOM where $C_j$ is inactive. Transitions that refer to triggers originating outside the system instance are considered to be always active.
Figure 2.7: Active Component $C_k$ with Internal Transition $m_0 \rightarrow m_1$

Figure 2.8: New Trigger $tg$ in $C_k$ with Internal Transition $m_0 \rightarrow m_1$
2.3.2 Transitions for Modal Connections

A modal connection can be inactive even if all components in the model are active. For non-modal connections, it is sufficient to check that the originating component of a trigger is active. For modal connections, we must also check if all segments of the connection that transports the trigger event are active in an SOM. Furthermore, for a particular transition to be active, several connections might have to be active. This situation occurs when the same trigger event triggers multiple component mode transitions simultaneously. Which transitions happen simultaneously depends on which connections are active in an SOM. The algorithm must be extended to record the set of connections that must be active for an SOM transition to be possible.

It is possible for the same event to trigger mode transitions in multiple components via a single connection. Listing 2.1 and Figure 2.9 illustrate the situation where transitions are triggered in a component and its subcomponent by the same external event. The connection segment to the outer component is always active, whereas the connection segment to the inner component is active only if the outer component is in mode $m_0$. The instance model for Top.i contains a single connection instance $ci$ that consists of these two segments. Following the previous definition of when a connection instance is active, $ci$ is active only if the outer component is mode $m_0$, so there are two possible SOMs:

- $(\top, \top, m_0, \top, m_0)$
- $(\top, \top, m_1, \top, m_1)$

Additional trigger events result in no further mode transitions because $ci$ remains inactive. This explanation is somewhat unsatisfactory, because triggering a mode transition in the outer component does not seem to require the continuation of the connection into outer’s subcomponents. As an alternative, we can treat event delivery to a mode transition and a subcomponent equivalent to fan-out, that is, connections to several subcomponents from the same parent component port. In the instance model, fan-out results in multiple independent connection instances, one for each destination subcomponent.

In the following example, we adopt this view and consider partial connection instances for delivering trigger events. The relevant partial connections are the prefixes of a connection instance that lead to a mode transition. We refer to such prefixes as trigger connections. A trigger connection is active if and only if each of its segments is active. The example model, then, has the following two trigger connections where $tc_1$ is always active and $tc_2$ is active if component outer is in mode $m_0$:

$$tc_1 = (\text{Top.i.c})$$
$$tc_2 = (\text{Top.i.c}, \text{Top.i.outer.c})$$

The system can now reach the following SOMs when a sequence of trigger events arrives at the external port:

- $(\top, \top, m_0, \top, m_{10})$
- $(\top, \top, m_1, \top, m_{11})$
- $(\top, \top, m_{10}, \top, m_{11})$

After the first trigger event, the SOM alternates between these two SOMs:

- $(\top, \top, m_1, \top, m_{11})$
- $(\top, \top, m_{10}, \top, m_{11})$
system Top
end Top;

system implementation Top.i
    subcomponents
        outer: S.outer;
    connections
        c: port tg -> outer.tg;
    end Top.i

system S
    features
        tg: in event port;
    end S;

system implementation S.outer
    subcomponents
        inner: system S.inner;
    connections
        c: port tg -> inner.tg in modes (m0);
    modes
        m0: initial mode; m1: mode;
        t0: m0 -[tg]-> m1;
        t1: m1 -[tg]-> m0;
    end s.outer;

system implementation S.inner
    modes
        m10: initial mode; m11: mode
        t10: m10 -[tg]-> m11;
    end inner;

Listing 2.1: Mode Transitions on Several Levels

Figure 2.9: Graphical View of the Model from Listing 2.1
We now define an SOM transition as a 4-tuple \((s, tg, d, tc) \in \text{SOM}_k \times TG \times \text{SOM}_k \times 2^{TC}\), where TG is the set of all trigger sources in the model and TC is the set of trigger connections. The set tc consists of all trigger connections that must be active for the transition to be enabled.

All cases from the previous section must be modified as follows except where a trigger \(tg\) triggers an SOM transition in \(\text{SOM}_k\) and a mode transition in \(C_k\):

- When adding transitions for component \(C_k\), we disregard all transitions that are not enabled, because either the originating component is inactive or at least one of its trigger connections in \(tc\) is inactive in the source SOM.

- When a mode transition \(t\) in \(C_k\) has a new active trigger, we add transitions for this trigger and each active trigger connection by which the trigger event can reach \(t\). The same event could potentially reach a mode transition via more than one connection, and each of these connections can be active independent of the others.

Special consideration is required when both of these things are present:

- an SOM transition \((\text{som}_i, tg, \text{som}_j, tc) \in \text{TN}_{k-1}\)
- a transition \((m, tg, m')\) in component \(C_k\)

Assume \(tg\) reaches \(C_k\) via a trigger connection \(c\). If \(c\) is active in \(\text{som}_i \times \{\top_m\}\), we must add this SOM transition:

\[
(\text{som}_i \times \{\top_m\}, tg, \text{som}_j \times \{\top_{m'}\}, C \cup c)
\]

In contrast, when \(c\) is not active, we add this transition:

\[
(\text{som}_i \times \{\top_m\}, tg, \text{som}_j \times \{\top_m\}, C)
\]

In general, it is unknown which connections are active before all components have been processed, except if \(c\) is a non-modal connection:

- If \(c\) is non-modal, it is active, and only the first candidate transition must be added.
- If \(c\) is modal, we must add both transitions as candidates and disregard transitions that will become inactive when additional components are processed.

The SOM transitions of the example model are

\[
\text{TN} = \{(\text{som}_0, tg, \text{som}_1, \{cn_1\}),
(\text{som}_1, tg, \text{som}_2, \{cn_1, cn_2\}),
(\text{som}_2, tg, \text{som}_1, \{cn_1\})\}
\]

where
\[
\text{som}_0 = (\top_m, \top_{m_0}, \top_{m_{10}}),
\text{som}_1 = (\top_m, \top_{m_1}, \top_{m_{11}}),
\text{som}_2 = (\top_m, \top_{m_0}, \top_{m_{11}})
\]

Once all \(N\) components in the model are processed, the last set of SOM transitions \(\text{TN}_N\) may still contain SOM transition candidates that must be discarded before determining final SOM reachability. At this point, all the remaining transitions are active, which implies that all connections that occur in a transition are active. This means that we can remove any SOM transition that is dominated by another transition. A transition \(tn = (\text{som}_i, tg, \text{som}_j, C)\) dominates another transition \(tn' = (\text{som}_i, tg, \text{som}_k, C')\), if \(C' \subset C\).

Note that adding transition candidates can potentially lead to a combinatorial explosion of transitions, most of which may be removed only in this last step.
2.4 Testing SOM Reachability

An SOM can be discarded if it is not reachable from the initial SOM. The initial SOM $\text{s}om_0$ is the SOM that consists of initial component modes only, that is, each mode state in $\text{s}om_0$ is one of the following:

- $\top_m$ for an active component
- $\bot_m$ for an inactive component

The initial SOM is, of course, reachable. Another SOM $\text{s}om'$ is reachable if $\text{TN}_k$ contains a transition from a reachable mode to $\text{s}om'$.

Processing parent components before their subcomponents guarantees that an unreachable SOM cannot become reachable by extending it with modes from subsequent components. Adding a subcomponent cannot influence whether a component is active or activate an inactive transition. Therefore, we can discard unreachable SOMs after processing each component to keep the number of candidate SOMs as small as possible.

2.5 The Full Algorithm

Putting everything together, we arrive at the algorithm shown in this section. The first version (see Figure 2.10) is simpler but creates the same transitions multiple times, whereas the second version (see Figure 2.11) creates each transition only once.

In the pseudocode, we use the notation $l[k]$ to extract the k-th element from a list $l$; and $(t; n)$ to append an element $n$ to a tuple $t$:

$$\quad \text{if } t = (e_1, e_2, \ldots, e_n), \text{ then } (t; e) = (e_1, e_2, \ldots, e_n, e)$$

For set comprehension, we use an abbreviated notation. For example, if the variable $a$ is defined elsewhere, the notation

$$\quad \{(b, c) \mid \exists (a, b, c) \in \text{SET}\}$$

is short for

$$\quad \{(b, c) \mid \exists x, b, c : (x, b, c) \in \text{SET} \land x = a\}.$$
\[ \text{Input:} \ \text{The list } \text{CS of component instances as visited by pre-order traversal of an AADL instance model } M \]

\[ \text{Output:} \ \text{The tuple } (\text{SOM}, i, T N), \ \text{where SOM is the set of reachable system operation modes in } M, \ i \ \text{is the initial SOM, and } T N \ \text{is the set of transitions between them} \]

\[ \text{for } k \leftarrow 0 \ \text{to } \text{len}(\text{CS}) - 1 \ \text{do} \]
\[ C \leftarrow \text{CS}[k] \]
\[ \text{if } k = 0 \ \text{then} \]
\[ \text{SOM}_0 \leftarrow \text{MS}_0(C) \]
\[ i \leftarrow \text{Tm}(C) \]
\[ T N_0 \leftarrow \{ (T_m, t g, T_{m'}, \{t c\}) \mid \exists (m, t g, m', t c) \in \text{trans}(C) \} \]
\[ \text{else} \]
\[ \text{SOM}_k \leftarrow \bigcup_{s \in \text{SOM}_{k-1}} s \times \text{MS}_k(C, s) \]
\[ i \leftarrow (i; \text{state}(i, \text{init}(C))) \]
\[ T N_k \leftarrow \emptyset \]
\[ \text{forall} \ (p s, p t g, p d, T C) \in T N_{k-1} \ \text{do} \]
\[ \text{PTGS} \leftarrow \{ t g \mid \exists (p s, t g, p d, T C) \in T N_{k-1} \} \]
\[ \text{forall} \ (m s, t g, m d, t c) \in \text{trans}(C) \ \text{do} \]
\[ \text{TGS} \leftarrow \{ t g \mid \active(p s, C) \land \exists (m, t g, m', c) \in \text{trans}(C) \} \]
\[ s \leftarrow (p s; \text{state}(p s, m s)) \]
\[ \text{if } p t g = t g \land \active(s, \{p t g\} \cup T C) \land \active(s, \{C, t g, t c\}) \ \text{then} \]
\[ t n \leftarrow (s, t g, (p d; \text{state}(p d, m d)), T C \cup \{t c\}) \]
\[ T N_k \leftarrow T N_k \cup \{t n\} \]
\[ \text{else} \]
\[ \text{if } p t g \notin \text{TGS} \land \exists (s, \{p t g\} \cup T C) \ \text{then} \]
\[ t n \leftarrow (s, t g, (p d; \text{state}(p d, m d)), T C) \]
\[ T N_k \leftarrow T N_k \cup \{t n\} \]
\[ \text{if } t g \notin \text{TGS} \land \exists (s, \{C, t g, t c\}) \ \text{then} \]
\[ t n \leftarrow (s, t g, (p s; \text{state}(p s, m d)), \{t c\}) \]
\[ T N_k \leftarrow T N_k \cup \{t n\} \]

\[ (\text{SOM}_k, T N_k) \leftarrow \text{removeUnreachable}(\text{SOM}_k, i, T N_k) \]

\[ \text{// remove dominated transitions} \]
\[ T N_k \leftarrow T N_k \setminus \{(s, t g, d, T C) \in T N_k \mid \exists (s, t g, d, T C') \in T N_k : T C \subset T C' \} \]

\[ (\text{SOM}, T N) \leftarrow \text{removeUnreachable}(\text{SOM}_k, i, T N_k) \]

\[ \text{return} \ (\text{SOM}, i, T N) \]

\[ \text{Figure 2.10: Reachability Algorithm} \]
**Input**: The list CS of component instances as visited by pre-order traversal of an AADL instance model M

**Output**: The tuple (SOM, i, TN), where SOM is the set of reachable system operation modes in M, i is the initial SOM, and TN is the set of transitions between them.

```
for k ← 0 to len(CS) − 1 do
    C ← CS [k]
    if k = 0 then
        SOM0 ← MS0 (C)
        i ← Tinit (C)
        TN0 ← {{(m, tg, m'), {tc}} | ∃ (m, tg, m', tc) ∈ trans(C)} // root component
    else
        SOMk ← ∪ s ∈ SOMk−1, s × MS (C, s)
        i ← (i; state (i, init (C))
        TNk ← ∅
        forall ps ∈ SOMk do
            TGS ← if active (ps, C) then {tg | ∃ (m, tg, m', c) ∈ trans(C)} else ∅ // triggers used in C
            PTGS ← {tg | ∃ (ps, tg, pd, TC) ∈ TNk−1} // previously used triggers
            forall (ps, ptg, pd, TC) ∈ TNk−1 do // propagate transitions from TNk−1 to TNk
                if ptg ∈ TGS then
                    forall (ms, tg, md, tc) ∈ trans(C) such that tg = ptg do
                        s ← (ps; state (ps, ms))
                        if active (s, {tg} ∪ TC) then
                            if active (s, {tg, tc}) then // create merged transition (Fig. 2.7)
                                tn ← (s, tg, (pd; state (pd, md)), TC ∪ {tc})
                            else // copy transition (Fig. 2.6)
                                tn ← (s, tg, (pd; state (pd, ms)), TC)
                        TNk ← TNk ∪ {tn}
                    else
                        if active (s, {tg, tc}) then // like new trigger (Fig. 2.8)
                            tn ← (s, tg, (ps; state (ps, md)), {tc})
                            TNk ← TNk ∪ {tn}
                    else // copy transition (Fig. 2.6)
                        forall m ∈ modes (C) do
                            s ← (ps; state (ps, m))
                            if active (s, {ptg} ∪ TC) then
                                tn ← ((s, ptg, (pd; state (pd, m), TC)
                            TNk ← TNk ∪ {tn}
                        forall (ms, tg, md, tc) ∈ trans(C) do // transitions for new triggers (Fig. 2.8)
                            s ← (ps; state (ps, ms))
                            if tg ∉ PTGS ∧ active (s, {C, tg, tc}) then
                                tn ← (s, tg, (ps; state (ps, md)), {tc})
                            TNk ← TNk ∪ {tn}
                        (SOMk, TNk) ← removeUnreachable (SOMk, i, TNk)
        // remove dominated transitions
        TNk ← TNk \ {{s, tg, d, TC} ∈ TNk | ∃ (s, tg, d, TC') ∈ TNk : TC ⊆ TC'}
        (SOM, TN) ← removeUnreachable (SOMk, i, TNk)
    return (SOM, i, TN)
```

*Figure 2.11: Reachability Algorithm Without Duplicate Transition Creation*
3 Example Models

In this section, we show four example models and their resulting SOMs and SOM transitions. Some examples have variants to demonstrate how the result changes when components or connections are active in different modes.

The analysis results are shown in form of a diagram to be read as follows:

- Each rectangle represents a model component and is labeled with a component’s name. The components are listed in the order in which they are processed by the reachability analysis algorithm.
- Inside each component, we show possible mode states as ovals (active or inactive, and associated component mode). The same mode state may occur multiple times if it is part of multiple SOMs. Mode states are connected with read lines to form one or more tree structures.
- The reachable SOMs of the system are the paths from the root to a leaf of a tree. Such a path selects a mode state for each component in an SOM. The initial SOM is the single path where all mode states are filled.

The SOM transitions are shown as arrows between the tree leaf nodes in the component at the bottom of the diagram. Each transition is labeled with the fully qualified name of the trigger event port in the instance model.

3.1 Non-Modal Connections

In the first three example models, all connections are non-modal.

Example 1

The first example (Listing 3.1 and Figure 3.1) has a modal system instance with mode transitions triggered by

- an external event $e_0$
- a subcomponent $a$ emitting a trigger event $e_1$
- a subcomponent $b$ with modes and transitions triggered by the event from $a$

In the first variant (3.1a), subcomponents $a$ and $b$ are always active. In the next two variants (3.1b and 3.1c), one subcomponent is active only in mode $m_0$. The final variant (3.1d) prevents the two subcomponents from being active simultaneously so a mode change to $m_{11}$ in $b$ is never triggered.

Note that for the second variant (3.1b), the components are processed in a different order compared to the other variants. This is an artifact of the model instantiation as implemented in OSATE: Refined subcomponents are inserted into the instance model before inherited components.
package Example_1
public
system S
  features
    e0: in event port;
  end S;
system implementation S.i0
  subcomponents
    a: system A;
    b: system B;
  connections
    c: port a.e1 -> b.e1;
  modes
    m0: initial mode;
    m1: mode;
    m0 -[e0]-> m1;
    m1 -[e0]-> m0;
end S.i0;

system implementation S.i1 extends S.i0
  subcomponents
    a: refined to system A in modes (m0);
end S.i1;

system implementation S.i2 extends S.i0
  subcomponents
    b: refined to system B in modes (m0);
end S.i2;

system implementation S.i3 extends S.i0
  subcomponents
    a: refined to system A in modes (m0);
    b: refined to system B in modes (m1);
end S.i3;

system A
  features
    e1: out event port;
end A;

system B
  features
    e1: in event port;
  modes
    m10: initial mode;
    m11: mode;
    m10 -[e1]-> m11;
    m11 -[e1]-> m10;
end B;
end Example_1;

Listing 3.1: Example 1
(a) System $S_{i0}$

(b) System $S_{i1}$

(c) System $S_{i2}$

(d) System $S_{i3}$

Figure 3.1: SOMs and SOM Transitions for Example 1
Example 2

In this example (Listing 3.2 and Figure 3.2), we show a system implementation with two modes $m_0$ and $m_1$ and with mode transitions back and forth that are triggered by an event from a subcomponent $a$.

In the first variant (3.2a), $a$ is always active, so both transitions are possible. The second variant (3.2b) has $a$ active only in the initial mode, so the system can reach, but never leave, mode $m_1$. In the third variant (3.2c), $a$ is not active in the initial mode, such that no mode transition can occur.

```java
package Example_2

public
    system S
end S;

system implementation S.i0
    subcomponents
        a: system A;
    modes
        m0: initial mode;
        m1: mode;
        m0 -[a.e1]--> m1;
        m1 -[a.e1]--> m0;
end S.i0;

system implementation S.i1 extends S.i0
    subcomponents
        a: refined to system A in modes (m0);
end S.i1;

system implementation S.i2 extends S.i0
    subcomponents
        a: refined to system A in modes (m1);
end S.i2;

system A
    features
        e1: out event port;
end A;
end Example_2;
```

Listing 3.2: Example 2
Figure 3.2: SOMs and SOM Transitions for Example 2
**Example 3**

In this example (Listing 3.3 and Figure 3.3), the system instance has an array of subcomponents with two modes each. All mode transitions in the array components are triggered by the same external event; transitions happen simultaneously in all array components, resulting in two SOMs.

```java
package Example_3
public
  system S
  end S;

  system implementation S.i0
  subcomponents
    a: system A;
    b: system B.i[4];
  connections
    c: port a.e1 -> b.e1 {ConnectionPattern => ((One_To_All));}
  end S.i0;

  system A
  features
    e1: out event port;
  end A;

  system B
  features
    e1: in event port;
  end B;

  system implementation B.i
  modes
    m10: initial mode;
    m11: mode;
    m10 -[e1]-> m11;
    m11 -[e1]-> m10;
  end B.i;
end Example_3;
```

*Listing 3.3: Example 3*
Figure 3.3: SOMs and SOM Transitions for Example 3
3.2 Modal Connections

The final example uses modal connections.

Example 4

The fourth example (Listing 3.4 and Figure 3.4) has

- a modal system instance with mode transitions triggered by an external event $e_0$
- a subcomponent $a$ emitting a trigger event $e_1$
- subcomponents $b_1$ and $b_2$ with modes and transitions triggered by the event from $a$

The model has three variants, where the connections from $a$ to $b_1$ and $b_2$ are

- always active ($S.i0$, Figure 3.4a)
- active in mode $m_1$ only ($S.i1$, Figure 3.4b)
- active in different modes ($S.i2$, Figure 3.4c)

When the connections are active in different modes (variant 3.4c), the mode transitions in subcomponents $b_1$ and $b_2$ are no longer simultaneous, resulting in more reachable SOMs than in the first two variants.
package Example_4

public

system S
features
  e0: in event port;
modes
  m0: initial mode;
  m1: mode;
  m0 -[e0] -> m1;
  m1 -[e0] -> m0;
end S;

system implementation S.i0
subcomponents
  a: system A;
  b1: system B;
  b2: system B;
connections
  c1: port a.e1 -> b1.e1;
  c2: port a.e1 -> b2.e1;
end S.i0;

system implementation S.i1 extends S.i0
connections
  c1: refined to port in modes (m1);
  c2: refined to port in modes (m1);
end S.i1;

system implementation S.i2 extends S.i0
connections
  c1: refined to port in modes (m1);
  c2: refined to port in modes (m0);
end S.i2;

system A
features
  e1: out event port;
end A;

system B
features
  e1: in event port;
modes
  m10: initial mode;
  m11: mode;
  m10 -[e1] -> m11;
  m11 -[e1] -> m10;
end B;
end Example_4;

Listing 3.4: Example 4
Figure 3.4: SOMs and SOM Transitions for Example 4
4 Conclusion and Future Work

In this report, we described an algorithm that determines the system operation modes and transitions between them for AADL models under a few simplifying assumptions. In this section, we give a brief overview of how the algorithm can be extended to include a wider set of AADL models. As future work, we want to implement all these extensions and potentially integrate the analysis with the instantiation process in OSATE.

Derived Modes

It is also possible to extend the algorithm to handle models with derived modes. If a component \( C \) with derived modes is active, we create its mode state as \( \top \bar{m} \), and the associated mode \( \bar{m} \) is computed from the mode mapping in the AADL model. If \( C \) is inactive, its mode state is \( \bot \). We cannot associate a single mode with this state because \( C \)'s mode at resumption is determined by the resuming mode in the parent component. As a result, the resumption policy property does not apply to \( C \). This straightforward change is already part of our Java implementation.

Internal and Processor Features

The next assumption concerns the origin of trigger events. So far we assumed that these start at a port of a component. However, AADL also allows internal features as the source of an event.

In AADL, an internal feature \( \text{if} \) can be referenced as \( \text{self}.\text{if} \) in a mode transition or a connection declaration. For the purposes of analyzing SOM reachability, such a port is equivalent to a subcomponent port, where the subcomponent is left out of the model. We can, therefore, preprocess the AADL model, replace each internal feature with a new subcomponent that has a single event (or event data) port, and use the new port wherever the internal port occurs. These additional components are non-modal and have only an initial mode, such that their mode states can simply be removed from the SOMs in the algorithm’s output.

Similarly, an AADL processor feature \( \text{pf} \) can be referenced as \( \text{processor}.\text{pf} \) in a mode transition or connection declaration. It acts as a proxy for a feature of the processor the component is bound to, such that the actual port is determined by the value of property \( \text{Actual Processor Binding} \). In the reachability analysis, a processor feature can be handled by extending trigger connections that end at a processor feature with a segment that ends at the actual processor’s port. If the processor binding is modal, multiple extended trigger connections must be created, one for each binding, and the added segment must be modal. The analysis algorithm itself does not need to be modified.

Multiple Synchronization Domains

In an AADL model with multiple synchronization domains \( D_j, 0 \leq j \leq n \), each domain is a synchronized system on its own. For the SOM reachability analysis, we must consider events that originate in one domain and trigger mode transitions in another. Events that are transmitted between domains can have arbitrary delays, making it impossible, in general, to derive constraints on the order of such events. It should be possible to analyze each domain separately, treating inter-domain events as external events for purposes of the analysis, and then merging the results (SOM, \( i, TN \)) into a combined transition system. Working out the details of how to merge the results is left for future work.
A Appendix

In this appendix, we describe the data model and Java implementation of the SOM reachability analysis.

A.1 Data Model

The data model for the analysis implementation is defined as an Ecore model using the Eclipse Modeling Framework (EMF) [4]. Figure A.2 shows a class diagram for the data model.

The top-level class is the SOMGraph, which contains the data objects created for the analysis, and the result of the analysis is the fully filled SOMGraph data structure, from which the reachable SOMs and SOM transitions can be extracted.

SOMs are represented as a tree of SOMNodes, where each node represents a mode state of a component in the instance model. The SOMNodes for a component are owned by a SOMLevel object that references the component instance. There is one level per component instance. The levels are organized in a list that is owned by the graph object.

With the two subclasses ActiveNode and InactiveNode, we represent mode states as follows:

<table>
<thead>
<tr>
<th>Mode State</th>
<th>Class</th>
<th>Referenced Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>ActiveNode</td>
<td></td>
<td>Active component without modes</td>
</tr>
<tr>
<td>⊤_m</td>
<td>ActiveNode</td>
<td>m</td>
<td>Active component in mode m</td>
</tr>
<tr>
<td>⊥</td>
<td>InactiveNode</td>
<td></td>
<td>Inactive component without modes</td>
</tr>
<tr>
<td>⊥_m</td>
<td>InactiveNode</td>
<td>m</td>
<td>Inactive component resuming in mode m</td>
</tr>
</tbody>
</table>

Each SOMNode object references its predecessor in the current SOM. The sequence of mode states in an SOM is the reverse of the sequence of SOMNodes starting from the last level and following the parent references to a node on the first level. The set $SOM^k$, then, is the set of all such paths that start on the level for component $C_k$.

The initial SOM is identified by the path that starts at the initial mode ($\text{initialNode}$) of the last level. The parent of an initial node in level $k$ is the initial node of level $k-1$ if $k > 1$.

Figure A.1 shows a notional example of the correspondence between the SOM graph and the components in the instance model. The gray nodes mark the initial SOM.

In addition to nodes, a level also contains transitions between nodes on this level. Each Transition object represents a transition or transition candidates between SOMs in this level, that is, a transition’s source ($\text{src}$) and destination ($\text{dst}$) nodes are contained in the same level as the transition itself. The transitions in the level for a component instance $C_k$ are the elements of $TN^k$. Each transition references the set of connections (reference connections) that must be active for the transition to be enabled.

Each transition references the event that can trigger it. These events are represented by class Trigger. A trigger also references the component in which the event originates. All possible triggers are stored in a map TriggerMap that enables finding the trigger that corresponds to, for example, an event port instance. Because of the way EMF handles object identity, we cannot use features directly as keys in the map but instead must create separate key objects (TriggerKey). The data model supports regular component and internal features as triggers. However, internal features are not yet supported in the implementation, because they are not part of the AADL instance model in OSATE.
Each SOMNode has additional references to components and connections that are known to be inactive in the SOM that this node represents. These lists store the results of determining if a component or connection is inactive to avoid repeated calculation of the same results.
Figure A.2: Class Diagram for the Analysis Data Model
A.2 Java Implementation

This section includes the Java code that implements the SOM reachability algorithm. The OSATE plug-in includes additional code that is generated from the Ecore data model described in Section A.1, as well as code and configuration files needed for integration into the Eclipse plug-in framework.

The implementation is in a class `ReachabilityAnalyzer`. The field `graph` contains the data structure that is filled by an invocation of method `createSOMGraph` (line 23).

Compared to the algorithm presented in Chapter 2, an additional step is needed because the instance model does not contain the necessary trigger connections. The method `populateTriggers` (line 66) collects the event ports that can trigger a mode transition and also creates the trigger connections. These connections are added to the instance model EMF resource such that we can use EMF functionality to find connections that traverse a component to determine if the connection is active. Note that we create trigger connections also if no actual connection is involved, that is, if

- an external event triggers a mode transition in the system instance object
- a mode transition is triggered from a subcomponent event port that is not connected inside the subcomponent

The first level is then populated with nodes and transitions from the system instance in method `populateRootLevel` (line 204). The levels for each subsequent component are filled by a call to `populateNextLevel` in `processComponent` (line 55).

After levels for all components have been created, the final step is to delete any remaining dominated transition candidates from the last level. This happens in method `removeDominatedTransitions` (line 401).

For each component, we do the following:

- create a new level
- add nodes based on the component’s modes (`populateNodes`)
- add transitions based on transitions on the previous level and the component’s mode transitions (`populateTransitions`)
- mark reachable nodes
- remove unreachable nodes (`checkReachability` and `cleanup`)

The nodes for a component level are created in `populateNodes` (line 271). The implementation also handles derived nodes: If a component with derived modes is active, we create an `ActiveNode` with the mode value computed from the mode mapping in the AADL model. For an inactive component with derived modes, we create an `InactiveNode` without a mode. This is sufficient because the mode at resumption is determined by the resuming mode in the parent component.

If a component is inactive, we initially create `InactiveNodes` for each component mode. We later create the transitions based on the resumption policy so superfluous inactive nodes are unreachable.

The transitions for a level (for component $C_k$) are created in `populateTransitions` (line 343) in two steps:

1. For a given parent node $pn$, we add transitions to its child nodes based on $C_k$’s mode transitions, provided $C_k$ is active in both the SOM represented by $pn$ and the target SOM.
• If the mode transition in $C_k$ has the same trigger as a transition out of $pn$, the transitions are merged into one.

• If it does not, a new transition is created.

This step handles the situations shown in Figure 2.8 and one transition in Figure 2.7.

2. We propagate the transitions that leave $pn$ to the child nodes (see Figure 2.6). For merged transitions, we also add a modal trigger connection. For merged transitions that are triggered via a non-modal connection, we store the parent transition and child node in a map $skip$ to indicate that the parent transition should not be propagated to the child node because it is dominated by the child transition.

Note that no special handling for transitions between derived modes is necessary since such a component cannot contain mode transitions. All transitions between nodes representing derived modes are propagated from the previous level.

```java
public final class ReachabilityAnalyzer {
    private SOMGraph graph;
    private SOMLevel lastLevel;
    /** Container for trigger connections */
    private ComponentInstance tcHolder;
    /** Map component instances to the corresponding level in the SOM graph */
    private Map<ComponentInstance, SOMLevel> ci2sl = new HashMap<>();

    public ReachabilityAnalyzer() {
        // Create a reachability analyzer with default configuration
    }

    public void createSOMGraph(ComponentInstance root) {
        var rs = root.eResource().getResourceSet();
        var uri = makeURI(root);
        var res = rs.getResource(uri, false);
        if (res == null) {
            res = rs.createResource(uri);
        } else {
            res.unload();
        }
        graph = new SOMGraph();
        res.getContents().add(graph);

        // create dummy component to hold trigger connections
        tcHolder = InstanceFactory.eINSTANCE.createComponentInstance();
        root.eResource().getContents().add(tcHolder);

        // populate triggers and create trigger connections
        populateTriggers(root);

        // fill first level
        populateRootLevel(root);

        // process remaining components
        root.getComponentInstances().stream().forEach(this::processComponent);
    }

    private void processComponent(ComponentInstance c) {
        // Process component instances depth-first, pre-order
    }
}
```
private void populateTriggers(ComponentInstance root) {
    var visitor = new InstanceSwitch<Boolean>() {
        @Override
        public Boolean caseComponentInstance(ComponentInstance ci) {
            return false;
        }

        @Override
        public Boolean caseModeTransitionInstance(ModeTransitionInstance mt) {
            for (var f : mt.getTriggers()) {
                Iterator<ConnectionReference> crIter = CrossReferenceUtil.getInverse(
                    InstancePackage.eINSTANCE.getConnectionReference_Destination(), f, f.eResource());
                if (f.getContainingComponentInstance() == mt.getContainingComponentInstance()) {
                    // triggered from outside
                    if (f.getContainingComponentInstance() instanceof SystemInstance) {
                        // f is a feature of the system instance that triggers a
                        // mode transition in the system instance itself
                        addTrigger(f);
                        addTriggerConnection(f, mt);
                    }
                    else {
                        // triggered from inside
                        if (!crIter.hasNext()) {
                            // f is subcomponent feature that is not connected inside the subcomponent
                            addTrigger(f);
                            addTriggerConnection(f, mt);
                        }
                    }
                }
            }
            while (crIter.hasNext()) {
                // trigger comes via a connection
                var cr = crIter.next();
                var conn = (ConnectionInstance) cr.getOwner();
                addTrigger(conn.getSource());
                addTriggerConnection(conn, cr, mt);
            }
        }

        @Override
        public Boolean defaultCase(EObject object) {
            return true;
        }
    }

    @Override
    public Boolean caseComponentInstance(ComponentInstance ci) {
        return false;
    }

    @Override
    public Boolean caseModeTransitionInstance(ModeTransitionInstance mt) {
        for (var f : mt.getTriggers()) {
            Iterator<ConnectionReference> crIter = CrossReferenceUtil.getInverse(
                InstancePackage.eINSTANCE.getConnectionReference_Destination(), f, f.eResource());
            if (f.getContainingComponentInstance() == mt.getContainingComponentInstance()) {
                // triggered from outside
                if (f.getContainingComponentInstance() instanceof SystemInstance) {
                    // f is a feature of the system instance that triggers a
                    // mode transition in the system instance itself
                    addTrigger(f);
                    addTriggerConnection(f, mt);
                }
                else {
                    // triggered from inside
                    if (!crIter.hasNext()) {
                        // f is subcomponent feature that is not connected inside the subcomponent
                        addTrigger(f);
                        addTriggerConnection(f, mt);
                    }
                }
            }
            while (crIter.hasNext()) {
                // trigger comes via a connection
                var cr = crIter.next();
                var conn = (ConnectionInstance) cr.getOwner();
                addTrigger(conn.getSource());
                addTriggerConnection(conn, cr, mt);
            }
        }
    }

    @Override
    public Boolean defaultCase(EObject object) {
        return true;
    }

    private void addTrigger(ConnectionInstanceEnd f) {
        Assert.isTrue(f instanceof FeatureInstance, "connection doesn't start with feature");
        var tk = new FeatureKey((FeatureInstance) f);
        graph.getTriggers().putIfAbsent(tk, tk.getTrigger());
    }

    private void addTriggerConnection(ConnectionInstance conn, ConnectionReference last, ModeTransitionInstance mt) {
        var crs = new ArrayList<ConnectionReference>();
        int tcLen = 0;
        for (var cr : conn.getConnectionReferences()) {
            tcLen += 1;
            if (cr == last) {
                crs.add(cr);
                addTriggerConnection(conn, cr, last);
```java
break;
}
for (var c : tcHolder.getConnectionInstances()) {
    if (c.getSource() != conn.getSource() || c.getDestination() != mt || c.getConnectionReferences().size() != tcLen) {
        int i = 0;
        while (i < tcLen) {
            var cr0 = c.getConnectionReferences().get(i);
            var cr1 = crs.get(i);
            if (cr0.getContext() == cr1.getContext() && cr0.getConnection() == cr1.getConnection()) {
                i++;
            }
        }
        if (i != tcLen) {
            return;
        }
    }
    var tc = InstanceFactory.eINSTANCE.createConnectionInstance();
    tc.setKind(ConnectionKind.MODE_TRANSITION_CONNECTION);
    tc.setSource(conn.getSource());
    tc.setDestination(mt);
    boolean modal = false;
    for (var cr : crs) {
        var r = InstanceFactory.eINSTANCE.createConnectionReference();
        r.setContext(cr.getContext());
        r.setConnection(cr.getConnection());
        tc.getConnectionReferences().add(r);
        modal = modal || !cr.getConnection().getAllInModes().isEmpty();
    }
    tcHolder.getConnectionInstances().add(tc);
}
/**
 * Create a trigger connection without connection instance.
 * @param f the triggering feature
 * @param mt the triggered mode transition
 */
private void addTriggerConnection(FeatureInstance f, ModeTransitionInstance mt) {
    for (var c : tcHolder.getConnectionInstances()) {
        if (c.getSource() == f && c.getDestination() == mt && c.getConnectionReferences().isEmpty()) {
            return;
        }
    }
    var tc = InstanceFactory.eINSTANCE.createConnectionInstance();
    tc.setSource(f);
    tc.setDestination(mt);
    tcHolder.getConnectionInstances().add(tc);
}
/**
 * Add modes and transitions for the root component
 * @param root
 */
private void populateRootLevel(ComponentInstance root) {
    var newLevel = createSOMLevel(root);
    var nodes = newLevel.getNodes();
    var transitions = newLevel.getTransitions();
    SOMNode initial = null;
    for (var iter = EcoreUtil.<EObject> getAllContents(root, true); iter.hasNext();)
        var eo = iter.next();
    if (visitor.doSwitch(eo)) {
        iter.prune();
    }
}
```

---

CMU/SEI-2024-TR-003 | SOFTWARE ENGINEERING INSTITUTE | CARNEGIE MELLON UNIVERSITY

[Distribution Statement A] Approved for public release and unlimited distribution.
```java
} else {
  // system instance has modes
  var somNodes = new HashMap<ModeInstance, SOMNode>();
  for (var m : root.getModelInstances()) {
    var n = createActiveNode(m);
    nodes.add(n);
    if (m.isInitial()) {
      Assert.isTrue(initial == null, “initial already set”);
      initial = n;
    }
    somNodes.put(m, n);
  }
  for (var mt : root.getModeTransitionInstances()) {
    for (var tc : mt.getDstConnectionInstances()) {
      var s = somNodes.get(mt.getSource());
      var d = somNodes.get(mt.getDestination());
      var end = tc.getSource();
      var tk = new FeatureKey((FeatureInstance) end);
      var tg = graph.getTriggers().get(tk);
      var t = createTransition(s, d, tg, tc);
      transitions.add(t);
    }
  }
  // mark reachable som nodes in new level
  Objects.requireNonNull(initial);
  newLevel.setInitialNode(initial);
  checkReachability(initial);
  cleanUp(newLevel);
  lastLevel = newLevel;
}

/**
 * @param c
 */
private void populateNextLevel(ComponentInstance c) {
  var newLevel = createSOMLevel(c);
  populateNodes(newLevel, c);
  populateTransitions(newLevel, c);
  checkReachability(newLevel.getInitialNode());
  cleanUp(newLevel);
  lastLevel = newLevel;
}

/**
 * @param level - the new level
 * @param c - the component to process
 */
private void populateNodes(SOMLevel level, ComponentInstance c) {
  var nodes = level.getNodes();
  SOMNode initial = null;
  for (var pn : lastLevel.getNodes()) {
    if (!pn.isReachable()) {
      continue;
    }
    // ci active in current partial SOM?
    var active = isActive(c, pn);
    var modes = c.getModeInstances();
    if (modes.isEmpty()) {
      // component has no modes
      // create one node for c
      var n = active ? createActiveNode(pn) : createInactiveNode(c, pn);
      nodes.add(n);
      if (pn == lastLevel.getInitialNode()) {
        Assert.isTrue(initial == null, “initial already set”);
        initial = n;
      }
    } else if (modes.get(0).isDerived()) {
      // component has derived modes
      if (active) {
```
// find the derived mode for the current SOM
// create at most one active node for c
for (var m : modes) {
  Assert.isTrue(active);
  var pm = getContainerMode(c, pn);
  if (m.getParents().contains(pm)) {
    var n = createActiveNode(m, pn);
    n.setDerived(true);
    level.getNodes().add(n);
    if (pn == lastLevel.getInitialNode() && pm.isInitial()) {
      Assert.isTrue(initial == null, "initial already set");
      initial = n;
    }
    break;
  }
}
else {
  // create one inactive node for c
  var n = createInactiveNode(c, pn);
  n.setDerived(true);
  nodes.add(n);
  var pm = getContainerMode(c, pn);
  if (pn == lastLevel.getInitialNode() && pm.isInitial()) {
    Assert.isTrue(initial == null, "initial already set");
    initial = n;
  }
  else {
    // component has regular modes
    // create one node per mode
    for (var m : modes) {
      var n = active ? createActiveNode(m, pn) : createInactiveNode(m, pn);
      nodes.add(n);
      var pm = getContainerMode(c, pn);
      if (pn == lastLevel.getInitialNode() && m.isInitial()) {
        Assert.isTrue(initial == null, "initial already set");
        initial = n;
      }
    }
    Objects.requireNonNull(initial);
    level.setInitialNode(initial);
  }
}
/**
 * Add transitions on the level for the new component.
 * @param level - the new level
 * @param c - the component instance to process
 */
void populateTransitions(SOMLevel level, ComponentInstance c) {
  var transitions = level.getTransitions();
  for (var pn : lastLevel.getNodes()) {
    Map<Transition, Set<SOMNode>> skip = new HashMap<>();
    Set<TriggerKey> ptks = pn.getOutTransitions()
      .stream()
      .map(tn -> tn.getTrigger().getKey())
      .collect(Collectors.toCollection(HashSet::new));
    for (var ptn : pn.getOutTransitions()) {
      skip.put(ptn, new HashSet<>());
    }
    if (pn.getChildren().get(0).isActive()) {
      // new component is active before transition
      for (var mt : c.getModeTransitionInstances()) {
        for (var tc : mt.getDstConnectionInstances()) {
          var tk = new FeatureKey((FeatureInstance) tc.getSource());
          var tg = graph.getTriggers().get(tk);
          if (ptks.contains(tk)) {
            for (var ptn : pn.getOutTransitions()) {
              skip.put(ptn, new HashSet<>());
            }
          }
        }
      }
    }
  }
}
if (!isModal(tc)) {
    // ptn is dominated by the merged transition
    skip.get(ptn).add(findChildNode(pn, mt.getSource()));
} else {
    // add transitions for trigger that occurs on the new level
    addTransition(pn, mt, tg, tc, transitions);
}

// propagate transitions for triggers from the previous level
for (var ptn : pn.getOutTransitions()) {
    propagateTransition(ptn, c, transitions, skip);
}

private void removeDominatingTransitions() {
    var toRemove = new ArrayList<Transition>();
    for (var n : lastLevel.getNodes()) {
        var byTrigger = n.getOutTransitions().stream()
            .collect(Collectors.groupingBy(Transition::getTrigger));
        for (var tns : byTrigger.values()) {
            if (tns.size() > 1) {
                tns.stream().filter(tn -> tns.stream().anyMatch(otn -> otn != tn &&
                        otn.getConnections().containsAll(tn.getConnections())))
                    .forEach(tn -> toRemove.add(tn));
            }
        }
    }
    for (var tn : toRemove) {
        tn.getSrc().getOutTransitions().remove(tn);
        tn.getDst().getInTransitions().remove(tn);
        tn.getTrigger().getTransitions().remove(tn);
        lastLevel.getTransitions().remove(tn);
    }

    // maybe an SOM is now unreachable
    lastLevel.getNodes().stream().forEach(n -> n.setReachable(false));
    checkReachability(lastLevel.getInitialNode());
    cleanUp(lastLevel);
}

private void propagateTransition(Transition ptn, ComponentInstance c, List<Transition> transitions, Map<Transition, Set<SOMNode>> skip) {
    var psn = ptn.getSrc();
    var pdn = ptn.getDst();
    var sn = psn.getChildren().get(0);
    var dn = pdn.getChildren().get(0);
    Assert.isTrue(psn.getChildren().size() == pdn.getChildren().size());
    if (sn.isActive() && !dn.isActive()) {
        // deactivating, need to interpret policy
        var policy = getResumptionPolicy(c);
        for (int i = 0; i < psn.getChildren().size(); i++) {
```java
var s = psn.getChildren().get(i);
if (!skip.get(ptn).contains(s) && isTransitionActive(s, ptn)) {
    SOMNode d;
    if (s.hasMode() && policy.get(s.getMode()) == ResumptionPolicy.RESTART) {
        d = findChildNode(pdn, getInitialMode(c));
    } else {
        d = pdn.getChildren().get(i);
    }
    var tn = createTransition(s, d, ptn);
    transitions.add(tn);
}
}

/* Add a transition to the new level based on a triggered a mode transition. */
/* @param pn - the parent SOM node */
/* @param mt - the mode transition */
/* @param tg - the trigger */
/* @param tc - the trigger connection */
/* @param transitions - the list of transitions on the current lager */
private void addTransition(SOMNode pn, ModeTransitionInstance mt, Trigger tg, ConnectionInstance tc, List<Transition> transitions) {
    if (!(tg instanceof FeatureTrigger)) {
        return;
    }
    SOMNode sn = findChildNode(pn, mt.getSource());
    Assert.notNull(sn, "no node for source");
    if (sn.isActive() && isTriggerActive(sn, tg, tc)) {
        SOMNode dn = findChildNode(pn, mt.getDestination());
        Assert.notNull(dn, "no node for destination");
        Assert.isTrue(dn.isActive(), "dst not active");
        var tn = createTransition(sn, dn, tg, tc);
        transitions.add(tn);
    }
}

/* Merge a new transition and a parent transition if they are triggered by the same event. */
/* @param ptn - the parent transition */
/* @param mt - the mode transition */
/* @param tg - the trigger */
/* @param tc - the trigger connections */
/* @param transitions - the list of transitions on the current lager */
private void mergeTransition(Transition ptn, ModeTransitionInstance mt, Trigger tg, ConnectionInstance tc, List<Transition> transitions) {
    if (!((tg instanceof FeatureTrigger))) {
        return;
    }
    SOMNode psn = ptn.getSrc();
    SOMNode pdn = ptn.getDst();
    SOMNode sn = findChildNode(psn, mt.getSource());
    Assert.notNull(sn, "no node for source");
    if (isTriggerActive(sn, tg, tc)) {
        SOMNode dn = findChildNode(pdn, mt.getDestination());
        Assert.notNull(dn, "no node for destination");
        Assert.isTrue(dn.isActive(), "trying to merge with inactive destination");
        var tn = createTransition(sn, dn, tg, tc);
        tn.getConnections().addAll(ptn.getConnections());
        transitions.add(tn);
    }
}

/* Check if a trigger is active. */
```
private boolean isTriggerActive(SOMNode n, Trigger tg, ConnectionInstance tc) {
  if (n.getActiveComponents().contains(tg.getComponent())) {
    return false;
  }
  return tc == null || !n.getActiveConnections().contains(tc);
}

private boolean isTransitionActive(SOMNode n, Transition tn) {
  Trigger tg = tn.getTrigger();
  if (n.getActiveComponents().contains(tg.getComponent())) {
    return false;
  }
  var inactiveConns = n.getActiveConnections();
  return tn.getConnections().stream().noneMatch(cr -> inactiveConns.contains(cr.getOwner()));
}

private void checkReachability(SOMNode from) {
  from.setReachable(true);
  for (var t : from.getActiveTransitions()) {
    var d = t.getDest();
    if (!d.isReachable()) {
      checkReachability(d);
    }
  }
}

private void cleanUp(SOMLevel level) {
  var tns = (level.getActiveTransitions()).stream()
    .filter(tr -> !tr.getSrc().isReachable())
    .map(tr -> {
      // clean up bidi cross references
      tr.setSrc(null);
      tr.setDest(null);
      tr.setTrigger(null);
      return tr;
    }).toList();
  level.getActiveTransitions().removeAll(tns);
  var ns = (level.getActiveNodes()).stream().filter(n -> !n.isReachable()).toList();
  level.getActiveNodes().removeAll(ns);
}

private boolean isActive(ComponentInstance ci, SOMNode n) {
  * A trigger is active if the originating component is active and
  * the connection that transports the trigger is active.
  * @param n - the current SOM
  * @param tg - the trigger to check
  * @param tc - the connection via which the trigger enters the component, may be null
  * @return whether the trigger is active in the current SOM
  */
  private boolean isTriggerActive(SOMNode n, Trigger tg, ConnectionInstance tc) {
    if (n.getActiveComponents().contains(tg.getComponent())) {
      return false;
    }
    return tc == null || !n.getActiveConnections().contains(tc);
  }
  */
  private boolean isTransitionActive(SOMNode n, Transition tn) {
    Trigger tg = tn.getTrigger();
    if (n.getActiveComponents().contains(tg.getComponent())) {
      return false;
    }
    var inactiveConns = n.getActiveConnections();
    return tn.getConnections().stream().noneMatch(cr -> inactiveConns.contains(cr.getOwner()));
  }
  */
  private void checkReachability(SOMNode from) {
    from.setReachable(true);
    for (var t : from.getActiveTransitions()) {
      var d = t.getDest();
      if (!d.isReachable()) {
        checkReachability(d);
      }
    }
  }
  */
  private void cleanUp(SOMLevel level) {
    var tns = (level.getActiveTransitions()).stream()
      .filter(tr -> !tr.getSrc().isReachable())
      .map(tr -> {
        // clean up bidi cross references
        tr.setSrc(null);
        tr.setDest(null);
        tr.setTrigger(null);
        return tr;
      }).toList();
    level.getActiveTransitions().removeAll(tns);
    var ns = (level.getActiveNodes()).stream().filter(n -> !n.isReachable()).toList();
    level.getActiveNodes().removeAll(ns);
  }
  */
  private boolean isActive(ComponentInstance ci, SOMNode n) {
  }
```java
var pci = (ComponentInstance) ci.eContainer();
var pl = getSOMLevel(pci);
var pn = n;
while (pn.eContainer() != pl) {
    pn = pn.getParent();
} // pn is the som node for the containing component in the current som ending with n
// ci is active if the container is active and
// ci is active in the current mode of the container
return pn.isActive() && (ci.getInModes().isEmpty() || ci.getInModes().contains(pn.getMode()));
```

**Get the containing component’s mode in the current SOM.**

This requires that the modes for the containing component have already been entered into the graph.

* @param ci - the component instance to check
* @param n - the current leaf som node
* @return the mode of the component containing ci, null if no mode
*/

```java
private ModeInstance getContainerMode(ComponentInstance ci, SOMNode n) {
    var pci = (ComponentInstance) ci.eContainer();
    var pl = getSOMLevel(pci);
    var pn = n;
    while (pn.eContainer() != pl) {
        pn = pn.getParent();
    } // pn is the som node for the containing component in the current som ending with n
    Assert.isNotNull(pn);
    return pn.getMode();
}
```

**Create a new SOM level for a given component and add it to the SOM graph.**

* @param c - the component associated with the new level
* @return the new SOM level
*/

```java
private SOMLevel createSOMLevel(ComponentInstance c) {
    var newLevel = new SOMLevel();
    newLevel.setComponent(c);
    ci2sl.put(c, newLevel);
    graph.getLevels().add(newLevel);
    return newLevel;
}
```

**Get the SOM level for a component**

* @param c - the component
* @return the level for the given component
*/

```java
private SOMLevel getSOMLevel(ComponentInstance c) {
    return ci2sl.get(c);
}
```

**Create an active SOM node.**

* @param m - the mode for which to create the node
* @return the new node
*/

```java
private ActiveNode createActiveNode(ModeInstance m) {
    return createActiveNode(m, null);
}
```

**Create a node for an active component without modes**

* Inactive components and connections are the same as for the parent
* @param pn - the parent node
* @return the new node
*/

```java
private ActiveNode createActiveNode(SOMNode pn) {
    var n = new ActiveNode(pn);
    var icons = n.getInactiveConnections();
    return n;
}
```
/**
 * Create a node for an active component in a mode.
 * @param m - the mode for the new node
 * @param pn - the parent node
 * @return the new active node
 */
private ActiveNode createActiveNode(ModeInstance m, SOMNode pn) {
    var n = new ActiveNode(m, pn);
    var iconns = n.getInactiveConnections();
    if (pn != null) {
        iconns.addAll(pn.getInactiveConnections());
    }
    var c = m.getComponentInstance();
    var crIter = CrossReferenceUtil
        .getInverse(InstancePackage.eINSTANCE.getConnectionReference_Context(), c, c.eResource());
    while (crIter.hasNext()) {
        var cr = (ConnectionReference) crIter.next();
        var conn = (ConnectionInstance) cr.getOwner();
        if (conn.getKind() == ConnectionKind.MODE_TRANSITION_CONNECTION) {
            var ims = cr.getConnection().getAllInModes();
            if (!ims.isEmpty() && !ims.contains(m.getMode())) {
                iconns.add(conn);
            }
        }
    }
    return n;
}

/**
 * Create a node for an inactive component without modes.
 * @param c - the component
 * @param pn - the parent node
 * @return the new inactive node
 */
private InactiveNode createInactiveNode(ComponentInstance c, SOMNode pn) {
    var n = new InactiveNode(pn);
    fillInactiveNode(n, c, pn);
    return n;
}

/**
 * Create a node for an inactive component.
 * @param m - the mode after resumption
 * @param pn - the parent node
 * @return the new inactive node
 */
private InactiveNode createInactiveNode(ModeInstance m, SOMNode pn) {
    var n = new InactiveNode(m, pn);
    fillInactiveNode(n, c, pn);
    return n;
}

/**
 * Store the inactive components and connections for a new inactive node.
 * @param n - the new node
 * @param c - the component
 * @param pn - the parent node
 */
private void fillInactiveNode(InactiveNode n, ComponentInstance c, SOMNode pn) {
    var ics = n.getInactiveComponents();
    ics.addAll(pn.getInactiveComponents());
    ics.add(c);
    var iconns = n.getInactiveConnections();
    iconns.addAll(pn.getInactiveConnections());
```java
var crIter = CrossReferenceUtil
    .getInverse(InstancePackage.eINSTANCE.getConnectionReference(), c, c.eResource());
while (crIter.hasNext()) {
    var cr = (ConnectionReference) crIter.next();
    var conn = (ConnectionInstance) cr.getOwner();
    if (conn.getKind() == ConnectionKind.MODE_TRANSITION_CONNECTION) {
        iconns.add(conn);
    }
}

/**
 * Create a transition.
 * @param sn − the source SOM node of the new transition
 * @param dn − the destination SOM node of the new transition
 * @param tg − the trigger of the new transition
 * @param tc − the trigger connection
 * @return the new transition
 */
private Transition createTransition(SOMNode sn, SOMNode dn, Trigger tg, ConnectionInstance tc) {
    var tn = new Transition();
    tn.setSrc(sn);
    tn.setDst(dn);
    tn.setTrigger(tg);
    if (tc != null) {
        tn.getConnections().add(tc);
    }
    return tn;
}

/**
 * Create a transition based on a parent transition.
 * @param src − the source SOM node of the new transition
 * @param dst − the destination SOM node of the new transition
 * @param ptn − the parent transition
 * @return the new transition
 */
private Transition createTransition(SOMNode src, SOMNode dst, Transition ptn) {
    var t = new Transition();
    t.setSrc(src);
    t.setDst(dst);
    t.setTrigger(ptn.getTrigger());
    t.getConnections().addAll(ptn.getConnections());
    return t;
}

/**
 * Get the initial mode of a component.
 * @param c − the component
 * @return the initial mode of this component; null if the component has no modes.
 */
private ModeInstance getInitialMode(ComponentInstance c) {
    for (var m : c.getModeInstances()) {
        if (m.isInitial()) {
            return m;
        }
    }
    return null;
}

/**
 * Get the child of a given SOM node for a specific mode.
 * @param pn − the parent node
 * @param m − the mode
 * @return the child node for the mode
 */
private SOMNode findChildNode(SOMNode pn, ModeInstance m) {
    if (pn.getChildren().size() == 1) {
        return pn.getChildren().get(0);
    }
    for (var n : pn.getChildren()) {
        if (n.getMode() == m) {
            return n;
        }
    }
    return null;
}
```

/**
 * Get the resumption policy of a component.
 * @param c - the component
 * @return a map containing the resumption policy value for each component mode
 */
private Map<ModeInstance, ResumptionPolicy> getResumptionPolicy(ComponentInstance c) {
  if (c.getModeInstances().isEmpty()) {
    return Collections.emptyMap();
  }

  Map<ModeInstance, ResumptionPolicy> m2policy = new HashMap<>();
  ResumptionPolicy policy = ResumptionPolicy.RESTART;
  try {
    // try to get non modal value
    var nmv = ThreadProperties.getResumptionPolicy(c);
    if (nmv.isPresent()) {
      policy = nmv.get();
    }
    m2policy = new HashMap<>();
    for (var m : c.getModeInstances()) {
      m2policy.put(m, policy);
    }
  }
  catch (PropertyIsModalException e) {
    // get modal property value
    var p = ThreadProperties.getResumptionPolicyProperty(c);
    var ipa = c.getOwnedPropertyAssociations().stream()
      .filter(pa -> pa.getProperty() == p)
      .findFirst();
    Assert.isTrue(ipa.isPresent());
    var dpa = ((PropertyAssociationInstance) ipa.get()).getPropertyAssociation();
    int i;
    for (i = 0; i < dpa.getOwnedValues().size(); i++) {
      var mpv = dpa.getOwnedValues().get(i);
      for (var m : mpv.getInModes()) {
        for (var mi : c.getModeInstances()) {
          if (mi.getMode() == m) {
            var pe = mpv.getOwnedValue();
            pe = CodeGenUtil.resolveNamedValue(pe);
            m2policy.put(mi, ResumptionPolicy.valueOf(pe));
            break;
          }
        }
      }
    }
    var mpv = dpa.getOwnedValues().get(i - 1);
    if (mpv.getInModes().isEmpty()) {
      var pe = mpv.getOwnedValue();
      pe = CodeGenUtil.resolveNamedValue(pe);
      policy = ResumptionPolicy.valueOf(pe);
      m2policy = new DefaultedHashMap<ModeInstance, ResumptionPolicy>(policy, m2policy);
    }
  }
  return m2policy;
}

public void setConfiguration(ReachabilityConfiguration config) {
  this.config = config;
}

public ReachabilityConfiguration getConfiguration() {
  return config;
}

private URI makeURI(ComponentInstance root) {
  var uri = root.eResource().getURI();
  var fn = uri.segment(uri.segmentCount() - 1);
  uri = uri.trimSegments(1)
    .appendSegment("reports")
    .appendSegment("sosreachability")
    .appendSegment(fn);
  uri = uri.trimFileExtension().appendFileExtension("modemodel");
  return uri;
}

/**
 * @return the config
 */
public ReachabilityConfiguration getConfig() {
    return config;
}

/**
 * @param config the config to set
 */
public void setConfig(ReachabilityConfiguration config) {
    this.config = config;
}

/**
 * @return the graph
 */
public SOMGraph getGraph() {
    return graph;
}

/**
 * @return the lastLevel
 */
public SOMLevel getLastLevel() {
    return lastLevel;
}

/**
 * Cache if a trigger connection is modal
 */
private Map<ConnectionInstance, Boolean> tcModal = new HashMap<>();

/**
 * Determine if a trigger connection is modal.
 * The result is cached.
 * @param tc
 * @return
 */
private boolean isModal(ConnectionInstance tc) {
    if (tcModal.containsKey(tc)) {
        return tcModal.get(tc);
    }
    boolean modal = false;
    for (var cr : tc.getConnectionReferences()) {
        if (!cr.getConnection().getAllInModes().isEmpty()) {
            modal = true;
            break;
        }
    }
    tcModal.put(tc, modal);
    return modal;
}
References

URLs are valid as of the publication date of this document.


<table>
<thead>
<tr>
<th>1. AGENCY USE ONLY</th>
<th>2. REPORT DATE</th>
<th>3. REPORT TYPE AND DATES COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Leave Blank)</td>
<td>May 2024</td>
<td>Final</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE AND SUBTITLE</th>
<th>5. FUNDING NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachability of System Operation Modes in AADL</td>
<td>FA8702-15-D-0002</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6. AUTHORS</th>
<th>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lutz Wrage</td>
<td>Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8. PERFORMING ORGANIZATION REPORT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMU/SEI-2024-TR-003</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</th>
<th>10. SPONSORING/MONITORING AGENCY REPORT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI Administrative Agent AFSC/CMC/AZS 5 Elgin Street Hanscom AFB, MA 01731-2100</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11. SUPPLEMENTARY NOTES</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>12A. DISTRIBUTION/AVAILABILITY STATEMENT</th>
<th>12B. DISTRIBUTION CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified/Unlimited, DTIC, NTIS</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13. ABSTRACT (MAXIMUM 200 WORDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components in an AADL (Architecture Analysis and Design Language) model can have modes that determine which subcomponents and connections are active. Transitions between modes are triggered by events originating from the modeled system’s environment or other components in the model. Modes and transitions can occur on any level of the component hierarchy. The combinations of component modes (called system operation modes or SOMs) define the system’s configurations. It is important to know which SOMs can actually occur in the system, especially in the area of system safety, because a system may contain components that should not be active simultaneously, for example, a car’s brake and accelerator. This report presents an algorithm that constructs the set of reachable SOMs for a given AADL model and the transitions between them.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. SUBJECT TERMS</th>
<th>15. NUMBER OF PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL, Architecture Analysis and Design Language, system operation mode, SOM, reachability</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. PRICE CODE</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>17. SECURITY CLASSIFICATION OF REPORT</th>
<th>18. SECURITY CLASSIFICATION OF THIS PAGE</th>
<th>19. SECURITY CLASSIFICATION OF ABSTRACT</th>
<th>20. LIMITATION OF ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified</td>
<td>Unclassified</td>
<td>Unclassified</td>
<td>UL</td>
</tr>
</tbody>
</table>


CMU/SEI-2024-TR-003 | SOFTWARE ENGINEERING INSTITUTE | CARNEGIE MELLON UNIVERSITY 48

[Distribution Statement A] Approved for public release and unlimited distribution.