Verification of Parameterized Hierarchical State Machines Using Action Language Verifier *

Tuba Yavuz-Kahveci
CISE Department
University of Florida
Gainesville, FL 32611
tyavuz@cise.ufl.edu

Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA 93106
bultan@cs.ucsb.edu

Abstract

Action Language Verifier (ALV) is an infinite-state symbolic model checker. ALV can verify (or falsify, by generating counter-examples) temporal logic properties of systems that can be modeled using a combination of boolean logic and linear arithmetic expressions on boolean, enumerated and (possibly unbounded) integer variables and parameterized integer constants. In this paper, we apply ALV to the verification of parameterized hierarchical state machine specifications. We extend the standard notation for hierarchical state machines by introducing primitives for explicit specification of asynchronous processes and their finite and parameterized instantiations. We define the formal semantics of these primitives, where the states of the parameterized processes are mapped to integer variables using the counting abstraction technique. We apply the presented approach to the specification and analysis of an airport ground traffic controller and verify several correctness properties of this specification using ALV.

1 Introduction

Hierarchical state machines (HSMs) have been very influential in specification of computer systems after David Harel's seminal work on Statecharts [21]. Variations on HSMs have become part of popular object oriented design languages [5], are supported by commercial design support tools [22], influenced the requirements specification languages [23], and have been investigated from automated verification perspective [6, 7, 17, 18]. In this paper, we are focusing on automated verification of *parameterized* HSMs. In addition to including well-known concepts from Statecharts, we extend the HSMs using an instantiation operator as a way of specifying the asynchronous composition of HSMs. A parameterized HSM contains parameterized components which can be instantiated arbitrary (unbounded but finite) number of times. In order to verify parameterized HSMs we translate them to the Action Language, and then use the Action Language Verifier to model check their CTL properties. Although the Action Language translation is done manually at this point, it can be automated based on the HSM semantics used in this paper.

Action Language is a specification language for reactive software systems [13]. The Action Language Verifier (ALV) [1, 16] consists of 1) a compiler that converts Action Language specifications to symbolic representations, and 2) an infinite-state symbolic model checker which verifies (or falsifies by generating counter-examples) CTL properties of Action Language specifications. ALV specializes on systems specified with linear arithmetic constraints on integer variables. It uses the Composite Symbolic Library [28, 29] as its symbolic manipulation engine. Composite Symbolic Library integrates multiple symbolic representations: BDDs for boolean and enumerated variables, polyhedral or automata representations for integer variables, and BDDs for bounded integer variables.

Since Action Language allows specifications with unbounded integer variables, fixpoint computations are not guaranteed to converge. ALV uses conservative approximation techniques, reachability and acceleration heuristics to achieve convergence. ALV uses the counting abstraction technique [19] for verification of parameterized systems [26, 27]. Counting abstraction generates a set of integer variables and a set of linear arithmetic constraints on these variables to represent the behavior of arbitrary number of finite state processes. In this paper we use this feature of ALV to verify parameterized HSMs.

Our results demonstrate that infinite state model checking tools can be effective in verifying properties of parameterized hierarchical state machines. Model checking infinite state systems specified by linear arithmetic constraints,

*This work is supported in part by the NSF grant CCR-0341365.

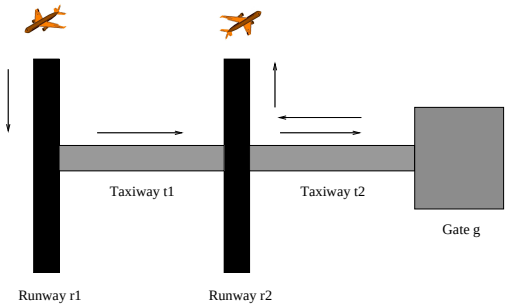

Figure 1. An airport ground network.

such as the ones targeted by ALV, has been an active research area in the automated verification community in recent years [8, 15, 20, 24, 25] Among the tools developed based on the results in this area [2, 3, 4], ALV is unique in combining multiple symbolic representations. Additionally, Action Language provides high level constructs such as the ability to declare parameterized processes, which is supported by ALV using automated counting abstraction.

We applied the approach presented in this paper to a simple airport ground network system shown in Figure 1. In this example, the ground network consists of two runways, two taxiways, and a single gate. An arriving airplane lands using runway r1, navigates on taxiway t1, crosses runway r2, and navigates on taxiway t2 to reach gate g, where it parks. A departing airplane starts from gate g, navigates on taxiway t2 and takes off using runway r2. The control logic for such a system must avoid accidents and deadlock. We specified the control logic for this example using parameterized HSMs as shown in Figure 2. We modeled the ground network as a hierarchical state machine which consists of synchronous state machines that model individual resources, e.g., state `r1` models runway r1. Each airplane is modeled as a hierarchical state machine (`Airplane`) which consists of substates that model the status of the airplane, e.g., state `landing` models landing status. The state `Airplane[*]` denotes asynchronous composition of *arbitrary number* of `Airplane` state machines. The whole system is modeled as the synchronous composition of `Airplane[*]` and the state machines modelling the ground network resources (`r1`, `r2`, `t1`, `t2`, and `g`).

The rest of the paper is organized as follows. In Section 2 we define the HSM notation and its semantics. In Section 3 we define the parameterized HSMs. In Section 4 we discuss the verification of parameterized HSMs using ALV. Finally, in Section 5, we give our conclusions.

2 Hierarchical State Machines

In this section we define the Hierarchical State Machine (HSM) notation and its semantics.

2.1 Modes

A mode[1] denotes a control point. It is represented by an annotated rectangle. A mode is called *active* if the HSM is currently at that mode. There are two types of modes: *basic modes* and *complex modes*. A complex mode describes a hierarchy and it is composed of submodes which can be either basic modes or complex modes. A complex mode can be one of two types: an *OR-mode* or an *AND-mode*. An OR-mode is constructed by connecting its submodes via transitions. If an OR-mode is active, then exactly one of its submodes is also active. One of the submodes of an OR-mode is identified as the *initial mode* and represented by an incoming curved arc. An AND-mode is constructed by separating its submodes by dashed lines. An AND-mode denotes concurrent execution of its submodes. If an AND-mode is active, then all of its submodes are also active.

We denote the set of all mode names in an HSM as $Mode$. Given a mode m, $Sub(m)$ denotes the set of modes which are the submodes of m. A mode m_1 is the super-mode of the mode m_2 ($Sup(m_2) = m_1$) if m_2 is a submode of m_1, i.e., $m_2 \in Sub(m_1)$. A mode m_1 is said to be an *ancestor* of the mode m_2 if either m_1 is a super-mode of m_2 or m_1 is an ancestor of the super-mode of m_2. Similarly, m_1 is said to be a descendant of m_2 if either m_1 is a submode of m_2 or m_1's super-mode is a descendant of m_2. $Desc(m)$ denotes the set m's descendants. Two modes m_1 and m_2 are called *peer modes*, if they are submodes of the same node. The *lowest-common ancestor* of m_1 and m_2, $LCA(m_1, m_2)$, is defined as the mode that is an ancestor of both m_1 and m_2 and which does not have a descendant that is the ancestor of both m_1 and m_2. The *top-most peer ancestor* of m_1 with respect to m_2, $TPA(m_1, m_2)$, is the ancestor of m_1 that is a submode of the lowest-common ancestor of m_1 and m_2. A mode m is active if and only if $m.act = true$.

We define three functions $S, SA, SD : Mode \rightarrow Formula$ which map modes to formulas. Given a mode m, $S(m)$ is the formula denoting that m is currently active. We define $S(m)$ as $S(m) = SA(m) \wedge SD(m)$ where the formulas $SA(m)$ and $SD(m)$ denote the constraints on the ancestors and the descendants of m, respectively. They are recursively defined as follows:

$$SA(m) = \begin{cases} m_1.act \wedge SA(m_1) & \text{if } m_1 = Sup(m) \text{ and } m_1 \text{ is an OR-mode} \\ m_1.act \wedge SA(m_1) \wedge \bigwedge_{m_2 \in Sub(m_1), m_2 \neq m} SD(m_2) & \text{if } m_1 = Sup(m) \text{ and } m_1 \text{ is an AND-mode} \\ true & \text{if } m \text{ is the root mode} \end{cases}$$

[1] We use the word *mode* here since we use the word *state* to refer to the configurations of the hierarchical state machines.

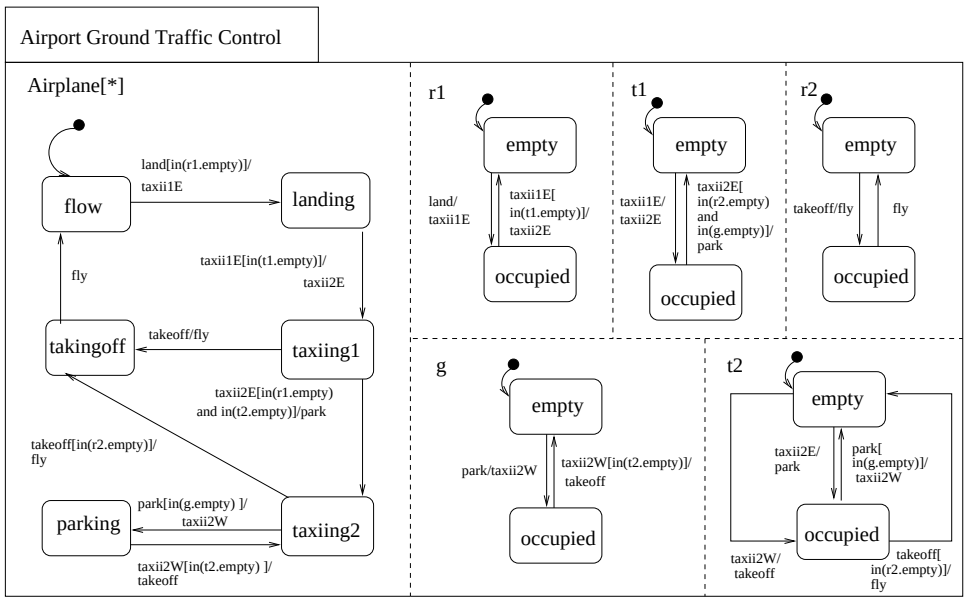

Figure 2. Hierarchical state machine specification of the airport ground network control system.

$$SD(m) = \begin{cases} m.act & \text{if m is a basic mode} \\ (\bigoplus_{m_1 \in Sub(m)} m_1.act) \wedge (\bigwedge_{m_1 \in Sub(m)} m_1.act \Rightarrow SD(m_1)) \wedge m.act & \text{if m is an OR-mode} \\ (\bigwedge_{m_1 \in Sub(m)} SD(m_1)) \wedge m.act & \text{if m is an AND-mode} \end{cases}$$

where the expression with the exclusive-or operator ($\bigoplus$) denotes that exactly one of the submodes of an active OR-mode should be active.

2.2 Transitions

A transition models an atomic mode change and is represented by a solid arrow connecting two modes. Let $t^{m_d}_{m_s}$ denote a transition from m_s to m_d, where m_s and m_d denote the source mode and the destination mode of t, respectively. When a transition $t^{m_d}_{m_s}$ is taken, the source mode m_s, some of its ancestors and all of its descendants become inactive, and the destination mode m_d and all of its ancestors and some of its descendants become active. We restrict the transitions so that, if a mode m is the source or destination mode of a transition, then $Sup(m)$ must be an OR-mode, i.e., there are no transitions from or to the submodes of AND-modes. Moreover, given a transition $t^{m_d}_{m_s}$, if m_s (m_d) is a descendant of a submode of an AND-mode then m_d (m_s) must be a descendant of the same submode of that AND-mode, i.e., for all $t^{m_d}_{m_s}$, for all m, if $Sup(m)$ is an AND-mode, then $m_s \in Desc(m)$ if and only if $m_d \in Desc(m)$.

Transitions can be labeled with three fields $trigger[cond]/generate$ where $trigger$ is the trigger event of the transition, $generate$ is the event generated by the transition, and $cond$ is a boolean combination of predicates in the form $in(m.sm)$. The $cond$ and $generate$ fields are optional. The predicate $in(m.sm)$ denotes that the submode sm of the OR-mode m is currently active. We model the events using boolean variables. A transition can be taken if the trigger event and the $cond$ are true (if $cond$ is not given, it is assumed to be *true* by default). Taking the transition sets the trigger event to *false* and sets the generated event (if it exists) to *true*. We call the formula which denotes this transition semantics the *transition formula*. Let $f(t^{m_d}_{m_s})$ denote the transition formula for the transition $t^{m_d}_{m_s}$. We write the transition formula as $f(t^{m_d}_{m_s}) = g(t^{m_d}_{m_s}) \wedge u(t^{m_d}_{m_s})$ where $g(t^{m_d}_{m_s})$ is called the *guard* and states that the trigger event and the $cond$ are true, and $u(t^{m_d}_{m_s})$ is called the *update* and sets the value of the trigger event to *false* and the generated event (if it exists) to *true* in the next state. A transition preserves the values of all the events that it does not update. When the mode m_s is active, the transition $t^{m_d}_{m_s}$ can be taken provided that the $g(t^{m_d}_{m_s})$ evaluates to *true*. If there is more than one transition originating from the same mode and if the guards of more than one of them evaluate to *true* at the same time, then one of the transitions is taken nondeterministically.

Now, we will define the semantics of executing a transition. Let $LU : Mode \times Mode \rightarrow Formula$ denote the mapping from a pair of modes (m_1, m_2), where m_2 is a descendant of m_1, to a formula which denotes the state of m_2 and its ancestors up to m_1 after a transition exits m_2. We use primed modes to denote the values of the attributes after

the transition is taken ($m.act$ is the value of the act attribute of mode m at the current state, whereas $m'.act$ is the value of the act attribute of mode m at the next state, after the transition is taken).

$$LU(m_1, m_2) = \begin{cases} \neg m_1'.act \wedge \neg m_2'.act & \text{if } m_2 \in Sub(m_1) \\ \neg m_2'.act \wedge & \text{if } m_2 \notin Sub(m_1) \\ LU(m_1, Sup(m_2)) & \end{cases}$$

Let $LD : Mode \rightarrow Formula$ denote the mapping from a mode m to a formula which denotes the state of mode m and all its descendants after m is exited by a transition.

$$LD(m) = \begin{cases} \neg m'.act & \text{if } m \text{ is a basic mode} \\ \neg m'.act \wedge & \text{if } m \text{ is an OR-mode} \\ \bigwedge_{m_1 \in Sub(m)} & \\ (m_1.act \Rightarrow LD(m_1)) & \\ \neg m'.act \wedge & \text{if } m \text{ is an AND-mode} \\ \bigwedge_{m_1 \in Sub(m)} LD(m_1) & \end{cases}$$

Let $Left : Mode \times Mode \rightarrow Formula$ denote the mapping from a pair of modes (m_1, m_2) to a formula which denotes the state of m_2, its descendants, and its ancestors up to m_1, after a transition exits m_2.

$$Left(m_1, m_2) = \begin{cases} LD(m_2) & \text{if } m_1 = m_2 \\ LU(m_1, Sup(m_2)) & \text{if } m_2 \in \\ \wedge LD(m_2) & Desc(m_1) \end{cases}$$

When a transition $t_{m_s}^{m_d}$ is taken, the destination mode m_d, all of its ancestors, and some of its descendants become active. Below we define the effect of executing a transition on the destination mode and its ancestors and its descendants.

Let $EnU : Mode \times Mode \rightarrow Formula$ denote the mapping from a pair of modes (m_1, m_2), where m_2 is a descendant of m_1, to a formula which denotes the state of mode m_2 and its ancestors up to m_1 after a transition enters m_2.

$$EnU(m_1, m_2) = \begin{cases} m_1'.act \wedge m_2'.act & \text{if } m_2 \in \\ & Sub(m_1) \\ EnU(m_1, Sup(m_2)) & \text{if } m_2 \notin \\ \wedge m_2'.act & Sub(m_1) \end{cases}$$

Let $EnD : Mode \rightarrow Formula$ denote the mapping from a mode m to a formula which denotes the state of mode m and all its descendants after m is entered by a transition.

$$EnD(m) = \begin{cases} m'.act & \text{if } m \text{ is a} \\ & \text{basic mode} \\ m'.act \wedge EnD(m.init) & \text{if m is an} \\ & \text{OR-mode} \\ m'.act \wedge & \text{if m is an} \\ \bigwedge_{m_1 \in Sub(m)} EnD(m_1) & \text{AND-mode} \end{cases}$$

Let $En : Mode \times Mode \rightarrow Formula$ denote the mapping from a pair of modes (m_1, m_2), where m_2 is a descendant of m_1, to a formula which denotes the state of m_2, its descendants, and its ancestors up to m_1, after a transition enters m_2.

$$En(m_1, m_2) = \begin{cases} EnD(m_2) & \text{if } m_1 = m_2 \\ EnU(m_1, Sup(m_2)) & \text{if } m_2 \in \\ \wedge EnD(m_2) & Desc(m_1) \end{cases}$$

Let $IdND : Mode \rightarrow Formula$ denote the mapping from a mode m to a formula which denotes that for all the modes which are not descendants of m, the act attribute remains the same in the next state.

$$IdND(m) = \bigwedge_{m \notin Desc(m)} m'.act = m.act$$

Similarly, $IdD : Mode \rightarrow Formula$ is the mapping from a mode m to a formula which denotes that for all the modes which are the descendants of m, the act attribute remains the same in the next state.

$$IdD(m) = \bigwedge_{m \in Desc(m)} m'.act = m.act$$

Now, we can define the semantics of a transition $t_{m_1}^{m_2}$ as

$$\begin{aligned} [\![t_{m_1}^{m_2}]\!] \quad = \quad & S(m_1) \wedge Left(m_3, m_1) \wedge En(m_4, m_2) \\ & IdND(m_5) \wedge f(t_{m_1}^{m_2}) \end{aligned}$$

where $m_3 = TPA(m_1, m_2)$, $m_4 = TPA(m_2, m_1)$, and $m_5 = LCA(m_1, m_2)$. Then, the semantics of a mode m is defined as follows:

$$[\![m]\!] = \begin{cases} false & \text{if } m \text{ is a basic mode} \\ \bigvee_{t_{m_1}^{m_2}, m = LCA(m_1, m_2)} [\![t_{m_1}^{m_2}]\!] & \text{if } m \text{ is an OR-mode} \\ \vee \bigvee_{m_1 \in Sub(m)} [\![m_1]\!] & \\ \bigwedge_{m_3 \in Sub(m)} ([\![m_3]\!] \vee & \text{if } m \text{ is an AND-mode} \\ (IdD(m_3) \wedge & \\ \bigwedge_{t_{m_1}^{m_2}, m_3 = LCA(m_1, m_2)} & \\ \neg g(t_{m_1}^{m_2})) & \end{cases}$$

3 Parameterized HSMs

We incorporate an instantiation operator to HSMs as a way of specifying asynchronous composition of a set of identical state machines. The instantiation operator is represented by the suffix "$[n]$" that is appended to a complex mode name where n can be a number denoting the number of instantiations or it can be the "*" character denoting the arbitrary number of instantiations. We call a mode with arbitrary number of instantiations a *parameterized mode*. For instance, in Figure 2, `Airplane` is a parameterized mode. The semantics of the mode instantiation is defined as

$$[\![M[n]]\!] = \bigvee_{1 \le i \le n} [\![M_i]\!]$$

where M_i denotes the HSM that is obtained from M by appending i to each mode name. We explain the semantics of parameterized modes below.

3.1 Parameterized Modes

We use an abstraction technique called *counting abstraction* [19] to define the semantics of parameterized modes. Counting abstraction is used to abstract the local states of a set of identical components. The abstracted system does not keep track of the state of each component, rather, it keeps track of the number of components in each state. This is achieved by introducing a set of integer variables, one per state, counting the number of components in that state.

We extend the notation we introduced so far by introducing the superscript P to denote the parameterized case. We associate a counter $m.c$ with each mode m, which denotes the number of instances of the parameterized state machine that are active in m. We define $S^P, SA^P, SD^P : Mode \rightarrow Formula$ as $S^P(m) = SA^P(m) \wedge SD^P(m)$ where SA and SD are recursively defined as follows:

$$SA^P(m) = \begin{cases} m_1.c > 0 \wedge SA^P(m_1) & \text{if } m_1 = Sup(m) \text{ and } m_1 \text{ is an OR-mode} \\ m_1.c > 0 \wedge SA^P(m_1) \wedge_{m_2 \in Sub(m_1), m_2 \neq m} SD^P(m_2) & \text{if } m_1 = Sup(m) \text{ and } m_1 \text{ is an AND-mode} \\ true & \text{if } m \text{ is the root mode} \end{cases}$$

$$SD^P(m) = \begin{cases} m.c > 0 & \text{if m is a basic mode} \\ (\bigvee_{m_1 \in Sub(m)} m_1.c > 0 \wedge SD^P(m_1)) \wedge m.c > 0 & \text{if m is an OR-mode} \\ (\bigwedge_{m_1 \in Sub(m)} m_1.c > 0 \wedge SD^P(m_1)) \wedge m.c > 0 & \text{if m is an AND-mode} \end{cases}$$

Below, we present the transition semantics for parameterized modes, based on the notation for the non-parameterized case in Section 2. Let $Dec(c) \equiv c' = c - 1$ and $Inc(c) \equiv c' = c + 1$.

$$LU^P(m_1, m_2) = \begin{cases} Dec(m_1.c) \wedge Dec(m_2.c) & \text{if } m_2 \in Sub(m_1) \\ Dec(m_2.c) \wedge LU^P(m_1, Sup(m_2)) & \text{if } m_2 \notin Sub(m_1) \end{cases}$$

$$LD^P(m) = \begin{cases} Dec(m.c) & \text{if } m \text{ is a basic mode} \\ Dec(m.c) \wedge (\bigoplus_{m_1 \in Sub(m)} m_1.c > 0 \wedge LD^P(m_1)) & \text{if } m \text{ is an OR-mode} \\ Dec(m.c) \wedge \bigwedge_{m_1 \in Sub(m)} LD^P(m_1) & \text{if } m \text{ is an AND-mode} \end{cases}$$

$$Left^P(m_1, m_2) = \begin{cases} LD^P(m_2) & \text{if } m_1 = m_2 \\ LU^P(m_1, Sup(m_2)) \wedge LD^P(m_2) & \text{if } m_2 \in Desc(m_1) \end{cases}$$

$$EnU^P(m_1, m_2) = \begin{cases} Inc(m_1.c) \wedge Inc(m_2.c) & \text{if } m_2 \in Sub(m_1) \\ EnU^P(m_1, Sup(m_2)) \wedge Inc(m_2.c) & \text{if } m_2 \notin Sub(m_1) \end{cases}$$

$$EnD^P(m) = \begin{cases} Inc(m.c) & \text{if } m \text{ is a basic mode} \\ Inc(m.c) \wedge EnD^P(m.init) & \text{if m is an OR-mode} \\ Inc(m.c) \wedge \bigwedge_{m_1 \in Sub(m)} EnD^P(m_1) & \text{if m is an AND-mode} \end{cases}$$

$$En^P(m_1, m_2) = \begin{cases} EnD^P(m_2) & \text{if } m_1 = m_2 \\ EnU^P(m_1, Sup(m_2)) \wedge EnD^P(m_2) & \text{if } m_2 \in Desc(m_1) \end{cases}$$

$$
\begin{array}{lcl}
IdND^P(m) & = & \bigwedge_{m' \notin Desc(m)} m'.c = m.c \\
IdD^P(m) & = & \bigwedge_{m' \in Desc(m)} m'.c = m.c
\end{array}
$$

$$
\begin{array}{lcl}
[\![t_{m_1}^{m_2}]\!]^P & = & S^P(m_1) \wedge Left^P(m_3, m_1) \wedge En^P(m_4, m_2) \\
 & & IdND^P(m_5) \wedge f(t_{m_1}^{m_2})
\end{array}
$$

where $m_3 = TPA(m_1, m_2)$, $m_4 = TPA(m_2, m_1)$, and $m_5 = LCA(m_1, m_2)$.

$$
[\![m]\!]^P = \begin{cases}
false & \text{if } m \text{ is a basic mode} \\
\bigvee_{t_{m_1}^{m_2}, m = LCA(m_1, m_2)} [\![t_{m_1}^{m_2}]\!]^P \vee \bigvee_{m_1 \in Sub(m)} [\![m_1]\!]^P & \text{if } m \text{ is an OR-mode} \\
\bigwedge_{m_3 \in Sub(m)} ([\![m_3]\!]^P \vee (IdD^P(m_3) \wedge \bigwedge_{t_{m_1}^{m_2}, m_3 = LCA(m_1, m_2)} \neg g(t_{m_1}^{m_2})) & \text{if } m \text{ is an AND-mode}
\end{cases}
$$

3.2 Avoiding Redundancy

It is possible to determine all the active modes in an HSM just by looking at the basic modes, i.e., the set of basic modes that are active determines all the active modes. For the parameterized HSMs this results in the following property: Given a complex mode m, the counter $m.c$ can be defined in terms of the counters of its submodes based on the following equivalences:

$$
m.c = \begin{cases}
\sum_{m_1 \in Sub(m)} m_1.c & \text{if } m \text{ is an OR-mode} \\
m_1.c & \text{if } m \text{ is an AND-mode and } m_1 \in Sub(m)
\end{cases}
$$

We can reduce the number of counters by eliminating the counters for the complex modes and inferring their values from the counters of their submodes based on the equivalences given above. To achieve this, in the functions SA^P and SD^P, we replace the constraints in the form $m.c > 0$, where m is a complex mode, with $Gt(m.c)$ which is defined as:

$$
Gt(m.c) = \begin{cases}
m.c > 0 & \text{if } m \text{ is a basic mode} \\
\bigvee_{m_1 \in Sub(m)} Gt(m_1.c) & \text{if } m \text{ is an OR-mode} \\
Gt(m_1.c) & \text{if } m \text{ is an AND-mode and } m_1 \in Sub(m)
\end{cases}
$$

Additionally, in the functions LU^P, LD^P, EnU^P, and EnD^P, for every complex mode m, we get rid of the $Inc(m.c)$ and $Dec(m.c)$ constraints. This reduction improves the efficiency of the verification by reducing the number of integer variables in the parameterized system.

4 Verification of HSMs Using ALV

Figure 3 shows the translation of the HSM for the airport ground control given in Figure 2 to Action Language. An Action Language specification consists of integer, boolean and enumerated variables, parameterized integer constants and a set of modules and actions which are composed using synchronous and asynchronous composition operators [13, 16]. Note that, a parameterized integer constant is an unspecified constant which can take an arbitrary integer value. Semantically, each Action Language module corresponds to a transition system with a set of states, a set of initial states and a transition relation. The top level module is always called the `main` module. The variable declarations of a module define the set of states of that module. The initial expression of a module defines the set of initial states of that module. A module expression (which starts with the name of the module) defines the transition relation of the module in terms of its actions and submodules using asynchronous and synchronous composition operators. Each action in an Action Language specification defines a single execution step. In an action expression primed variables denote the next-state values for the variables and unprimed variables denote the current-state values.

In Action Language asynchronous composition of two actions a_1 and a_2, denoted a_1 `|` a_2, is defined as the disjunction of their transition relations. However, an action preserves the values of the variables which are not modified by itself. Two actions a_1 and a_2 can also be combined with synchronous composition a_1 & a_2. Semantics of synchronous composition corresponds to conjunction if two actions are always enabled. However, if one component of synchronous composition is not enabled in a state, this would deadlock the composed system if conjunction is used as its semantics. To prevent this, in such a state, the disabled component makes a synchronous idle transition and stays in the same state which allows other components to progress. Formal semantics of Action Language is given in [26].

The Action Language translation of the HSM specification in Figure 2, shown in Figure 3, follows the HSM semantics we described in the previous sections. There is one enumerated variable to encode each OR-mode in the HSM specification. Each complex mode is represented with a module. Each transition in the HSM specification is represented by one action in the Action Language translation. Then, the overall transition relation is defined by combin-

```
module main()
  enumerated sr1, sr2, st1, st2, sg {empty, occupied};
  open boolean land, taxii1E, taxii2E, taxii2W, fly, park, takeoff;
  initial: land and !taxii1E and !taxii2E and !taxii2W and !fly and !park and !takeoff;
  module Airplane()
    enumerated state {flow, landing, taxiing1, taxiing2, takingoff, parking};
    initial: state=flow;
    a1: state=flow and sr1=empty and land and state'=landing and !land' and taxii1E';
    a2: state=landing and st1=empty and  taxii1E and  state'=taxiing1 and !taxii1E' and taxii2E';
    a3: state=taxiing1 and sr2=empty and st2=empty and sg=empty and taxii2E
        and state'=taxiing2 and !taxii2E' and park';
    a4: state=taxiing2 and sr2=empty and takeoff and state'=takingoff and fly' and !takeoff';
    a5: state=taxiing2 and sg=empty and park and state'=parking and taxii2W' and !park';
    a6: state=parking and st2=empty and taxii2W and state'=taxiing2 and takeoff' and !taxii2W';
    a7: state=takingoff and fly and state'=flow and !fly';
    Airplane: a1 | a2 | a3 | a4 | a5 | a6 | a7 ;
  endmodule
  module r1()
    initial: sr1=empty;
    r11: sr1=empty and land and !land' and taxii1E' and sr1'=occupied;
    r12: sr1=occupied and taxii1E and st1=empty and sr1'=empty and !taxii1E' and taxii2E';
    r1: r11 | r12;
  endmodule
  module t1()
    initial: st1=empty;
    t11: st1=empty and taxii1E and st1'=occupied and taxii2E' and !taxii1E';
    t12: st1=occupied and taxii2E and sr2=empty and st2=empty and
         sg=empty and  st1'=empty and park' and !taxii2E';
    t1: t11 | t12;
  endmodule
  module r2()
    initial: sr2=empty;
    r21: sr2=empty and takeoff and sr2'=occupied and fly' and !takeoff';
    r22: sr2=occupied and fly and sr2'=empty and !fly';
    r2: r21 | r22;
  endmodule
  module t2()
    initial: st2=empty;
    t21: st2=empty and taxii2E and sr2=empty and st2'=occupied and park' and !taxii2E';
    t22: st2=occupied and park and sg=empty and st2'=empty and taxii2W' and !park';
    t23: st2=occupied and takeoff and sr2=empty and st2'=empty and fly' and !takeoff';
    t24: st2=empty and taxii2W and st2'=occupied and takeoff' and !taxii2W';
    t2: t21 | t22 | t23 | t24;
  endmodule
  module g()
    initial: sg=empty;
    g1: sg=empty and park and sg'=occupied and taxii2W' and !park';
    g2: sg=occupied and taxii2W and st2=empty and sg'=empty and takeoff' and !taxii2W';
    g: g1 | g2;
  endmodule
  module EnvEvent()
    // this module generates the environment events nondeterministically
    EnvEvent: land'=land or land';
  endmodule
  module EventConstraint()
    // this module specifies that at any execution step at most one event
    // can be consumed and at most one event can be generated
    ...
  endmodule
  main: (Airplane()* & r1() & t1() & r2() & t2() & g() | EnvEvent()) & EventConstraint();
  spec: AG(EX(true))
  spec: AG(sr1=occupied and st1=occupied => AX(sr1=occupied))
  spec: AG(st1=occupied and (sr2=occupied or sg=occupied) => AX(st1=occupied))
endmodule
```

Figure 3. Action Language specification for the airport ground control model given in Figure 2.

ing the transitions (or submodes) of OR-modes with asynchronous composition |, and the submodes of AND-modes with synchronous composition &.

In the Action Language translation, the events are represented as boolean variables. The event variables are declared to be *open* which means that their updates have to be explicitly stated (i.e., they do not preserve their value unless it is explicitly stated). We declare an *event constraint* module which is synchronously composed with the rest of the system. This module restricts the transition relation so that at each execution step at most one event can be consumed and at most one event can be generated. We also declare an *environment event* module which is asynchronously composed with the HSM. This module nondeterministically generates the events that are generated by the environment. Although the Action Language specification in Figure 3 is generated manually, we believe that it can be automated based on the translation approach we followed for the example in Figure 2.

4.1 Parameterized Verification

The counting abstraction technique [19] is integrated to ALV in order to verify properties of parameterized systems with arbitrary number of finite state modules. In Action Language, a module can be marked to be parameterized which is denoted by the suffix "*". Note that, in the Action Language translation shown in Figure 3 the parameterized mode in the HSM specification in Figure 2 is marked to be parameterized. When a module is marked to be parameterized, ALV generates an abstract transition system in which the local variables of the parameterized module is replaced by a set of integer variables, one integer variable for each valuation of the local variables of the parameterized module. These integer variables keep track of the number of instances of the parameterized module in each local state (which corresponds to a valuation of the local variables). An additional parameterized constant is introduced to denote the number of instances of the parameterized module. Counting abstraction preserves the CTL properties that do not involve the local states of the abstracted processes. When properties of a system are verified using the counting abstraction, the result will hold for any number of instances of the parameterized module and if a counter-example is generated it corresponds to a concrete counter-example. Note that counting abstraction technique works only for modules with finite number of local states.

Counting abstraction technique may generate a large number of (unbounded) integer variables to encode the local states of the parameterized modules. Hence, efficient verification with integer variables is crucial for the scalability of parameterized verification. ALV specializes in verification of such systems.

ALV is a symbolic model checker for CTL which uses efficient symbolic representations for integer variables. ALV computes the truth set of a given temporal property based on the least and greatest fixpoint characterizations of CTL operators. It uses iterative fixpoint computations starting from the fixpoint for the innermost temporal operator in the formula.

ALV uses the Composite Symbolic Library [28, 29] as its symbolic manipulation engine. Composite Symbolic Library integrates multiple symbolic representations: BDDs for boolean and enumerated variables, polyhedral or automata representations for integer variables, and BDDs for bounded integer variables. Composite Symbolic Library provides an abstract interface which is inherited by every symbolic representation that is integrated to the library. Originally, ALV was developed using a Polyhedral representation for linear arithmetic constraints [15, 28]. Recently, it has been extended with an automata representation for linear arithmetic constraints [10, 11]. ALV also uses BDDs to encode boolean and enumerated variables. These symbolic representations can be used in different combinations. For example, polyhedral and automata representations can be combined with BDDs using a disjunctive representation. ALV also supports efficient representation of bounded arithmetic constraints using BDDs [9].

In the presence of unbounded integer variables (such as the ones generated by the counting abstraction) model checking is undecidable. Hence, ALV uses conservative approximation techniques during verification. There are three possible outcomes when one uses ALV to verify a parameterized system: 1) ALV verifies the property which means that the property is provably correct, 2) ALV generates a counter-example which means that the property is provably incorrect, and 3) ALV states that it is unable to verify or falsify the property. ALV uses several heuristics to minimize the occurrence of the third outcome as much as possible.

The undecidability of the model checking problem for unbounded systems implies that the fixpoint computations are not guaranteed to converge. ALV uses several conservative approximation heuristics to achieve convergence [12, 14, 15, 16]: 1) Truncated fixpoint computations to compute lower bounds for least fixpoints and upper bounds for greatest fixpoints, 2) Widening heuristics both for polyhedra [15] and automata representations [12] to compute upper bounds for least fixpoints (and their duals to compute lower bounds for greatest fixpoints), 3) Approximate reachability analysis using a forward fixpoint computation and widening heuristics, 4) Accelerations based on loop-closures which extract disjuncts from the transition relation that preserve the boolean and enumerated variables but modify the integer variables, and then compute approximations of the transitive closures of the integer part.

4.2 Experiments

After we translated the HSM model of the airport ground network traffic control in Figure 2 to the Action Language specification shown in Figure 3, we verified several correctness properties using ALV. The first five properties we verified were:

```
AG(EX(true))
AG(sr1=occupied and st1=occupied
       => AX(sr1=occupied))
AG(st1=occupied and
     (sr2=occupied or st2=occupied)
       => AX(st1=occupied))
AG(st2=occupied and sg=occupied and sr2=occupied
       => AX(st2=occupied))
AG(sg=occupied and st2=occupied
       => AX(sg=occupied))
```

We verified the five properties listed above both for concrete number of airplanes (2, 4, 8, 16, 32 and 64) and arbitrary number of (parameterized) airplanes. The first property denotes absence of deadlock. Other properties make sure that the airplanes follow the rules of the airport topology when they are moving across the runways and taxiways. We verified one more property indicating that at any reachable state of the system there is at most one airplane in the `taxiing2` state. To specify that property for the concrete cases we wrote invariants in the following form:

```
AG(state1=taxiing2 => state2!=taxiing2 and
                      state3!=taxiing2 and ...)
```

where `state`i denotes the `state` variable of the i'th instance of the `Airplane` module. Note that since each instantiation of the `Airplane` module is identical, proving this property ensures that the property holds for any instantiation. For the parameterized case, we declared an auxiliary integer variable `count` which is initialized to 0, and is incremented when an `Airplane` module enters the `taxiing2` state and is decremented when an `Airplane` module exits the `taxiing2` state. We then verified the property `AG(count<=1)`.

Table 1 shows the transition system construction time, verification time and memory usage. The first five rows show the results for the concrete cases with 2, 4, 8, 16, 32 and 64 instances of the `Airplane` module and the bottom row (denoted by P) shows the results for the parameterized case. For the experiments we used a machine with a 2.8 GHertz Pentium 4 processor and 2 GBytes of main memory.

For the concrete cases the specification is a finite state model and ALV works as a BDD based model checker (i.e., does not use any arithmetic constraint manipulation). The fixpoint computations for the first five properties converged in the first iteration for all cases. The fixpoint computation for the last property took 11 iterations for the case with 2 airplanes, 18 iterations for the case with 4 airplanes, 22 iterations for the cases with 8, 16, 32 and 64 airplanes, and 23 iterations for the parameterized case.

Number of Airplanes	Construction Time (sec)	Verification Time (sec)	Memory (MB)
2	0.08	0.02	1.68
4	0.21	0.16	4.63
8	0.56	1.08	15.75
16	1.34	3.24	39.80
32	3.25	9.69	64.45
64	10.25	26.21	124.35
P	41.32	13.85	15.15

Table 1. Verification results.

Note that the transition system construction time for the parameterized case takes longer than the concrete cases. There are two reasons 1) for the parameterized case the counting abstraction is being computed during the transition system construction and 2) the parameterized case is using linear arithmetic constraint manipulation during the transition system construction which is more expensive than boolean logic manipulation.

The verification time for the parameterized case is between the verification times for the concrete cases with 32 airplanes and 64 airplanes. In terms of memory usage parameterized case performs even better and uses less memory than the concrete case with 8 airplanes. More importantly, the result we obtain for the parameterized case is much stronger than any of the concrete cases. From the verification results for the parameterized case we can deduce that the airport ground traffic control model given in Figure 2 satisfies the properties listed above for any number of airplanes.

5 Conclusions

In this paper, we extended the standard notation for the hierarchical state machines by introducing primitives for explicit specification of asynchronous processes and their finite and parameterized instantiations. We used the counting abstraction technique in defining the semantics of the parameterized instantiation operator. We showed that hierarchical state machine specifications can be translated to the Action Language and their properties can be verified using the Action Language Verifier. The Action Language Verifier is an infinite-state symbolic model checker which can verify parameterized Action Language specifications using automated counting abstraction. We showed that this feature of the Action Language Verifier can be used to verify parameterized hierarchical state machine specifications automatically. As a case study, we modeled an airport ground traffic control system using the hierarchical state machines, translated the specification to the Action Language, and ver-

ified its properties both for different, concrete number of instantiations and for arbitrary number of instantiations using the Action Language Verifier.

References

[1] ALV: Action language verifier. Available at: `http://www.cs.ucsb.edu/~bultan/composite/`

[2] BRAIN: Backward reachability analysis with integers. Available at: `http://www.cs.man.ac.uk/~voronkov/BRAIN/`

[3] FAST: Fast acceleration of symbolic transition systems. Available at: `http://www.lsv.ens-cachan.fr/fast/`

[4] LASH: The Liège automata-based symbolic handler. Available at: `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`

[5] OMG's UML 1.5 specification. Object Management Group. Available at: `http://http://www.uml.org/`

[6] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proceedings of the 27th Symposium on Principles of Programming Languages*, pages 390–402, January 2000.

[7] R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 280–295, July 2000.

[8] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolic transion systems. In *Proceedings of the 15th International Conference on Computer Aided Verification*, pages 118–121, July 2003.

[9] C. Bartzis and T. Bultan. Construction of efficient BDDs for bounded arithmetic constraints. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 394–408, April 2003.

[10] C. Bartzis and T. Bultan. Efficient image computation in infinite state model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification*, pages 249–261, July 2003.

[11] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *International Journal of Foundations of Computer Science*, 14(4):605–624, August 2003.

[12] C. Bartzis and T. Bultan. Widening arithmetic automata. In *Proceedings of the 16th International Conference on Computer Aided Verification*, pages 321–333, July 2004.

[13] T. Bultan. Action Language: A specification language for model checking reactive systems. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 335–344, June 2000.

[14] T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, January 2000.

[15] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[16] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 382–386, November 2001.

[17] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[18] W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. E. Warner. Decoupling synchronization from local control for efficient symbolic model checking of statecharts. In *Proceedings of the 21st International Conference on Software Engineering*, pages 142–151, May 1999.

[19] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 53–68, July 2000.

[20] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Journal of Software Tools and Technology Transfer*, 3(3):250–270, 2001.

[21] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[22] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw Hill, 1998.

[23] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specifications of process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[24] T. Rybina and A. Voronkov. Using canonical representations of solutions to speed up infinite-state model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 400–411, 2002.

[25] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–19, April 2000.

[26] T. Yavuz-Kahveci. *Specification and Automated Verification of Concurrent Software Systems*. PhD thesis, University of California, Santa Barbara, 2004.

[27] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 169–179, July 2002.

[28] T. Yavuz-Kahveci and T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. *International Journal on Software Tools for Technology Transfer*, 5(1):15–33, November 2003.

[29] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 335–344, April 2001.