Interpretation of AADL Behavior Annex into synchronous formalism using SSA

Yue Ma, Jean-Pierre Talpin, Thierry Gautier

Interpretation of AADL Behavior Annex into synchronous formalism using SSA

Yue Ma Jean-Pierre Talpin Thierry Gautier
INRIA, Unité de Recherche Rennes-Bretagne-Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France
Email: {Yue.Ma, Jean-Pierre.Talpin, Thierry.Gautier}@irisa.fr

Abstract

This article focuses on the essence and distinctive features of the AADL behavioral aspects, for which we use the code generation infrastructure of the synchronous modeling environment SME. It introduces an effective method for transforming a behavior specification consisting of transitions and actions into a set of synchronous equations. We present an approach for this transformation using SSA as an intermediate formalism. This interpretation minimizes introducing new state variables and transitions.

1 Introduction

Nowadays, embedded systems are an integral part of safety critical systems in various domains, such as avionics, automotive and telecommunications. Typically, they have a long life cycle of development and maintenance. In this process, architecture design and early analysis of embedded systems are two of the major challenges for designers using modeling languages such as AADL [1] (Architecture Analysis and Design Language) to describe the systems.

AADL is a language which supports the modeling of component-based systems as an assembly of software components mapped onto execution platforms. The modeling aspect of system design activity is becoming increasingly essential, since it allows prototyping and experiments without necessarily having a physical implementation of the system. The Behavior Annex is proposed as an extension of AADL to offer a way to specify the behaviors of components. AADL allows a fast design entry and software/ hardware co-design. However, system validation and verification is a critical challenge. What we are seeking for is to use formal methods to ensure the quality of system designs.

Synchronous languages [4], such as Signal [2, 3], have been used successfully for the design of real-time critical applications to significantly ease the modeling and validation of software components. Their associated toolsets provide formal transformation, automatic code generation, verification, etc.

Signal is a dataflow relational language that relies on the polychronous model [3]. Signal handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as $\mathtt{x}$ and implicitly indexed by the discrete pace of its clock, noted $\hat{x}$. At a given instant, a *signal* may be present or absent. Two signals are synchronous if they are always present (or absent) at the same instants.

In Signal, a process (written P or Q) consists of the synchronous composition (noted $P \,|\, Q$) of equations (written $x := y\ f\ z$) over signals. An equation $x := y\ f\ z$ defines the output signal x by the result of the application of operator f to its input signals y and z. In addition to the extension to signals of usual functions on values (e.g., boolean or arithmetic operations), the specific basic Signal equations are the **delay** $x := y\$1$ init v, the **sampling** $x := y$ when z, and the **merging** $x := y$ default z. The reader is referred to [3] for definitions. The process P/x restricts the lexical scope of the signal x to the process P. The abstract syntax of a process P in Signal is defined as:

$$P, Q \ ::= \ x := y\ f\ z \mid P \,|\, Q \mid P/x$$

Relying on these bases, we propose an approach to automatically interpret AADL Behavior Annex into the synchronous formalism Signal. We are concerned with a method to embed notations of the AADL Behavior Annex in a suitable model of computation for the purpose of formal verification and code generation. To model and compile all distinctive programming features of AADL behaviors, transitions and actions, we use the code generation infrastructure of the synchronous modeling environment SME [5]. The SME environment is a front-end of Polychrony [7] (which is a toolset for Signal) in the Eclipse environment based on Model-Driven Engineering (MDE) technologies.

The model transformation specified here relies on an inductive SSA (static single assignment) [6] transformation algorithm across transitions/actions, that produces intermediate representation. A program is said to be in SSA form whenever each variable in the program appears only once on the left hand side of an assignment. Only one new value of a variable x should at most be defined within an instant. The SSA form of a program replaces assignments of a program variable x with assignments to new versions of x, uniquely indexing each assignment. The ϕ operator is needed to choose the value depending on the program control-flow, where a variable can be assigned in both branches of a conditional statement or in the body of a loop. As SSA is an intermediate representation, the introduction

of the ϕ-function will not consume execution cost. The compilation will optimize the final execution code. It introduces an effective method for transforming behavior specifications consisting of transitions and actions into a set of synchronous equations. This transformation minimizes the needed state variables and component synchronization.

In this paper, we show how to not only translate the core imperative programming features into equations, but also extend it to the mode automata that control the activation of such elementary transitions and actions. We give an overview of AADL and Behavior Annex in Section 2. The principles of the interpretation from AADL Behavior Annex to Signal are described in Section 3. Experimental results are provided by a case study. Some related works and conclusions are given in Section 4 and Section 5.

2 AADL and Behavior Annex

AADL is an SAE standard aimed at high level design and evaluation of the architecture of embedded systems. The language employs formal modeling concepts for the description of software and hardware architecture. It focuses on the description of systems using the component-based paradigm. A set of predefined components are offered:

• *Application software components* include process, thread, thread group, subprogram, and data components.

• *Execution platform components* model the hardware part of the system. This includes the processor, memory, device, and bus components.

• *Composite components* model components consisting of both hardware and software. A system component models a component containing execution platform, application software and other composite components.

The AADL *Behavior Annex* [8] is an extension of the core of the standard to offer a way to specify the local functional behavior of the components. It supports describing precisely the behaviors, such as port communication, computation, timing, etc. A Behavior Annex can be attached to a thread or a subprogram: threads or subprograms start from an initial state, and a transition to a complete (resp. return) state ends a thread (resp. subprogram). Transitions may be guarded by conditions, and actions may be attached.

Figure 1 is a behavior specification of a *door_handler* thread from a SDSCS (Simplified Doors and Slides Control System) [9]. It comprises two transitions with an initial state *s0* which is also a complete state. The thread will execute the transitions depending on the guarded conditions.

We will formalize the semantics of AADL behavior annex by isolating the core syntactic categories that characterize its expressive capability: transitions and actions.

Behavior specifications The behavior specification defines a transition system. A behavior specification **behavior_spec** x A describes system transitions from a source state s to a destination state t. A transition can be guarded with events or boolean conditions, and actions S can be attached. When a **behavior_spec** is first triggered, it starts execution from an initial state, specified as s : **initial state**, and ends with a complete (return for subprogram) state, specified as t : **complete state** (**return state**). From state s_i, it may perform a transition A to a new state s_j, written $s_i \overset{[g]}{\rightarrow} s_j\{S\}$, if the guard g is true.

$\mathbf{behavior\_spec}\ x\ A$ where $A ::= s_i \overset{[g]}{\rightarrow} s_j\{S\} \mid A_1 \| A_2$

and $g ::= [\mathbf{on}\ exp]\ [e]\ [\mathbf{when}\ exp] \mid exp$,

and $s ::= (\mathbf{initial} \mid \mathbf{complete} \mid \mathbf{return})\ \mathbf{state}$

• The guard g can either be an expression exp or contain an optional event or event data receipt e and an optional boolean condition ([**on** exp] [**when** exp]).

• The condition may depend on the received data, and it can be split in two parts: the **on** part expresses conditions over the current state, and the **when** part expresses a condition over the data to be read.

• An event e can be a receipt from an event port ($p?$), or from an event data port ($p?(x)$) where x is a state variable or a data subcomponent.

Actions Actions are sequences of operations on variables that are performed during the execution of transitions.

(action) $S ::= x := f(y,z) \mid p? \mid p?(x) \mid p! \mid p!(x)$

$\mid \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \mid \mathbf{for}\ (x\ \mathbf{in}\ X)\ S$

$\mid \mathbf{delay}(min,\ max) \mid \mathbf{computation}(min,\ max) \mid S_1; S_2$

• An assignment $x := f(y,z)$ defines the value of x from the function f of the current values of y and z.

• The conditional $\mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$ executes S_1 if the current value of x is true, otherwise executes S_2.

• The finite loop $\mathbf{for}\ (x\ \mathbf{in}\ X)\ S$ allows iterations over finite integer ranges or over unparameterized types, which are specified by a data classifier.

• The timing actions $\mathbf{delay}(min,\ max)$ and $\mathbf{computation}(min,\ max)$ specify non-deterministic waiting time and computation time intervals. The difference is that $\mathbf{computation}(min,\ max)$ consumes CPU, while $\mathbf{delay}(min,\ max)$ does not.

```
thread implementation door_handler.imp
annex behavior_specification {**
  states
   s0: initial complete state;
  transitions
   s0 -[on (dps > 3) and handle]-> s0 {
         warn_diff_pres:=true;
         door_info:=locked;
         if (on_ground) door_locked:=false;
         else door_locked:=true; end if;
        };
   s0 -[on not (dps > 3 and handle)]-> s0 {
         warn_diff_pres:=false;
         door_info:=locked;
         if (in_flight) door_locked:= true;
         else door_locked:=false; end if;
        };
  **};
end door_handler.imp;
```

Figure 1. Behavior Annex example

• The message sending or receiving actions express the communication of messages. A statement $p!$ calls the send service on an event or event data port p. The event is immediately sent to the destination with the stored data if any. $p!(x)$ writes data x to the event data port p and calls the send service. $p?$ dequeues an event from event port p. $p?(x)$ dequeues a data in the variable x.

• Sequences of actions $S_1; S_2$ are executed in order.

3 Interpretation and semantics of AADL Behavior Annex

In this section, we will present general rules to interprete the AADL Behavior Annex into Signal. This interpretation uses SSA as an intermediate formalism. The transitions and actions are transformed into a set of synchronous equations.

In order to distinguish between the transitions of different interpretation stage, we use $S \in \mathscr{S}$ to represent the general action in the original transition $T \in \mathscr{T}$, $B \in \mathscr{B}$ to represent the basic actions attached to the intermediate transition $U \in \mathscr{U}$, and $A \in \mathscr{A}$ to represent the SSA form actions attached to the SSA form transition $W \in \mathscr{W}$. We introduce a notation $def(A) := x$ referring the left hand side of an assignment $A\,(x := f(y,z))$, and $use(A) := f(y,z)$ referring the right hand side.

$\mathscr{B} \ni B ::= x := f(y,z) \mid p? \mid p?(x) \mid p! \mid p!(x) \mid B_1; B_2$

$\mathscr{S} \ni S ::= B \mid \textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{for } (x \textbf{ in } X)\ S$

$\mid \textbf{delay}(min,\ max) \mid \textbf{computation}(min,\ max) \mid S_1; S_2$

$\mathscr{A} \ni A ::= x := f(y,z) \mid A_1; A_2$

where $\forall x \in def(A), x$ occurs at most once in A

$\mathscr{T} \ni T ::= s \xrightarrow{c} t\{S\} \mid T_1 \parallel T_2$

$\mathscr{U} \ni U ::= s \xrightarrow{c} t\{B\} \mid U_1 \parallel U_2$

$\mathscr{W} \ni W ::= s \xrightarrow{c} t\{A\} \mid W_1 \parallel W_2$

Because actions are attached to transitions, we must take into consideration the states which they depart from and enter in, when interpreting the actions. We use $\mathcal{I}(T)$ to note the interpretation of a transition T to Signal process. The transformation $\mathcal{I}(T)$ of the AADL transitions/actions to Signal is addressed in three steps:

$$\mathcal{I}(T): \quad T \xrightarrow{\mathcal{I}_T} U \xrightarrow{\mathcal{I}_U} W \xrightarrow{\mathcal{I}_W} P$$

• Step 1: $T \xrightarrow{\mathcal{I}_T} U$. Each transition $T \in \mathscr{T}$ with attached sequence of general actions $S \in \mathscr{S}$, is decomposed into sets of basic transitions $U \in \mathscr{U}$, in which all the actions are basic actions $B \in \mathscr{B}$.

• Step 2: $U \xrightarrow{\mathcal{I}_U} W$. For each intermediate transition $U \in \mathscr{U}$, the basic actions $B \in \mathscr{B}$ are depicted in SSA form $A \in \mathscr{A}$. Each use of an original variable x is replaced by a new version, so that the actions can be executed in the same instant.

• Step 3: $W \xrightarrow{\mathcal{I}_W} P$. Translate the SSA form actions $A \in \mathscr{A}$ to Signal equations.

The interpretation of transitions $T_1 \parallel T_2$ can be parallel:

$\mathcal{I}(T_1 \parallel T_2) = \mathcal{I}(T_1) \parallel \mathcal{I}(T_2)$

$\mathcal{I}(T) = \mathcal{I}_W(W)$ where $W = \mathcal{I}_U(U)$, and $U = \mathcal{I}_T(T)$

Each step of the interpretation $\mathcal{I}_T, \mathcal{I}_U, \mathcal{I}_W$ will be explained in detail in the following subsections.

3.1 Actions to basic actions

A transition from a source state s when condition c is satisfied, to a destination state t, with attached general action S, noted as $s \xrightarrow{c} t\{S\}$, can be decomposed into a set of intermediate transitions $U \in \mathscr{U}$, in which the actions in each new transition are basic actions $B \in \mathscr{B}$.

$$s \xrightarrow{c} t\{S\} \quad \xrightarrow{\mathcal{I}_T} \quad U$$

where $U = s_i \xrightarrow{c_i} s_j\{B_i\} \mid U_1 \parallel U_2$, and $B_i \in \mathscr{B}$

We use $\mathcal{I}_T(T)$ to represent this interpretation from a general transition $T \in \mathscr{T}$ to basic transition $U \in \mathscr{U}$. The interpretation of the transitions can be parallel.

$$\mathcal{I}_T(T_1 \parallel T_2) = \mathcal{I}_T(T_1) \parallel \mathcal{I}_T(T_2)$$

We also use the notation $\mathcal{I}_{TS}$ to note this transformation, the action S is decomposed into $B; S'$, where B is the basic actions from the beginning of S, and S' is the rest.

$$\mathcal{I}_T(s \xrightarrow{c} t\{S\}) = \mathcal{I}_{TS}(s, c, t, S', B) = U$$

where $S = B; S'$, and $B \in \mathscr{B}$, and $U \in \mathscr{U}$

• At the very beginning of the interpretation, there is no basic action before S.

$$\mathcal{I}_T(s \xrightarrow{c} t\{S\}) = \mathcal{I}_{TS}(s, c, t, S, \phi)$$

• Suppose $S' = S_1; S_2$ where S_1 is the first action of S'. If action S_1 is a basic action $S_1 \in \mathscr{B}$, then it can be merged to the previous basic action set B. Otherwise, if B is empty, decompose S_1 and S_2 respectively, and apply the defined rules for each one. If B is not empty, introduce a new transition for action B, and decompose S_1, S_2 similarly to the previous case.

$$\mathcal{I}_{TS}(s, c, t, S_1; S_2, B) =$$

$$\begin{cases} \mathcal{I}_{TS}(s, c, t, S_2, B; S_1) & \text{if } S_1 \in \mathscr{B} \\ (U_1 \parallel U_2) & \text{if } S_1 \notin \mathscr{B} \text{ and } B = \phi \\ (s \xrightarrow{c} s_1\{B\} \parallel U_3 \parallel U_4) & \text{if } S_1 \notin \mathscr{B} \text{ and } B \neq \phi \end{cases}$$

$$\text{where } \begin{cases} (U_1, s_1) = \mathcal{I}'_T(S_1, c, s) \\ U_2 = \mathcal{I}_{TS}(s_1, \textbf{true}, t, S_2, \phi) \\ (U_3, s_2) = \mathcal{I}'_T(S_1, \textbf{true}, s_1) \\ U_4 = \mathcal{I}_{TS}(s_2, \textbf{true}, t, S_2, \phi) \end{cases}$$

The transformation for the composite action $S \in \mathscr{S}$ and $S \notin \mathscr{B}$, noted as $\mathcal{I}'_T(S, c, s) = (U, t)$ (where $s \xrightarrow{c} t\{S\}$ is the original transition with composite action S, U is the resulting basic transition), will be represented in the following subsections.

$\mathcal{I}'_T(S, c, s) = (U, t)$ where $S \in \mathscr{S}$, and $S \notin \mathscr{B}$, and $U \in \mathscr{U}$

• If there is no action following S in the interpretation, which means that the action S is already a basic action $S \in \mathscr{B}$, then the resulting transition is the same as the original one.

$$\mathcal{I}_{TS}(s, c, t, \phi, S) = (s \xrightarrow{c} t\{S\})$$

Next, we will interpret each of the composite actions S, ($S \in \mathscr{S}$ and $S \notin \mathscr{B}$). We write $\mathcal{I}'_T(S, c, s) = (U, t)$ for the action S from source state s, with guard c. It returns an intermediate transition U (the actions in U are the basic actions), and an exit state t.

3.1.1 Condition

A conditional evaluates S_1 with the condition x to U_1 and S_2 with condition **not** x to U_2.

$$\mathcal{I}'_T(\textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2, c, s) = \left(\left(\begin{array}{ll} & s \xrightarrow{c \textbf{ and } x} s_1 \\ \| & s \xrightarrow{c \textbf{ and not } x} s_2 \\ \| & U_1 \\ \| & U_2 \end{array}\right), u\right)$$

where $\mathcal{I}_{TS}(s_1, \textbf{true}, u, S_1, \phi) = U_1$, $\mathcal{I}_{TS}(s_2, \textbf{true}, u, S_2, \phi) = U_2$

3.1.2 Loop

A loop statement **for** $(x \textbf{ in } X)\ \{S\}$ allows iterations over finite integer ranges or over unparameterized enumerated types, which are specified by a data classifier.

Integer range For an integer range, i **in** $M..N$, which means that M and N are two integers and $M < N$, the loop statement can be refined as:

$\mathcal{I}'_T(\textbf{for } (i \textbf{ in } M..N)\ \{S\},\ c,\ s) =$

$$\left(\left(\begin{array}{ll} & s \xrightarrow{c} t\{i := M\} \\ \| & U \\ \| & u \to v\{i := i+1\} \\ \| & v \xrightarrow{i \le N} t \\ \| & v \xrightarrow{i > N} w \end{array}\right), w\right), \text{where } \mathcal{I}_{TS}(t, \textbf{true}, u, S, \phi) = U$$

Enumeration range For an iteration over unparameterized enumeration, x **in** X, in which X is a data classifier of enumerated type $X = \{x_1, x_2, ..., x_n\}$, the loop statement can be translated as:

$\mathcal{I}'_T(\textbf{for } (x \textbf{ in } X)\ \{S\},\ c,\ s) =$

$$\left(\left(\begin{array}{ll} & s \xrightarrow{c} t\{i := 1; x := x_1\} \\ \| & U \\ \| & u \to v\{i := i+1; x := x_i\} \\ \| & v \xrightarrow{i \le n} t \\ \| & v \xrightarrow{i > n} w \end{array}\right),\ w\right)$$

where $X = \{x_1, x_2, ..., x_n\}$, $|X| = n$, and $\mathcal{I}_{TS}(t, \textbf{true}, u, S, \phi) = U$

3.1.3 Computation(m,n)

Computation(m, n) expresses the use of CPU for a possibly non-deterministic period of time between m and n. For this non-deterministic choice, we introduce a function **random**(m, n) to choose a random period r $(m < r \le n)$ while translating. In each iteration of i, a synchronization with a physical tick $ms?$ is performed.

$\mathcal{I}'_T(\textbf{computation } (m,n), c, s) =$

$$\left(\left(\begin{array}{ll} & s \xrightarrow{c} t\ :\ \{i := 1\} \\ \| & t \xrightarrow{i < r} t\ :\ \{ms?; i := i+1\} \\ \| & t \xrightarrow{i = r} u \end{array}\right),\ u\right)$$

where $r = \textbf{random}(m, n)$,

and $ms?$ synchronizes with the physical tick

3.1.4 Delay(m,n)

The difference between **delay** and **computation** is that **computation** consumes CPU, while **delay** does not, which means that, when a thread delays some time interval, other threads can execute during this time.

The interpretation of **delay** is more complicated, for it will request for rescheduling. We can represent the **delay** using a finite loop if its period can be determined by a random choice:

$\mathcal{I}'_T(\textbf{delay } (m,n), c, s) = \mathcal{I}'_T((\textbf{for } i \textbf{ in } 1..r)\{ms?\}, c, s) = (U, t)$

where $r = \textbf{random}(m, n)$,

and $ms?$ synchronizes with the physical tick

3.2 Basic actions to SSA form actions

Only one new value of a variable x should at most be defined within an instant in SSA form. An operation $x := f(y, z)$ takes the current value of y and z to define the new value of x by the product with f. A statement $p!(x)$ sends the value x to p. Execution may continue within the same symbolic instant unless a second emission is performed. A statement $p?(x)$ waits a signal from p.

We use the notation $\mathcal{I}_U$ to note the intermediate transition $U \in \mathscr{U}$ (with actions $B \in \mathscr{B}$) to be represented in SSA form $W \in \mathscr{W}$ (with actions $A \in \mathscr{A}$).

$$s \xrightarrow{c} t\{B\} \xrightarrow{\mathcal{I}_U} W \quad \text{where} \begin{cases} B \in \mathscr{B} \\ \mathscr{W} \ni W = s_i \xrightarrow{c_i} s_j\{A_i\} \mid W_1 \| W_2 \\ A_i \in \mathscr{A} \end{cases}$$

We use an environment E to associate each variable with its definition, an expression that locates it, $E : X \to X$. The environment of x is noted $E(x)$, and the domain of E is noted $\mathcal{V}(E)$. The restriction of a variable x to environment E is noted E/x, which satisfies:

$$\phi/x = \phi$$

$$(E \uplus \{y \mapsto z\})/x = \begin{cases} E & \text{if } y = x \\ (E/x) \uplus \{y \mapsto z\} & \text{if } y \neq x \end{cases}$$

We write $use_E(x)$ for the expression that returns the definition of the variable x in environment E, and $def_E(x)$ for the environment E storing variable x with its associated definition.

$$use_E(x) = \begin{cases} E(x) & \text{if } x \in \mathcal{V}(E) \\ x & \text{otherwise} \end{cases}$$

$$def_E(x) = (E/x) \uplus \{x \mapsto x'\} \quad \text{where } x' \notin \mathcal{V}(E)$$

We depict sequence of basic actions $B = B_1; B_2$ attached to an intermediate transition U defined in environment E to a set of new SSA form transitions W with an updated environment F, in which all the new actions A are in SSA form, and can be executed in the same instant. We use $\mathcal{I}_U(U)$ to note this interpretation from an intermediate transition to SSA form transition $W \in \mathscr{W}$, and we introduce another notation $\mathcal{I}'_U$ to define the transformation rules:

$$\mathcal{I}_U(s \xrightarrow{c} t\{B\}) = W \text{ where } \begin{cases} \mathcal{I}'_U(B_1, B_2, s, E) = (W, t, F) \\ B = B_1; B_2 \end{cases}$$

The following transformations can be defined:

• For a single assignment basic action $x := f(y, z)$ in a transition with environment E, the new transition will take its SSA form assignment: the final version of variable x and the definition of y and z defined in E are used, the new environment F only stores the final value of x defined in E.

$$\mathcal{I}'_U(x := f(y,z), \phi, s, E) = (s \to t\{a := f(b,c)\}, t, F)$$
where $a = F(x)$, $b = use_E(y)$, $c = use_E(z)$, $F = def_E(x)$

• $p?(x)$ transfers a data received from port p to the variable x. If no other action is placed before or after it, convert it to SSA form action attached to the transition. The environment is updated as $def_E(x)$.

$$\mathcal{I}'_U(p?(x), \phi, s, E) = (s \to t\{F(x) := p\}, t, F)$$
where $F = def_E(x)$

• $p!(x)$ writes data x to event or event data port p, and calls the *Send* service. A new unique version p' of p ($\{p \mapsto p'\}$) is added to update the original environment E.

$$\mathcal{I}'_U(p!(x), \phi, s, E) = (s \to t\{p' := use_E(x)\}, t, F)$$
where $F = E \biguplus \{p \mapsto p'\}$, and $p' \notin \mathcal{V}(E)$

In the same way, we have the following transformations:

• An assignment can be rewritten in SSA form, and merged to the previous basic action set B_1.

$$\mathcal{I}'_U(B_1, (x := f(y,z); B_2), s, E)$$
$$= \mathcal{I}'_U((B_1; a := f(b,c)), B_2, s, F) \quad \text{where} \begin{cases} a = F(x) \\ b = use_E(y) \\ c = use_E(z) \\ F = def_E(x) \end{cases}$$

• The receive message action is represented in SSA form, and merged to the previous B_1.

$$\mathcal{I}'_U(B_1, (p?(x); B_2), s, E)$$
$$= \mathcal{I}'_U((B_1; F(x) := p), B_2, s, F) \text{ where } F = def_E(x)$$

• The send message action can be merged if p has not been defined in E, and $\{p \mapsto p\}$ is added to update E. Otherwise, create a transition attached with the actions B_1 and the final values of all variables x defined in E. And apply the transformation rules for the rest of the actions with an environment $\{p \mapsto p\}$.

$$\mathcal{I}'_U(B_1, (p!(x); B_2), s, E) =$$
$$\begin{cases} \mathcal{I}'_U(B'_1, B_2, s, E \uplus \{p \mapsto p\}) & \text{if } p \notin \mathcal{V}(E) \\ (s \to t\{B_1; B'\}) \parallel W & \text{otherwise} \end{cases}$$
where $\begin{cases} B'_1 = B_1; p := use_E(x) \\ B' = \prod_{\forall x \in \mathcal{V}(E)} x := def_E(x); \\ \text{where } \prod \text{ is the composition of final value of } x \\ \mathcal{I}'_U(p!(x), B_2, t, \{p \mapsto p\}) = (W, u, F) \end{cases}$

3.3 SSA to Signal

Finally, all the transitions/actions can be represented in the following form:

$$\mathscr{W} \ni W ::= s \xrightarrow{c} t\{A\} \mid W_1 \parallel W_2$$
where $\forall x \in def(A), x$ occurs at most once in A

$$\mathscr{A} \ni A ::= x := f(y,z) \mid A_1; A_2$$

The interpretation $\mathcal{I}_A(A)^g_E = (P, F)$ of an SSA form action A takes as parameters the environment E and the guard g that leads to it. It returns a process P, and an updated environment F.

$$\mathcal{I}_A(x := f(y,z))^g_E = (x = f(a \textbf{ when } g,\ b \textbf{ when } g), F)$$
where $\begin{cases} a = y \text{ if } y \in E, \text{ else } y\$1 \\ b = z \text{ if } z \in E, \text{ else } z\$1 \\ F = E \cup \{x\} \end{cases}$

$$\mathcal{I}_A(A_1; A_2)^g_E = (P|Q, G) \quad \text{where} \begin{cases} \mathcal{I}_A(A_1)^g_E = (P, F) \\ \mathcal{I}_A(A_2)^g_F = (Q, G) \end{cases}$$

We use the notation $\mathcal{I}_W(W)^{st}$ to represent the interpretation from the SSA form transition $W \in \mathscr{W}$ to Signal process. A local signal st is introduced to represent the state variable, and nst represents the next state.

$$\mathcal{I}_W(W_1 \parallel W_2)^{st} = \mathcal{I}_W(W_1)^{st} \mid \mathcal{I}_W(W_2)^{st}$$
$$\mathcal{I}_W(s \xrightarrow{c} t\{A\})^{st} = (P \mid nst = t \textbf{ when } g)$$
where $\begin{cases} (P, E) = \mathcal{I}_A(A)^g_\phi \\ g = c \textbf{ when } (st = s) \end{cases}$

Following the three steps' interpretation, the behavior specification **behavior_spec** x X, can now be represented as **behavior_spec** x $(W \parallel$ **initial state** $s_0)$, $(W \in \mathscr{W})$. The interpretation for the behavior specification can be written as:

$$\mathcal{I}(\textbf{behavior\_spec } x \ (T \parallel \textbf{initial state } s_0))$$
$$= \mathcal{I}_W(\textbf{behavior\_spec } x \ (W \parallel \textbf{initial state } s_0))$$
where $W = \mathcal{I}_U(U)$, and $U = \mathcal{I}_T(T)$
$$\mathcal{I}_W(\textbf{behavior\_spec } x \ (W \parallel \textbf{initial state } s_0))$$
$$= (P \mid st = nst \ \$ \textbf{ init } s_0) \ / \ st, nst, \quad \text{where } P = \mathcal{I}_W(W)^{st}$$

3.4 A Case study

The intermediate generated code of the basic transitions/actions (step $T \xrightarrow{\mathcal{I}_T} U$) for the *door_handler* thread (previously presented in Figure 1) is depicted in Figure 2. It contains a transformation of the transitions as well as their attached actions. The interpretation introduces two intermediate states: *STATE_0, STATE_1*. The first three transitions rewrite the first original one. A transition is introduced for each branch of the condition action.

```
thread implementation door_handler.imp
  annex behavior_specification {**
    states
      STATE_0 : state;
      STATE_1 : state;
    transitions
      s0 -[ on dps > 3 and handle ]-> STATE_0 {
        warn_diff_pres := true;
        door_info := locked;};
      STATE_0 -[ on on_ground ]-> s0 {
        door_locked := false; };
      STATE_0 -[ on not on_ground ]-> s0 {
        door_locked := true; };
      s0 -[ on not dps > 3 and handle ]-> STATE_1 {
        warn_diff_pres := false;
        door_info := locked;};
      STATE_1 -[ on in_flight ]-> s0 {
        door_locked := true; };
      STATE_1 -[ on not in_flight ]-> s0 {
        door_locked := false; };
**};
end door_handler.imp;
```

Figure 2. Intermediate Behavior code

In this example, the SSA form of transitions is the same as depicted in Figure 2, since all the variables are already uniquely defined in each of the transitions. Each transition will then be interpreted to Signal equations. The generated code for the first SSA form transition is listed here, the guard and state variables (st, nst) are added.

| warn_diff_pres := true when $(st = s0)$
when $((dps > 3)$ and handle)
| door_info := locked when $(st = s0)$ when $((dps > 3)$and handle)
| nst := STATE_0 when $(st = s0)$ when $((dps > 3)$ and handle)
| st := nst $ 1 init s0

The program is translated into Signal following the general rules described above. Then simulation code (C code) is generated with the help of the Polychrony toolset. Traces can be added to be able to follow the simulation. Properties can be checked using Sigali which is a Signal companion model-checker. Similar experiments have been described in [10] for Signal code obtained through SSA from C/C++ parallel programs.

4 Related Works

A few approaches for defining the semantics or for interpreting the behavior annex of AADL have been proposed. AADL2BIP [11] studies a general methodology for translating AADL and behavior annex specification into BIP. It supposes the actual behaviors are described using the implementation language, so the actions are not considered, while the translation of the transitions is shown roughly. In our transformation, the transitions and actions are both presented. [12] proposes a formal semantics for a subset of AADL behavior annex using Timed Abstract State Machine (TASM). It defines the synchronization actions (remote subprogram call) which have not been considered in our approach, however, it does not present how the basic actions are translated. In AADL2Fiacre [13], the transformations from AADL and behavior annex to Fiacre are illustrated on a small example, but the semantics are not formally defined.

Our approach and tools are based on the studies and experimental results on the translation of C/C++ parallel codes into synchronous formalism using SSA transformation [10]. In the ANR project SPACIFY, [14] proposes an approach to model notations of the Synoptic language and to embed them in a suitable model of computation for the purpose of formal verification and code generation. It consists of an inductive SSA transformation algorithm across a hierarchy of blocks that produces synchronous equations.

5 Conclusion

We have presented in this paper the principle and implementation that interpret AADL behavior annex into a synchronous data-flow and multi-clocked model of computation. This interpretation is based on the use of SSA as an intermediate format. It gives a thorough description of an inductive SSA transformation algorithm across a hierarchy of transitions that produce synchronous equations.

Our technique uses the underlying model of computation of the SME platform. We obtain an effective method for transforming a hierarchy of behavior specifications consisting of transitions and actions into a set of synchronous equations. The impact of this transformation technique has a big advantage: it minimizes the number of state variables across hierarchical automata and hence creates a minimal number of transitions in the generated code.

For future works, we intend to implement an automatic transformation. Since the intermediate SSA form is very simple, the implementation can focus on optimizations and performances. One extension is to model the scheduling policy as well as a rescheduling algorithm when a system service is requested. Furthermore, an extension to composite state specified in Behavior Annex jointly with the actions would be interesting. Another extension study is the validation of message communication optimization.

References

[1] SAE. **Architecture Analysis & Design Language (standard SAE AS5506)**, September 2004, http://www.sae.org

[2] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire. **Programming Real-Time Applications with SIGNAL**, Proceedings of the IEEE 79(9), Sep 1991

[3] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann. **Polychrony for System Design**, Journal for Circuits, Systems and Computers, April 2003

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, R. De Simone, **The Synchronous Languages Twelve Years Later**, Proceedings of the IEEE, 2003

[5] C. Brunette, J.-P. Talpin, A. Gamatié, T. Gautier. **A metamodel for the design of polychronous systems**, Journal of Logic and Algebraic Programming, Special Issue on Applying Concurrency Research to Industry. Elsevier, 2008

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck. **Efficiently computing static single assignment form and the control dependence graph**, ACM Transactions on Programming Languages and Systems, Oct 1991

[7] INRIA Espresso Team, Polychrony tool, http://www.irisa.fr/espresso/Polychrony

[8] P. Dissaux, J.P. Bodeveix, M. Filali, P. Gaufillet, F. Vernadat, **AADL Behavioral annex**, DASIA, 2006

[9] O. Laurent, F. Pouzolz, **Airbus generic pilot application Aircraft doors management system**, CESAR report, 2009

[10] L. Besnard, T. Gautier, M. Moy, J.-P. Talpin, K. Johnson, F. Maraninchi, **Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form**, AVOCS'09, Sep 2009

[11] M. Y. Chkouri, A. Robert, M. Bozga, J. Sifakis, **Translating AADL into BIP - Application to the Verification of Real-time Systems**, MODELS'08, Toulouse, France, Sep 2008

[12] Z. Yang, K. Hu, D. Ma, L. Pi, **Towards a Formal Semantics for the AADL Behavior Annex**, DATE'09, 2009

[13] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat, **Fiacre: an Intermediate Language for Model Verification in the TOPCASED Environment**, INRIA report, 2008

[14] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, **Modular interpretation of heterogeneous modeling diagrams into synchronous equations using static single assignment**, INRIA report, September 2009