To answer this question, we first define an *external behavior* (or *behavior* for short) of an automaton. A behavior corresponding to an execution α is the subsequence of α consisting only of timed input and output actions, together with a *start time*. The start time of the behavior is equal to the time associated with the first state of α. A sequence β is said to be a behavior of automaton A if β is the behavior corresponding to some execution of A. Clearly any behavior of the automaton in Figure 2.4 will consist only of DELIVER_TOKEN and RETURN_TOKEN actions.

Using the notion of a behavior, we can define the "external" correctness of the token passing system as follows. A token passing system S is said to be correct if for any behavior β of S the following two properties hold in the sequence β:

- There must be a RETURN_TOKEN$_i$ between any DELIVER_TOKEN$_i$ and a later DELIVER_TOKEN$_j$ for any i, j.
- Suppose that in β, a RETURN_TOKEN$_i$ occurs in bounded time after every DELIVER_TOKEN$_i$ for all i. Then for any j and any suffix γ of β, a DELIVER_TOKEN$_j$ will occur in bounded time after the start of γ.

The first condition is a "safety" property. It guarantees that Process j will not receive the token until all processes that have received the token before j have returned the token. The second property is a "liveness property". It ensures that Process j will get the token periodically. However, we can only guarantee this property if all external clients return the token in bounded time after a token is delivered to them. Notice a modelling trick that is being used here. While an I/O automaton A must allow all possible inputs, we can specify that A exhibit correct behavior only on certain "well-formed" inputs.

2.2 Formal Summary of the I/O Automaton Model

We summarize our discussion so far. In this thesis, we will use the following model which is a special case of the model in [MMT91]. However, our terminology is slightly different from that of [MMT91].

An *automaton* A consists of five components:

- a finite set of actions *actions*(A) that is partitioned into three sets called the set of *input, output, and internal* actions. The union of the set of input actions and the set of output actions is called the set of *external* actions. The union of the set of output and internal actions is called the set of *locally controlled* actions.

- A finite set of states called *states*(A).

- A nonempty set *start*$(A) \subseteq$ *states*(A) of start states.

- A transition relation $R(A) \subseteq \mathit{states}(A) \times \mathit{actions}(A) \times \mathit{states}(A)$ with the property that for every state s and input action a there is a transition $(s, a, \tilde{s}) \in R(A)$.

- An equivalence relation *part*(A) partitioning the set of locally controlled actions into equivalence classes, such that for each class c in *part*(A) we have a positive real upper bound t_c. (Intuitively, t_c denotes an upper bound on the time to perform some action in class c.)

An action a is said to be *enabled* in state s of automaton A if there exist some $\tilde{s} \in \mathit{states}(A)$ such that $(s, a, \tilde{s}) \in R(A)$. An action a is *disabled* in state s of automaton A if it is not enabled in that state. Since one action may occur multiple times in a sequence, we often use the word *event* to denote a particular occurrence of an action in a sequence.

To model the passage of time we use a time sequence. A *time sequence* $t_0, t_1, t_2, \ldots$ is a non-decreasing sequence of non-negative real numbers; also the numbers grow without bound if the sequence is infinite. A *timed element* is a tuple (x, t) where t is a non-negative real and x is an element drawn from an arbitrary domain. A *timed state* for automaton A is a timed element (s, t) where s is a state of A. A *timed action* for automaton A is a timed element (a, t) where a is an action of A.

Let $X = (x_0, t_0), (x_1, t_1), \ldots$ be a sequence of timed elements. We will also use $x_j.\mathit{time}$ (which is read as the time associated with element x_j) to denote t_j.

We say that element x_j occurs within time t of element x_i if $j > i$ and $x_j.\mathit{time} \leq x_i.\mathit{time} + t$. We will use $X.\mathit{start}$ (which is read as the start time of X) to denote t_0.

Definition 2.2.1 *An* execution α *of automaton A is an alternating sequence of timed states and actions of A of the form* $(s_0, t_0), (a_1, t_1), (s_1, t_1), (a_2, t_2), (s_2, t_2), \ldots$ *such that the following conditions hold:*

1. $s_0 \in start$ *and* $(s_i, a_{i+1}, s_{i+1}) \in R$ *for all* $i \geq 0$.

2. *The sequence can either be finite or infinite, but if finite it must end with a timed state.*

3. *The sequence* $t_0, t_1, t_2, \ldots$ *is a time sequence.*

4. *If any action in any class* c *is enabled in any state* s_i *of* α *then within time* $s_i.time + t_c$ *either some action in* c *occurs or some state* s_j *occurs in which every action in* c *is disabled.*

Notice that the time assigned to any event a_i in α (i.e., $a_i.time$) is equal to the time assigned to the next state (i.e., $s_i.time$). Notice also that we have ruled out the possibility of so-called "Zeno executions" in which the execution is infinite but time stays within some bound.

Definition 2.2.2 *Consider an execution* α *of* A. *Let* γ *be the subsequence of* α *consisting of timed external actions, and let* t_0 *be the start time of* α. *The* behavior β *corresponding to* α *is the sequence* $\beta = t_0, \gamma$.

Notice that the start time of a behavior is the start time of the corresponding execution. The behaviors of automaton A are the behaviors corresponding to the executions of A.

Notice that a behavior is not the same "type" of sequence as an execution since a behavior consists of a start time followed by a sequence of timed actions. Formally, a *behavior sequence* β is a sequence $t_0, (a_1, t_1), (a_2, t_2), \ldots$ such that each a_i is drawn from some set of actions and such that $t_0, t_1, t_2, \ldots$ is a time sequence. Note that any behavior of an automaton is a behavior sequence. We will use $\beta.start$ to denote the first element in β; $\beta.start$ can be read as the start time of behavior sequence β. As before, we will use $a_j.time$ to denote t_j.

Consider an execution $\alpha = (s_0, t_0), (a_1, t_1), (s_1, t_1), (a_2, t_2), \ldots$. The untimed execution corresponding to α is the sequence $s_0, a_1, s_1, a_2, s_2, \ldots$. For brevity, we will frequently describe execution α by the corresponding untimed execution $s_0, a_1, s_2, \ldots$. By our notation, the time associated with any state s_i (or action a_j) in α is $s_i.time$ (or $a_j.time$).

Similarly consider a behavior $\beta = t_0, (b_1, t_1), (b_2, t_2), \ldots$ The untimed behavior corresponding to β is the sequence $b_1, b_2, \ldots$. We will often describe behavior β using the corresponding untimed behavior $b_1, b_2, \ldots$. Once again by our notation, the start time of β is $\beta.start$ and the time associated with any action b_j in β is $b_j.time$.

Notice that the time associated with the first state of an execution, $s_0.time$, is allowed to be an *arbitrary non-negative* real number. As we see below, this assumption allows a clean statement of an important lemma about stabilizing automata. In the timed automaton model [MMT91], each class also has an associated lower bound. In our model, the lower bound associated with each class is implicitly assumed to be zero. These two assumptions (or lack of assumptions) restrict us to modelling systems in which the value of time is not used by the protocol. In the protocols we describe later, we will use time only to model liveness guarantees and to measure time complexity.

The following lemma is useful later.

Lemma 2.2.3 *Consider any execution α of an automaton A. Any suffix of α that starts with a timed start state is also an execution of A.*

Proof: In essence, the lemma follows because we have no lower bounds on the time between actions. ∎

2.2.1 Composition and Hiding

To describe a collection of automata we will use a finite index set, say I. For example, an index set could be the set of vertices in a graph, or the set of edges in a graph. Thus we often speak of a collection $\{A_i, i \in I\}$ of automata, where I is an index set. Often we are interested in composing such a collection of automata.

Before automata can be composed they must obey certain restrictions. Clearly, the automata cannot share internal or output actions without violating the principle that each action is controlled by exactly one automaton. A collection of automata $\{A_i, i \in I\}$ is said to be *compatible* if the collection is finite, the output actions of all automata are disjoint, and the internal actions of any automaton are disjoint from the actions of any other automaton. Notice that this allows several automata to have a common input action. This can be used, for instance, to model a broadcast from one automaton to several other automata.

Let I be a finite index set. The composition of a compatible collection $\{A_i, i \in I\}$ is denoted by $A = \Pi_{i \in I} A_i$. A is the automaton formed as follows:

- An action π is an input action of A if π is the input action of some A_i in the collection and *is not* the output action of some other A_j in the collection. The set of output actions of A is the union of the output actions of the collection. The set of internal actions of A is the union of the internal actions of the collection.
- The set of states of A is the cross-product of the state sets of the automata in the collection. The set of initial states of A is the cross-product of the initial state sets of the automata in the collection.
- Let $s|i$ denote the projection of some state s of A onto automaton A_i. Then the transitions of A are the triples $(s, a, \tilde{s})$ such that for any $i \in I$:
  - If a is an action of A_i then $(s|i, a, \tilde{s}|i)$ is a transition of A_i.
  - If a is not an action of A_i then $s|i = \tilde{s}|i$.
- The partition of A is the union of the partitions in the collection. Thus if an action a belongs to some class c of any A_i, then a belongs to class c in A. The upper bound corresponding to class c in A is the upper bound corresponding to class c in A_i.

Note: Suppose we simply took the set of input actions of A to be the union of the set of input actions of the components. Then any action that is an input action of say A_i and an output action of say A_j will be classified both as an input and output action of A. It is to avoid this problem that the input actions of A are defined the way they are. Notice also that any action of the component automata is also an action of the composition.

We return to our claim that the I/O automaton model is *compositional*. We would like to show that the behavior of a composition can be inferred from the behavior of its components. We translate the following two lemmas from [MMT].

We use $\beta|A_i$ to represent the projection of a behavior β of A on to some constituent automaton A_i. The projection is the subsequence of β containing actions of A_i. We also assume that $\beta|A_i$ inherits the times of β in the natural way. Thus the time associated with any action in $\beta|A_i$ is the time associated with the corresponding action in β, and the start time of $\beta|A_i$ is the start time of β.

The first lemma shows how we can "cut" a behavior of the composition into behaviors of each of the pieces. The second lemma shows how we can "paste" a sequence of behaviors of the pieces into a behavior of the composition.

Lemma 2.2.4 Cut Lemma *Let $\{A_i, i \in I\}$ be a compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Let β be any behavior of A. Then $\beta|A_i$ is a behavior of A_i for every $i \in I$.*

Lemma 2.2.5 Paste Lemma *Let $\{A_i, i \in I\}$ be a compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Let β be a behavior sequence such that each action in β is an external action of A. If $\beta|A_i$ is a behavior of A_i for every $i \in I$, then β is a behavior of A.*

Finally, there is a hiding operator on automaton A with respect to some subset Σ of the actions of A. The result of this operation is to to create a new automaton that is identical to A except that all actions in Σ are reclassified as internal actions.

2.2.2 Useful Notation

In all the automata that we will describe in this thesis, the state of an automaton consists of values assigned to a set of variables. We use record notation to extract the values of specific variables in the state. We say that a variable x has a value v in state s if $s.x = v$. We sometimes omit the state s if it is clear by context which s we mean.

When we refer to the state s_i in execution α we mean the i-th state in the sequence α. An *interval* of an execution α is a contiguous subsequence of α that starts and ends with a timed state. The *duration* of an interval $[s_i, s_j]$ is $s_j.time - s_i.time$. An *interval* of a behavior β is a contiguous subsequence of β.

A *predicate* of automaton A is a subset of the states of A. A predicate S is described by a Boolean formula on variables; a state $s \in S$ iff the values of the variables of A in state s satisfy the Boolean formula. Thus if S is $x = 3$ then $s \in S$ iff $s.x = 3$. We also say that S holds in state s or S is true in state s to mean $s \in S$. We say that a predicate S remains true for time t after state s_i in execution α if for all states s_j that occur within time t of s_i in α, $s_j \in S$.

2.2.3 Modelling Asynchronous Protocols

In this thesis we will study "asynchronous" protocols. We wish such protocols to work regardless of timing assumptions. This is typically done using "fairness assumptions" instead of timing assumptions. However, we can model essentially the same thing by *ensuring that our protocols work regardless of the value of t_c assigned to any class c.*

The advantage of using parameterized upper bounds for classes comes in measuring time complexity. In the standard approach, after first proving correctness using "fair" executions, time complexity is then measured using the the assumption that the class times t_c are constants like 1 or 0. Often the time complexity arguments are extremely similar to the liveness arguments used in the proof of correctness. In our approach there is no need for this double effort; we replace liveness arguments by showing time bounds on certain events occurring. These time bounds are parameterized in terms of the class times t_c. Thus to obtain time complexity measures, we simply substitute particular values for t_c.

Chapter 3

Stabilization: Definitions and Properties

In this chapter we will describe the basic definitions of stabilization we will use for the rest of the thesis. We begin in the first section with a state-based definition of stabilization that corresponds to the standard definitions in the literature. In the next section we describe a new definition of stabilization in terms of external behaviors. In Section 3.4 we describe a two stage proof technique for proving stabilization properties. Then in Section 3.5 we describe a modularity result. This result shows that (under certain conditions) we can prove facts about the stabilization of a big system by proving facts about the stabilization of each of its parts.

3.1 Definitions of Stabilization based on Executions

All the existing definitions of stabilization are in terms of the states and executions of a system. We will begin with a definition of stabilization that corresponds to the standard definitions (for example, that of Katz and Perry [KP90]). In the next section, we will describe another definition of stabilization in terms of external behaviors. We believe that the definition of behavior stabilization is appropriate for large systems that require modular proofs. However, the definition of execution stabilization given below is essential in order to *prove* results about behavior stabilization. We begin with the definition of execution stabilization since it is also the definition that most readers are familiar with.

Suppose we define the correctness of an automaton in terms of a set C of legal executions. For example, recall that for the token passing system of Chapter 2 we defined the legal executions to be those in which there is exactly one token in every state, and in which every process periodically receives a token.

What do we mean when we say that an automaton A stabilizes to the executions in set C in time t? Intuitively, we mean that within time t, all executions of A begin to "look like" an execution in set C. For example, suppose C is the set of legal executions of a token passing system. Then in the initial state of A there may be zero or more tokens. However, the definition requires that within time t of the start of any execution of A, there is exactly one token in any state.

To formalize this, we begin with the definition of a t-suffix of an execution α. Intuitively, this is a suffix of α whose first element occurs no more than time t after the start of α. Although we will apply this definition only to executions, we will state the definition in terms of an arbitrary sequence of timed elements. Recall from Chapter 2 that a timed element is a tuple (x, t) where x is either a state or an action, and t is a time.

Definition 3.1.1 *Consider any two sequences of timed elements α and α'. We say that α' is a t-suffix of execution α if:*

- $\alpha'.start - \alpha.start \leq t$ *and*
- α' *is a suffix of* α.

We can now formally define execution stabilization to a set of executions:

Definition 3.1.2 *Let C be a set of sequences of timed elements. We say that automaton A stabilizes to the executions in C in time t if for every execution α of A there is some t-suffix of execution α that is in C.*

We also make the accompanying definition of execution stabilization to another automaton:

Definition 3.1.3 *We say that automaton A stabilizes to the executions of automaton B in time t if for every execution α of A there is some t-suffix of execution α that is an execution of B.*

So far, we have not made any assumptions about the automaton A that stabilizes to the executions specified by a set C or another automaton B. However, recall that in Dijkstra's original definition, an automaton is "self-stabilizing" if regardless of what state it starts in, an automaton "eventually" produces "legal" executions. Our definitions are more general than Dijkstra's definitions. To see this, we now extend Dijkstra's notion of "self-stabilization" to our timed setting.

As a stepping stone, for any automaton A we define $U(A)$ (which can be read as the unrestricted version of A) to be the automaton that is identical to A except that any state of A can be a start state of $U(A)$.

Definition 3.1.4 *For any automaton A, we let $U(A)$ denote the automaton that is identical to A except that $start(U(A)) = states(A)$.*

Next, we can say that an automaton A "self-stabilizes" in time t if the following holds: even if we start A in a state other than one of it's start states, the resulting execution will begin to "look like" a properly initialized execution of A within time t. Formally:

Definition 3.1.5 *We say that an automaton A self-stabilizes in time t if $U(A)$ stabilizes to the executions of A in time t.*

The following simple lemma shows that execution stabilization is transitive. This is an important lemma because it allows to prove execution stabilization in several stages.

Lemma 3.1.6 *If automaton A stabilizes to the executions of automaton B in time t_1 and B stabilizes to the executions of automaton C in time t_2, then A stabilizes to the executions of C in time $t_1 + t_2$.*

3.2 Definitions of Stabilization based on External Behavior

In Chapter 2, we argued that a major theme of the I/O Automaton model [LT89] is the focus on external behaviors of an automaton. For instance, the correctness of an automaton is specified in terms of its external behaviors. In Chapter 2 for instance, we showed how to specify the correctness of a token passing system without any reference to the state of the system. We did this by specifying the ways in which token delivery and token return actions can be interleaved

Thus it natural to look for a definition of stabilization in terms of external behaviors. We would also hope that such a definition would allow us to modularly "compose" results about the stabilization of parts of a system to yield stabilization results about the whole system.

Typically, the correctness of an IOA is specified by a set of legal behaviors P. An IOA A is said to *solve* P if the behaviors of A are contained in P. For stabilization, however, it is reasonable to weaken this definition and ask only that an automaton exhibit correct behavior after some finite time. In most of this thesis, we will use the behavior stabilization definitions for *specifying* the stabilization properties of a system.

As in the case of execution stabilization, we begin with the definition of a t-suffix of a behavior β. Intuitively, this is a portion of β that starts at time no more than t after the start of β. However, this is not as easy as defining a t-suffix of an execution. Recall that a behavior $\beta = (t_0, \gamma)$ consists of two components: a start time t_0 and a sequence of timed actions γ. Thus we cannot simply define a t-suffix of β to be a suffix of β as we did in the case of an execution. We would also like a t-suffix of β to be a behavior sequence: thus the t-suffix must have a start time as well as a sequence of timed actions.

Figure 3.1 is a pictorial view of a behavior β. The second row represents the sequence of actions of the behavior and the first row represents the times corresponding to each action as well as the start time of the behavior. The dashed line to the right of the start time represents an instant of time that occurs no more than time t after the start of the behavior. Now consider the behavior that starts at the time corresponding to the dashed line and consisting of all actions to the right of the dashed line. We will call such a behavior a t-suffix of behavior β. Intuitively, as we said before, this represents a portion of β. However the portion starts at time no more than t after the start of β.

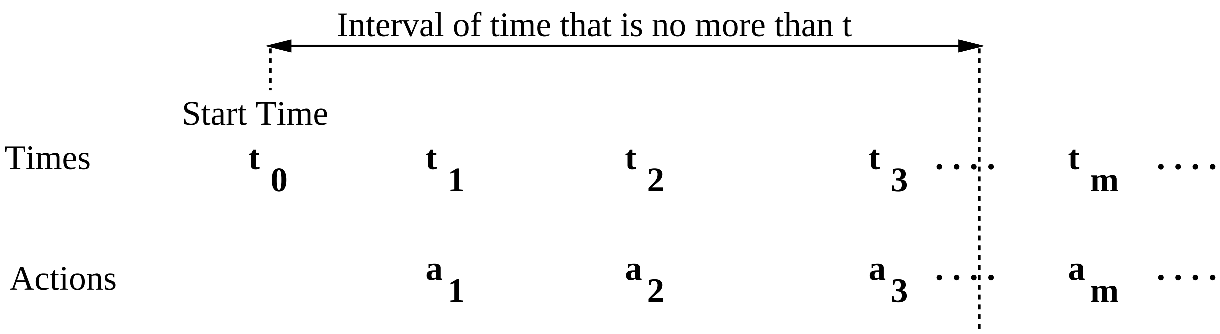

Figure 3.1: A pictorial view of a t-suffix of a behavior

Formally:

Definition 3.2.1 *Consider any two behavior sequences $\beta = t_0, \gamma$ and $\beta' = t_0', \gamma'$. We say that β' is a t-suffix of behavior β if:*

- *$\beta'.start - \beta.start \leq t$.*
- *γ' is a suffix of γ containing all actions in β that occur at times strictly greater than $\beta'.start$.*

Note that the definition allows γ' to contain some, none, or all of the actions in β that occur at times equal to $\beta'.start = t_0'$. Note that the definition is similar but yet different from the definition of a t-suffix of an execution. Using this, we can now define behavior stabilization analogous to execution stabilization:

Definition 3.2.2 *Let P be a set of behavior sequences. An IOA A stabilizes to the behaviors in P in time t if for every behavior β of A there is a t-suffix of behavior β that is in P.*

An automaton is said to *solve* another automaton B if every behavior of A is a behavior of B. Similarly, we can specify that A stabilizes to the behaviors of some other automaton B.

Definition 3.2.3 *An automaton A is said to stabilize to the behaviors of another automaton B in time t if for every behavior β of A there is a t-suffix of behavior β that is a behavior of B.*

The following lemmas are "obvious" in that they are what we expect to be true.

Lemma 3.2.4 *Every automaton stabilizes to its own behaviors in time* 0.

Lemma 3.2.5 *If every behavior of automaton* A *is a behavior of automaton* B*, then* A *stabilizes to the behaviors of* B *in time* 0.

The next lemma is a transitivity result for behavior stabilization. It allow us to prove behavior stabilization results in stages.

Lemma 3.2.6 *If automaton* A *stabilizes to the behaviors of automaton* B *in time* t_1 *and* B *stabilizes to the behaviors of automaton* C *in time* t_2*, then* A *stabilizes to the behaviors of* C *in time* $t_1 + t_2$.

Another obvious consequence of our definition is:

Lemma 3.2.7 *If automaton* A *stabilizes to the behaviors of an automaton* B *in time* t *and* $\tilde{t} \geq t$ *then* A *stabilizes to the behaviors of* B *in time* $\tilde{t}$.

The previous lemma motivates a natural complexity measure called the *stabilization time* from A to B. Intuitively, this is the smallest time after which we are guaranteed that A will behave like B. However, since we are dealing with a potentially infinite set of behaviors we have to be a little more careful.

Definition 3.2.8 *The stabilization time from* A *to* B *is the infimum of all* t *such that* A *stabilizes to the behaviors of* B *in time* t.

The next lemma is simple but important because it ties together the execution and behavior stabilization definitions. It states that execution stabilization implies behavior stabilization. In fact, the only method we know to *prove* a behavior stabilization result is to first prove a corresponding execution stabilization result, and then use this lemma. Thus the behavior and execution stabilization definitions complement each other in this thesis: the former is typically used for *specification* and the latter is often used for *proofs*.

Lemma 3.2.9 *If automaton A stabilizes to the executions of automaton B in time t then automaton A stabilizes to the behaviors B in time t.*

Proof: Let β be any behavior of A. Let α be some execution of A corresponding to β. From the hypothesis, there is some α' that is a t-suffix of execution α and is also an execution of B. Consider the behavior β' of B corresponding to execution α' of B. From the definitions, we can verify that that β' is a t-suffix of behavior β. ∎

3.3 Discussion on the Stabilization Definitions

First, notice that we have defined what it means for an arbitrary IOA to stabilize to some target set or automaton. For most of the thesis we will be interested in proving stabilization properties only for a special kind of automata: unrestricted automata. An unrestricted IOA (see Section 3.5) is one in which all states of the automaton are also start states. Such an IOA models a system that has been placed in an arbitrary initial state by an arbitrary initial fault. However, (this important observation is due to Arora and Gouda [AG92]), we might also be interested in modelling the response of a system to more restricted kinds of initial faults. Such restricted faults initially place a system A in some subset L of the states of A. After the initial fault, we would like A to behave like some other automaton B. *Thus our general definitions of stabilization are applicable to other, more restricted forms of initial faults.* While we will not mention this explicitly from this point on, many of the techniques developed in this thesis can also be applied to more restricted initial fault models.

Next, it is reasonable to ask whether our definitions are sufficient to cover all cases that arise in practice? They do cover all the examples in this thesis. There are two places we can weaken our definitions. The first is that instead of requiring that *all* behaviors (or executions) of A stabilize in time t, we might only that certain "well-formed" behaviors (or executions) of A stabilize. A "well-formed" behavior (execution) is a behavior (execution) with certain restrictions on the input actions. For instance, if A can pass a token to the external environment E, we might only require stabilization for those behaviors (executions) in which E returns the token to A in bounded time.

The second possible modification (which is sometimes needed, though again not in this thesis) is to only require (in Definitions 3.2.2 and 3.1.2) that the t-suffix of a behavior (execution) be a *suffix* of a behavior (execution) of the target set. One problem with this modified definition is that we know of no good proof technique to prove that the behaviors (executions) of an automaton are *suffixes* of a specified set of behaviors (executions).[1]By contrast, it is much easier to prove that every behavior of an automaton has a suffix that is in a specified set.

Thus we prefer to use the simpler definitions for this thesis.

3.4 Proof Technique

We begin by defining an extremely useful piece of notation that is used extensively in this thesis. This notation allows us to specify an IOA that is identical to another IOA except for start states. This is roughly the inverse of the $U(A)$ notation that creates an unrestricted version of automaton A without start states.

Definition 3.4.1 *For any automaton A and any subset L of states(A), we denote by $A|L$ the automaton that is identical to A except that start$(A|L) = L$.*

There has been a great deal of work in designing ordinary automata that have specific start states to solve specific problems. It would be nice to gain leverage from this existing body of work. Suppose we are given an IOA A that solves a set of behaviors P and we now wish to design an IOA B that stabilizes to the behaviors in P. Our goals are:

- We would like to use as much of the design of A as possible to design B.
- We would like to use as much of the proof that A solves P as possible to prove that B stabilizes to the behaviors in P.

We now describe one way in which these goals can be achieved. The following lemma is immediate from the definitions.

Lemma 3.4.2 *Consider an automaton A, $L \subseteq states(A)$, and a problem P. Suppose the following conditions are true:*

[1]Such proofs seem to involve arguments about reachable states. Familiar inductive proof techniques (such as invariant arguments, progress metrics etc.) do not seem to suffice for this purpose.

- *A stabilizes to the behaviors of $A|L$ in time t.*
- *Any behavior of $A|L$ is in P.*

Then A stabilizes to the behaviors in P in time t.

In the next two subsections, we describe the techniques used in this thesis for proving the two items on the list.

3.4.1 Proving that an Automaton Solves a Problem

To prove that any behavior of some automaton A is a behavior contained in some problem P, it suffices to prove that every behavior of A is a behavior of some other automaton B, and that every behavior of B is in P.

There are well-known techniques (e.g., [LT89]) to show any behavior of an automaton A is a behavior of another automaton B. A commonly used technique is a refinement mapping. The basic idea is to establish a suitable mapping f between a state of A and a state of B. Given an execution α of A, we use f to construct a mapped execution of B that has the same external behavior as α.

Theorem 3.4.3 Refinement Mapping: *Let A and B be automata with the same set of external actions. Let t_c be the upper bound on the time to perform actions in any class c of B. Let f be a mapping from the states of A to the states of B such that:*

1. *For any start state s of A, $f(s)$ is a start state of B.*
2. *For all transitions $(s, \pi, \tilde{s})$ of A, either*
   - *π is an action of B and $(f(s), \pi, f(\tilde{s}))$ is a transition of B OR*
   - *π is not an action of B and $f(s) = f(\tilde{s})$.*
3. *For any class c of B and any execution α of A, suppose some action in c is enabled in $f(s)$. Then within t_c time of s in α either:*
   - *Some action in class c occurs OR*

- *Some state $\tilde{s}$ occurs such that no action in class c is enabled in $f(\tilde{s})$.*

Then every behavior of A is a behavior of B

Proof: Consider any execution α of A. We extend the function f to map executions as well as states in the following way. Let $f(\alpha)$ be the sequence formed from α by:

- Removing every timed action in α that is not a timed action of B and the timed state following such an action.
- Replacing every timed state (s,t) in the remaining sequence by $(f(s),t)$.

Since we retain every action of B, the behavior corresponding to $f(\alpha)$ is the behavior corresponding to α.

Next, we verify that $f(\alpha)$ is an execution of B. To do so we check the four conditions in Definition 2.2.1. Let $\alpha = (s_0,t_0),(a_1,t_1),(s_1,t_1)\ldots$ Clearly, by construction, $f(\alpha)$ is an alternating sequence of timed states and actions of B. Let $f(\alpha) = (s'_0,t'_0)(a'_1,t'_1)(s'_1,t'_1)\ldots$.

We check the four conditions in turn:

1. First $s'_0 = f(s_0) \in start(B)$ by hypothesis. Next consider $(s'_i, a'_{i+1}, s'_{i+1})$ for all $i \geq 0$. Let a_k be the $i+1$'st action of B in α. Intuitively, a_k is the action in α that generated a'_{i+1} in $f(\alpha)$. Clearly, $a_k = a'_{i+1}$.

   Next, consider the smallest $j < k$ such that all actions between s_j and s_{k-1} in α are not actions of B. Then by construction, $f(s_j) = s'_i$ and $f(s_k) = s'_{i+1}$. Also by the hypothesis, $f(s_j) = f(s_{k-1})$ and $(f(s_{k-1}), a_k, f(s_k))$ is a transition of B. Thus $(s'_i, a'_{i+1}, s'_{i+1})$ is a transition of B.

2. The second condition follows trivially from the construction.

3. The third condition follows because any subsequence of a time sequence is a time sequence.

4. Suppose some action in some class c of B is enabled in some state s'_i of $f(\alpha)$. Then, by construction there is a corresponding state s_j in α such that if γ is the suffix of α starting with s_j, then $f(\gamma)$ is the suffix of $f(\alpha)$ starting with s'_i. Then, by hypothesis, either:

- Some action in class c of B occurs within t_c time of s_j in α. But by construction, the same action occurs within t_c time of s'_i in $f(\alpha)$.
- Some state s_k occurs within t_c time of s_j in α and such that all actions in class c are disabled in $f(s_k)$. Let k be the smallest index such that this is true for s_k. Now if a_k is not an action of B, then since $f(s_{k-1}) = f(s_k)$ this contradicts the assumption that k is the smallest index with this property. Thus a_k is an action of B and hence, by construction, $(f(s_k), s_k.time)$ occurs in $f(\gamma)$. Thus there is a state that occurs within t_c time of s'_i in $f(\alpha)$ and such that no action of c is enabled in this state.

∎

3.4.2 Proving that an Automaton Stabilizes to another Automaton

We give one such technique in the following definition and theorem. The technique is similar to techniques used for proving liveness properties (e.g., [OL82, MP91]) of concurrent programs. Our theorem is a small generalization of a theorem for proving stabilization properties that was previously proposed in [GM90].

Let L be a closed predicate of automaton A – once L is true in an any execution of A, L remains true for the rest of the execution. We would like to prove that in any execution of A, L becomes (and stays) true in bounded time. This implies that A stabilizes to the executions of $A|L$ in bounded time. We will describe a proof rule for this purpose. Intuitively, instead of proving directly that the goal L eventually holds we prove that a number of subgoals L_i (each of which is a predicate of A) become and stay true. The L_i are chosen so that L is true if all the individual L_i are true.

Each subgoal L_i can be intuitively thought of as "depending" on some other set of subgoals. This dependence relation is formalized by a partial order $<$. If $j < i$, then (intuitively) L_i "depends" on L_j. Suppose we could show that if all the subgoals that L_i depend on become and stay true, then L_i becomes and stays true. Then we can concude that eventually all subgoals (and hence L) become and stay true. Intuitively, this follows because the dependency relation is *acyclic* since it is formalized using a partial order.

We can extend this idea to a timed setting to show that L will become true in time proportional to the maximum length of a "dependency chain". A dependency chain is formalized using the standard concept of a *chain* in a partially ordered set. Consider a set $\{L_i, i \in I\}$ where $(I, <)$ is a finite partially ordered set. A chain is simply a sequence $L_1 < L_2 < \ldots < L_i$.

The preceding discussion motivates the following definition and theorem.

Definition 3.4.4 *We say that an automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t if:*

1. *$\mathcal{L} = \{L_i, i \in I\}$ of sets of states of A, where $(I, <)$ is a finite, partially ordered index set. We let height($\mathcal{L}$) denote the maximum length chain in the partial order.*
2. *$\cap_{i \in I} L_i \subseteq L$.*
3. *For all $i \in I$ and for all steps $(s, \pi, \tilde{s})$ of A, if s belongs to $\cap_{j \le i} L_j$, then $\tilde{s}$ belongs to L_i.*
4. *For every $i \in I$ and every execution α of A and every state s in α the following is true. Suppose that either $s \in \cap_{j<i} L_j$ or there is no $L_j < L_i$. Then there is some state $\tilde{s} \in L_i$ that occurs within time t of s in α.*

The first condition says there is a partial order on the predicates in $\mathcal{L}$. The second says that L becomes "true", when all the predicates in $\mathcal{L}$ become true. The third is a stability condition. It says that any transition of A leaves a predicate L_i true, if all the predicates less than or equal to L_i are true in the previous state. Finally the last item is a liveness condition. It says that if all all the predicates *strictly* less than L_i are true in a state, then within time t after this state, L_i will become true.

We define *height*(L_i), the height of a predicate $L_i \in \mathcal{L}$, to be the maximum length of a chain that ends with L_i in the partial order. The value of *height*($\mathcal{L}$) is, of course, the maximum height of any predicate $L_i \in \mathcal{L}$. By the liveness condition, within time t all predicates with height 1 become true; these predicates stay true for the rest of the execution because of the third stability condition. In general, we can prove by induction that within time $i \cdot t$ all predicates with height i become and stay true. This leads to a simple but useful theorem:

Theorem 3.4.5 Execution Convergence: *Suppose that automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t. Then, A stabilizes to the executions of $A|L$ in time height($\mathcal{L}$) $\cdot\, t$.*

Proof: By induction on $h, 0 \leq h \leq height(\mathcal{L})$, in the following inductive hypothesis.

Inductive Hypothesis: There is some state s that occurs within time $h \cdot t$ of the start of α such that $s \in L_i$ for all $L_i \in \mathcal{L}$ with $height(L_i) \leq h$.

The inductive hypothesis implies that there is some state s that occurs within time $h \cdot height(\mathcal{L})$ of the start of α and such that $s \in L$. The theorem follows from this fact taken together with Lemma 2.2.3.

Base Case, $h = 0$: Follows trivially since there is no $L_i \in \mathcal{L}$ with $height(L_i) = 0$.

Inductive Step, $0 \leq h \leq height(\mathcal{L}) - 1$: Assume it is true for h. Then there is some state s_i that occurs within time $h \cdot t$ of the start of α and such that for all $L_i \in \mathcal{L}$ with $height(L_i) \leq h$, $s_i \in L_i$. Consider any $L_j \in \mathcal{L}$ with $height(L_j) = h + 1$. By the fourth condition in Definition 3.4.4, we know there must some state $s_{f(j)} \in L_j$ that occurs within time t of s_i in α. Let $k = Max\{f(j) : height(L_j) = h + 1\}$. Then from the third condition in Definition 3.4.4, we see that s_k occurs within time $(h + 1) \cdot t$ of the start of α and such that for all $L_i \in \mathcal{L}$ with $height(L_i) \leq h + 1$, $s_i \in L_i$. ∎

3.5 Modularity Theorem

We will mostly deal with stabilization properties of a special class of automata called unrestricted automata or UIOA. Intuitively, a UIOA models a system that can start in an arbitrary state.

Definition 3.5.1 *A UIOA A is an automaton such that $start(A) = states(A)$ (i.e., all states are start states).*

However, we will often show that a UIOA A stabilizes to the behaviors of a second special kind of automaton called a Closed I/O Automaton or a CIOA . We define the *reachable states* of an automaton A to be the states that can occur in executions of A.

Definition 3.5.2 *A CIOA is an automaton such that every reachable state is also a start state.*

It is easy to see that:

Lemma 3.5.3 *Every UIOA is a CIOA .*

The two following lemmas are extremely convenient and are used often below without explicit reference. It is the reason we allow executions and behaviors to start with arbitrary values of time. Also, the next two lemmas depend crucially on the fact that there are no lower bounds on the time between actions.

Lemma 3.5.4 *Consider any execution α of a CIOA A. Any suffix of α that starts with a timed state is also an execution of A.*

Proof: Follows directly from Lemma 2.2.3 and the definition of a CIOA. ∎

Suppose we begin to view an automaton after it has "run for a while" and the resulting behavior is indistinguishable from an ordinary behavior of the automaton. Then, intuitively, we say that the automaton is *suffix-closed*. More formally:

Definition 3.5.5 *We say that an automaton A is suffix-closed if for every behavior β of A and every time $t \geq 0$, every t-suffix of behavior β is a behavior of A.*

A remarkable number of interesting automata we will study in this thesis are suffix-closed. This fact is explained by the following lemma:

Lemma 3.5.6 *Any CIOA A is suffix-closed.*

Proof: We will only sketch the main idea of the proof. Consider any behavior β of A and any $t \geq 0$. Let β' be any t-suffix of behavior β. Consider any execution α of A such that the behavior of α is β. The proof consists of using α to construct another execution α' of A such that the behavior of α' is b'; α' is essentially a suffix of α whose start time is adjusted to match the start time of β'.

The behavior β and corresponding execution α are sketched in Figure 3.2. By definition, for every action in β there is a corresponding external action in α which occurs at the same time. This is sketched by drawing the action in the behavior directly above the corresponding action in the execution. (However, since the execution will, in general, have internal actions not included in the behavior, the indices of the actions will not necessarily match. Thus in the figure a_1 in β corresponds to a_p in α.)

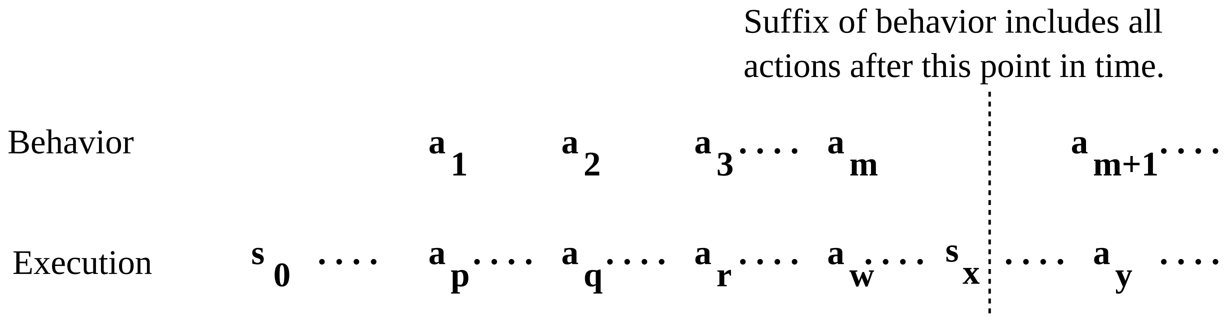

Figure 3.2: Obtaining the suffix of an execution corresponding to a t-suffix of the behavior of the execution.

The t-suffix β' can be sketched using a line that contains all actions in β occuring to the right of the line (see Figure 3.2). The line is drawn between two actions in β because the start time of β' may occur in between the times of two actions in β.

We need a suffix α' of α whose behavior is equal to β'. Thus we look for a state s_x in α corresponding to the vertical time line drawn in Figure 3.2. But we may not have a state in α whose time is equal to the start time of β'. So (intuitively) we choose s_x to be the first state that occurs to the "left" of the vertical time line. Then we choose α' to be the suffix of α starting with s_x and with the time of s_x adjusted to be equal to $\beta'.start$. This works for two reasons. First, by definition of a CIOA, s_x is a start state of A. Second, we have no lower bounds on the time between actions in A. Thus increasing the time of the initial state of an execution (and such that the resulting time is no greater than the time of the first action) still leaves us with a legal execution. ∎

The suffix-closed property is not just a interesting curiousity. It also provides the basis for the following important Modularity Theorem that we discuss next.

Our Modularity Theorem about the stabilization of composed automata may seem "obvious". We would expect that if each piece A_i of a composed system stabilizes to the behaviors of say B_i, then the composition of the A_i should stabilize to the composition of the B_i. Sadly, this is not quite true. There is a counterexample described in Section 3.5.1 which shows that if we allow some of the B_i to be arbitrary automata, then this statement is false.

The main problem is that for a given behavior of the system A, the component automata A_i may stabilize at different times. But if each of the A_i begin to "look like" the corresponding B_i at different times, then it may not be possible to paste the resulting behavior into a behavior that "looks like" a behavior of B. However, this problem does not arise if each of the B_i is suffix-closed. Thus we have the following result.

Theorem 3.5.7 Modularity: *Let I be a finite index set. Let $A_I = \{A_i, i \in I\}$ be a compatible set of automata and $B_I = \{B_i, i \in I\}$ be a second set of compatible, suffix-closed automata. Suppose also that for all $i \in I$, A_i stabilizes to the behaviors of B_i in time t. Let $A = \Pi_{i \in I} A_I$ and $B = \Pi_{i \in I} B_I$. Then A stabilizes to the behaviors of B in time t.*

Proof: The proof relies on the Cut Lemma (Lemma 2.2.4) that allows us to dissect a behavior of a system into its component behaviors, and the Paste Lemma (Lemma 2.2.5) that allows us to paste component behaviors into a system behavior.

Consider any behavior β of A. Consider the β' that is a t-suffix of behavior β and such that:

- $\beta'.start = t + \beta.start$.
- Any actions in β' occur at times strictly greater than t.

We can verify that such a β' exists from the definition of a t-suffix of a behavior. Intuitively, β' is chosen so that all component behaviors are guaranteed to have stabilized in β'.

Now consider any $i \in I$. By the Cut Lemma (Lemma 2.2.4), $\beta|A_i$ is a behavior of A_i. But because A_i stabilizes to the behaviors of B_i in time t, there must be some $t' \leq t$ and some β_i such that:

- β_i is a t'-suffix of $\beta|A_i$ and is also a behavior of B_i.
- $\beta_i.start = t' + \beta.start$.

Next consider $\beta'|A_i$. It can be verified that $\beta'|A_i$ is a t''-suffix of β_i for some t''. Thus by the fact that B_i is suffix-closed, $\beta'|A_i$ is a behavior of B_i. Thus $\beta'|A_i$ is a behavior of B_i for all i. Hence by the Paste Lemma (Lemma 2.2.5), β' is a behavior of B. The theorem follows since β' is a t-suffix of behavior β. ∎

The state of B_i consists of an integer variable $count_i \in \{0, \ldots, 2\}$

The initial value of $count_i = 0$ (* i.e., B is not a UIOA *)

INCREMENT$_i(k)$ (*output action, outputs counter value using parameter k*)
Precondition: $k = count_i$
Effect: $count_i := (count_i + 1) \bmod 3$

Any INCREMENT$_i$ action is in a separate class with upper bound t.

Figure 3.3: Specification for Automaton B_i

3.5.1 Discussion of the Modularity Theorem

In the hypothesis of the modularity theorem, we assumed that each of the B_i was suffix-closed. We show a counterexample to show that if the B_i are allowed to be arbitrary automata, then the theorem is false. Consider the specification of automaton B_i shown in Figure 3.3. Let A_i be a UIOA which is identical to B_i except that the start states of A_i are unrestricted (i.e., the initial value of $count_i$ in A can be any value in the range $\{0, \ldots, 2\}$).

It is easy to see that A_i stabilizes to the behaviors of B_i in time $3t$ because within that time the value of $count_i$ must reach 0. After such a state, any behavior of A_i is a behavior of B_i. Now consider an index set $I = \{1, 2\}$. Consider A which is the composition of A_1 and A_2 and B which is the composition of B_1 and B_2. We claim that A does not stabilize to B in time $3t$ (or in fact in any finite time).

To see this, we start with the following observation. In any behavior of B in which the actions of B_1 and B_2 strictly alternate, the counter values output in such a behavior will be of the form $0, 0, 1, 1, 2, 2, 0, 0, \ldots$. Now consider the behavior corresponding to an execution of A in which $count_1 = 0$ initially and $count_2 = 2$ initially and the actions of A_1 and A_2 strictly alternate starting with A_1. Then the counter values output in such a execution will be of the form $0, 2, 1, 0, 2, 1, 0, 2 \ldots$. From the earlier observation, it follows that there is no suffix of this behavior of A that is a behavior of B.

Suppose we weakened the definition of stabilization to allow any t-suffix of A to be a suffix of a behavior of B. With this weaker definition, the counterexample disappears.[2] Thus the suffix-closed requirement may be a consequence of our (stronger) stabilization definition. However, we did not find a way to prove the modularity theorem using the weaker definition. Without the suffix-closed requirement it is difficult to "paste" together behavior suffixes of the B_i's to create a behavior of B. A possible research direction would be to look for weaker conditions (than the B_i being suffix-closed) under which the modularity theorem would work. Another research direction would be to extend these results to systems with non-zero lower bounds on the time between actions.

3.6 Summary

The two main contributions of this chapter are the definitions of stabilization in terms of external behaviors (Definitions 3.2.2 and 3.2.3) and the modularity theorem (Theorem 3.5.7).

The definitions of stabilization in terms of external behaviors are different from previous definitions that are in terms of the states and executions of the underlying automaton. The external behavior definition allow us to define that automaton A stabilizes to another automaton B even though A and B have different state sets. This is most useful when A is a low level model (e.g., an implementation) and B is a high level model (e.g., a specification).

The modularity theorem gives us a formal basis for a building-block approach: it allows us to prove facts about the stabilization of a big system by proving facts about the stabilization of each of its parts. The requirement that each of the target automata be suffix-closed may seem restrictive. However, in a stabilizing setting this is not the case. Most interesting specifications (for problems such as message delivery, routing, and scheduling) are either suffix-closed or can be rephrased so they are suffix-closed.

We have already seen that any UIOA (an automaton for whom every state is a start state) is suffix-closed. Similarly, any CIOA (an automaton for whom every reachable state is a start state) is suffix-closed. In a stabilizing setting the basic building blocks

[2] I am grateful to Robert Gallager and Victor Luchanko of MIT for pointing this out.

are UIOAs since they model systems that can start in an arbitrary state. The methods developed in this thesis, on the other hand, tend to construct automata that are CIOAs.[3] Thus we can build stabilizing solutions modularly in several stages. In the first stage we compose s set of UIOAs to yield a CIOA. In the next stage, the resulting CIOAs are composed with other UIOAs to yield more CIOAs. This process can be repeated indefinitely to build a complex stabilizing solution to a problem. Since all the pieces used are suffix-closed, the modularity theorem can be used at each stage. As an example, the stabilizing spanning tree protocol of Chapter 8 is constructed using the stabilizing reset protocol of Chapter 7 which in turn can be constructed using a stabilizing Data Link implementation. Thus, despite its restrictions, the modularity theorem is extremely useful in this thesis.

We also have a definition of stabilization in terms of executions that corresponds to the standard definition (Definition 3.1.2). This definition in terms of executions is used in this thesis for two reasons. First, it is sometimes useful in its own right when the alternative would entail adding many superfluous actions.[4] Second, the definition in terms of executions is *essential for proving* results about stabilization of behaviors because our main tool for proving such results is Lemma 3.2.9.

The definitions we use give us several nice properties that we believe any definition of stabilization should provide. The properties we believe to be important are: transitivity for both behavior and execution stabilization (Lemma 3.2.6 and Lemma 3.1.6), the fact that execution stabilization implies behavior stabilization (Lemma 3.2.9), the fact that stabilization in time t implies stabilization in time greater than t (Lemma 3.2.7), and the Modularity theorem (Theorem 3.5.7).

The index contains pointers to the definitions given in the last two chapters. The appendix also contains a list of commonly used notation for easy reference.

[3] Our methods construct automata that stabilize to a specification automaton of the form $A|L$. If L is a closed predicate of A – i,e., no transition of A can falsify L – then $A|L$ is a CIOA.

[4] For example, the correctness of a spanning tree protocol is easily defined in terms of a *parent* variable at each node. For correctness, we could specify that the graph induced by the *parent* variables be a spanning tree of the network. An external behavior specification would require additional output actions to report the value of the *parent* variables at each node.