Trace Algebra for Automatic Verification of Real-Time Concurrent Systems

Jerry R. Burch

August 1992

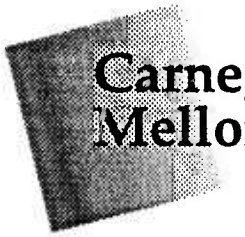
CMU-CS-92-179

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the National Science Foundation under Contract No. CCR-9005992.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

Keywords: Trace Algebra, Trace Structure Algebra, Conservative Approximation, Formal Verification, Abstraction, Real-Time, Continuous Time, Discrete Time, Speed-Dependent Asynchronous Circuits



School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Trace Algebra for Automatic Verification of Real-Time Concurrent Systems

JERRY R. BURCH

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Edmund M. Clarke Jr — MAJOR PROFESSOR — August 11, 1992 — DATE

R.T.R.M — DEAN — 8/21/92 — DATE

APPROVED:

PROVOST — 25 August 1992 — DATE

Abstract

Verification methodologies for real-time systems can be classified according to whether they are based on a continuous time model or a discrete time model. Continuous time often provides a more accurate model of physical reality, while discrete time can be more efficient to implement in an automatic verifier based on state exploration techniques. Choosing a model appears to require a compromise between efficiency and accuracy.

We avoid this compromise by constructing discrete time models that are *conservative approximations* of appropriate continuous time models. Thus, if a system is verified to be correct in discrete time, then it is guaranteed to also be correct in continuous time. We also show that models with explicit simultaneity can be conservatively approximated by models with interleaving semantics.

Proving these results requires constructing several different domains of agent models. We have devised a new method for simplifying this task, based on abstract algebras we call *trace algebra* and *trace structure algebra*. A trace algebra has a set of *traces* as its carrier, along with operations of *projection* and *renaming* on traces. A trace can be any mathematical object that satisfies certain simple axioms, so the theory is quite general. A *trace structure* consists, in part, of a subset of the set of traces from some trace algebra. In a *trace structure algebra*, operations of *parallel composition*, *projection* and *renaming* are defined on trace structures, in terms of the operations on traces. General methods for constructing conservative approximations are described and are applied to several specific real-time models. We believe that trace algebra is a powerful tool for unifying many models of concurrency and abstraction beyond the particular ones described in this thesis.

We also describe an automatic verifier based on the theory, and give examples of using it to verify speed-dependent asynchronous circuits. We analyze how several different delay models, including a new model called *chaos delay*, affect the verification results. The circuits and their specifications are represented in discrete time, but because of our conservative approximations, circuits that are verified correct are also correct in continuous time.

Acknowledgements

Ed Clarke has been my advisor during the last several years of my graduate student career. His guidance and support (and patience!) were essential to the completion of this thesis and the other research projects I have been involved in at CMU. Ed has taught me a great deal about formal verification and about how to do quality research.

The other members of my committee, Randy Bryant, Jeannette Wing, Al Mok and David Dill, provided many helpful ideas for improving my research and my writing. When my thesis research was not properly focused, Ed and the rest of the committee provided the support and the firm pressure that was necessary to get me back on track.

Others who have contributed to my development as a student and a researcher include Jon Doyle, Alain Martin and Fred Thompson.

Much of my research at CMU was in collaboration with David Long and Ken McMillan. I am very fortunate to have had a chance to work with them.

The School of Computer Science at CMU is a great place to be a graduate student, in no small part because of the people responsible for administration and facilities. My officemates, past and present, also helped create an intellectually stimulating environment.

I thank my parents, and the rest of my family, for never letting me doubt their love and support.

Contents

Chapter 1

Introduction

Modeling and verifying concurrent systems has grown into an important field of computer science. Several different categories of concurrent systems have been studied, including parallel programs, communication protocols and circuits. Over the last several years there has been increasing interest in modeling and verifying real-time systems. For our purposes, a real-time system is any system that, to be formally verified to satisfy its specification, must be modeled with explicit reference to quantitative time. Thus, if a system's specification is *timed* (constrains the time between events rather than just their order), then it is a real-time system. Another case is if the specification is untimed, but the correct operation of the system depends on timing assumptions about its components (such as an asynchronous circuit that is not speed-independent).

There are a large number of different real-time models in the literature. They can be classified according to whether they are continuous time models or discrete time models. Continuous time often provides a more accurate model of physical reality, while discrete time can be more efficient to implement in an automatic verifier based on state exploration techniques. Choosing a model appears to require a compromise between efficiency and accuracy.

We show how to avoid this compromise by taking advantage of the relationships between several different real-time models. All of the models we use are based on *trace structures*, which consist of sets of input and output events, and a set of *traces*. Each trace represents a possible behavior of the agent modeled by the trace structure.

There are many different kinds of traces, each is a different abstraction of physical behaviors. For example, with speed-independent interleaving semantics, traces are strings (from some formal language) that abstract time to be just a total order on events. Partial order based methods provide a different abstraction for behaviors by replacing total orders on events with

partial orders. In real-time models, traces include quantitative information about the time at which events occur.

We want to be able to use all of the above kinds of traces, as well as many other kinds, when modeling agents. Thus, the kind of trace that is used is a parameter in our method. Any mathematical object that satisfies certain minimum requirements can be used as a trace. These requirements are formalized as the axioms of *trace algebra*. A trace algebra has a set of traces as its domain, and defines the operations of *projection* and *renaming* (and sometimes *concatenation*) on traces.

We define several operations on trace structures, including *parallel composition, projection* and *renaming*. Consider the operation of parallel composition. For all of the different models we consider, this operation on trace structures has exactly the same definition, which is given in terms of the projection operation on traces. The operations of projection and renaming on trace structures are also defined the same way for all of our models. These operations on trace structures form a *trace structure algebra*. Thus, to construct a new trace structure algebra (which provides a domain of agent models), we need only define a new trace algebra (which is a domain of models for individual behaviors). Many of the basic properties of the operations on trace structures follow from the axioms of trace algebra, so they hold for any trace structure algebra.

Trace structures represent both implementations and specifications. An implementation (represented by a trace structure T) satisfies a specification (represented by T') if and only if the set of possible traces of T is contained in the set of possible traces of T'. Intuitively, the specification gives a set of legal behaviors; if all of the behaviors of the implementation are legal, then the implementation satisfies the specification. This particular criteria for satisfying a specification is called *trace set containment*. Since traces can be strings in a formal language, trace set containment is a generalization of the standard notion of *language containment*.

The verification methods we propose involve using two different models. For example, we might use a continuous time model and a discrete time model. As noted above, to construct these models (and the corresponding trace structure algebras) it is only necessary to construct two trace algebras. The continuous time model is the more physically accurate model; if a design satisfies its specification in continuous time, then we can be confident that the design will work properly when implemented. Thus, a continuous time model is used when providing a specification and an implementation to be verified. The specification is given as a continuous time trace structure and the implementation is given as the parallel composition of one or more continuous time trace structures (perhaps with some internal signals hidden). Each of these

continuous time trace structures is abstracted to form a discrete time trace structure. The resulting discrete time specification and implementation are input to an automatic verifier that is based on a discrete time model. The output of the verifier (*i.e.*, whether the implementation satisfies the specification in discrete time) indicates whether the implementation satisfies the specification in continuous time.

There are four cases to consider depending on whether or not the implementation satisfies its specification in discrete time or in continuous time. If the implementation is correct in both cases, or is not correct in both cases, then the discrete time verification accurately indicates whether the implementation is correct in continuous time. A *false positive* is the case where the implementation is correct in discrete time but not in continuous time; the automatic verifier inaccurately indicates that the implementation is correct. The method used to abstract continuous time trace structures into discrete time trace structures must insure that false positives never occur; this is the primary constraint to consider when abstracting continuous time trace structures. A *false negative* is the case where the implementation is correct in continuous time but not in discrete time; the automatic verifier inaccurately indicates that the implementation is incorrect. False negatives are undesirable, but not nearly as dangerous as false positives. The possibility of a false negative is the price one must pay for using a powerful abstraction technique.

It is not possible, in general, to use a discrete time trace structure to exactly represent the set of behaviors modeled by a continuous time trace structure; behaviors must be either added or removed, or both. If behaviors are added when abstracting a specification, then a false positive might result. To see this, consider the case where one of the added behaviors is a possible behavior of the implementation; then the implementation satisfies the specification in discrete time but not in continuous time. Thus, we want the discrete time abstraction of a continuous time specification to be a lower bound (under the set containment ordering) of the set of behaviors of the specification. False positives are avoided regardless of the tightness or looseness of the lower bound; however, a looser bound makes false negatives more likely.

The situation is different when abstracting components of an implementation. Here a false positive might result if behaviors are removed when abstracting. To see this, consider the case where one of the removed behaviors is not a possible behavior of the specification; then the implementation satisfies the specification in discrete time but not in continuous time. Thus, we want the abstraction of a component of an implementation to be a upper bound of the set of behaviors of the component. Again, a looser bound makes false negatives more likely.

We formalize these ideas with *conservative approximations*. When abstracting continuous

time with discrete time, an appropriate conservative approximation Ψ consists of a pair of mappings from continuous time trace structures to discrete time trace structures: a lower bound mapping Ψ_l, and an upper bound mapping Ψ_u. Suppose the implementation satisfies its specification when verified using the discrete time trace structures that result from applying Ψ_l to the specification and Ψ_u to the components of the implementation. By the definition of a conservative approximation, the implementation also satisfies its specification in continuous time. This insures that no false positives are possible.

A conservative approximation between two trace structure algebras can often be induced by certain relationships between the underlying trace algebras. For example, if there is a homomorphism between two trace algebras (in the universal algebra sense of homomorphism), then this induces a conservative approximation between trace structure algebras constructed from the trace algebras. Also, if a trace in one trace algebra is a set of traces from another trace algebra, then this induces a conservative approximation from trace structures over the first trace algebra to trace structures over the second. Conservative approximations from models with explicit simultaneity to models with interleaving semantics can be constructed in this manner; a trace with explicit simultaneity is represented by its set of *interleavings*, which is a set of interleaved traces.

The theoretical work described above was motivated by more practical issues concerning the verification of speed-dependent asynchronous circuits. We have developed a verifier for verifying such circuits that uses a discrete time model; it is a significant extension of the trace theory verifier developed by Dill [38, 39]. In chapter 5, we describe how to use the verifier to analyze two circuits. We also study the effects of using several different delay models in the verification, including inertial delay and a new model called *chaos delay*. We show that using inertial delay can lead to false positive verification results, and that chaos delay can avoid this problem without being overly conservative.

1.1 Major Results

The major results of this thesis are listed below.

- *Trace algebra* and *trace structure algebra,* which are powerful tools for constructing domains of agents models.
- Formalizing the concept of a conservative approximation from one trace structure algebra to another, and proving general theorems for constructing conservative approximations

based on relationships between trace algebras.

- Particular conservative approximations from continuous time models to discrete time models and from explicit simultaneity semantics to interleaving semantics.
- Formalizing the concept of the inverse of a conservative approximation, and characterizing the inverse of a broad class of conservative approximations.
- An automatic verifier that, using conservative approximations, combines the efficiency of discrete time models and the accuracy of continuous time models.
- Using the verifier on speed-dependent asynchronous circuits with several new delay models.

1.2 Motivating Example

In this section we give a concrete example of how using a discrete time model can lead to a false positive verification result. We do this by informally analyzing a circuit due to Brzozowski and Seger [13, 14]. A more formal analysis will be given in chapter 3. For this circuit, gates are modeled according to the *inertial delay* model. To illustrate the inertial delay model, consider a gate with a minimum and maximum delay of one. If the gate becomes firable at time t, and remains firable for one time unit, then it will fire at time $t + 1$. If the gate is only firable for periods of time less than one unit long, then it will not fire.

The example circuit is given in Figure 1.1. The buffers have arbitrary delay (*i.e.*, minimum delay of zero and unbounded maximum delay); the remaining gates have both their minimum and maximum delays equal to one. Initially all wires are low. Assume there is single transition on input w that occurs at time 0. Can this lead to a transition on output z?

First, consider a *synchronous time* model. In chapter 3, we give a taxonomy of real-time models, including synchronous time models (which are a particular kind of discrete time model); for now it is adequate to characterize synchronous time models by assuming that events can only occur at times 0, 1, 2, etc. We can argue that z cannot transition in a synchronous time model. Assume z transitions at time t. This implies that at time $t - 1$ we must have $y1 = 0$, $y2 = 1$, $y3 = 0$. These constraints on $y1$ and $y3$ imply that $x1 = x2$ and $x2 = x3$ at time $t - 2$. But having $x1 = x3$ at time $t - 2$ contradicts the fact that $y2 = 1$ at time $t - 1$. Thus, there can be no z transition.

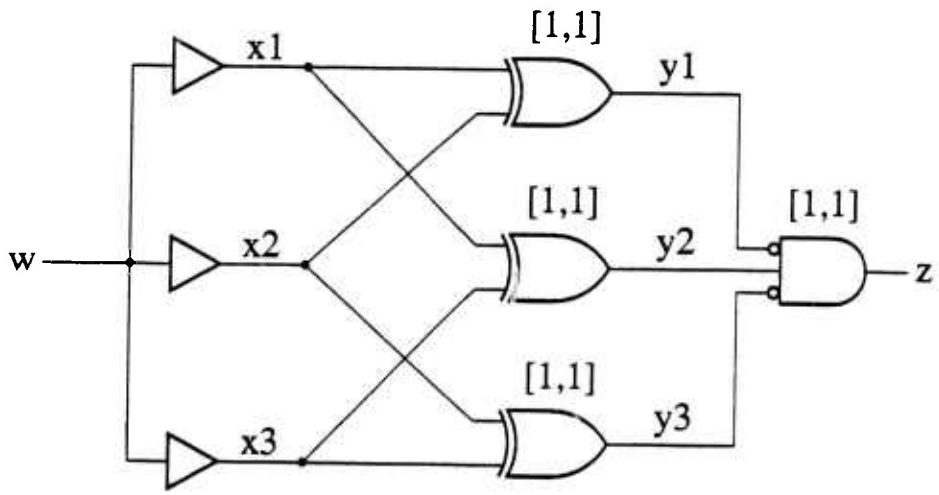


Figure 1.1: Circuit for demonstrating that discrete time models can lead to false positive verification results.

A z transition can occur in the continuous time model, however. Consider the behavior given by

$$\{(w,0), (x3,1.3), (x2,1.9), (y2,2.3), (x1,2.5), (z,3.3)\}.$$

The behavior is represented by a set of events; each event is an ordered pair designating an action and the time at which the action occurred. The order in which events occurred can be derived from the time stamps. Notice that the times between the $x1$ and $x2$ transitions and between the $x2$ and $x3$ transitions are less than one (so $y1$ and $y3$ do not transition), and the time between the $x1$ and $x3$ transitions is greater than one (leading to transitions of $y2$ and z). This is not possible in the synchronous time model we described above. As a result, the circuit can reach a state (where $z = 1$) in the continuous time model that is not reachable in the synchronous time model. This can lead to false positive verification results.

The example does not show that it is impossible to reliably avoid false positives when using a synchronous model for verification; it merely shows that false positives are possible if one is not careful about how gates are modeled in synchronous time. In fact, in chapter 3 we construct a conservative approximation from continuous time models to synchronous time models. When this conservative approximation is used to construct synchronous time models of gates (like the gates used in figure 1.1) from the corresponding continuous time models, then false positives (relative to the continuous time models, see below) are provably impossible. This results from the conservative approximation including extra behaviors in the synchronous time gate models, behaviors that were not included in the informal synchronous time model we used

to (incorrectly) argue that a z transition is not possible in figure 1.1.

Even when conservative approximations are used, there is another source of false positives that must be considered. Recall that using a conservative approximation from continuous time to discrete time (for example) guarantees that if an implementation satisfies its specification in synchronous time, then it also satisfies its specification in continuous time. In this case, because of the conservative approximation, we say that false positives are impossible *relative to the continuous time model.* However, it may still be possible to have a false positive *relative to the physical implementation;* that is, the implementation may satisfy its specification in the continuous time model, but still not work correctly when actually built. This may be caused either by errors in the formal specification or by errors in the continuous time models of the components of the implementation.

The possibility of errors in formal specifications is a very important problem that has received a lot of attention. In this thesis, however, we consider the simpler (but still surprisingly subtle) problem of avoiding errors in models of components. In chapter 5, we show that using inertial delay gates (like the ones we used to analyze figure 1.1) can lead to a false positive relative to a physical implementation. Other gate models, such as *chaos delay* (section 5.5), avoid these false positives, while reducing the chances of a false negative.

1.3 Related Work

Methodologies for formal verification provide formal semantics for agents and specifications, and means for describing agents and specifications in a convenient language and/or with data structures. They also provide ways of determining whether an implementation satisfies a specification, and to make this task easier, they often provide abstraction techniques. Isolating each of these properties of a verification methodology provides a natural way of organizing our description of related work.

1.3.1 Agent semantics

One of the most important distinguishing features of a verification methodology is the semantics used for agents. The most common semantics for untimed agents are state-transition systems (with either labeled states or labeled transitions), sets of sequences of states, and sets of sequences of events (or sets of events).

An early use of the term *trace* in a formal model of concurrency was in Hoare's *trace*

semantics for CSP [48, 49]. Here a possible behavior of an agent (a process, in this case) is represented by a *trace,* which is a finite sequence of communication actions. An agent is then modeled by a prefix-closed set of traces. To better model deadlock and divergence, this model was extended to include *failures* and *divergences* [7, 8, 88]. Reed *et al.* have developed a hierarchy of real-time extensions to these models [84, 86]. A timed trace is a sequence of *timed communications* (t, a), where a is a communication action and t is a real valued time stamp. Failures are also extended with timing information. *Timed stability values* have a role similar to divergences in untimed CSP.

Rem *et al.* have used traces to denote sequences of voltage transitions in asynchronous circuits, rather than sequences of communication actions [87]. Dill extended this model to implement an automatic verifier for speed-independent asynchronous circuits [38, 39]. Circuits were described by two sets of traces, a *success* set and a *failure* set (this notion of failure is not related to failures in CSP semantics), which represent requirements on the environment as well as on the circuit itself. Dill also formalized the notion that a model of a circuit is *receptive* iff it can never block any of its inputs. Although it was never implemented, Dill extended his model to include infinite traces for representing liveness properties [6, 38, 40].

Modeling behaviors with sequences of actions, as above, is known as *interleaving semantics.* The possibility of two actions occurring simultaneously is not explicitly represented in interleaving semantics. Thus, interleaving semantics is potentially less accurate then semantics with explicit simultaneity. Note that there is another notion of *interleaving* that is sometimes used in real-time software analysis: only one process is allowed to be running at a time. This contrasts with *maximal parallelism* models, where it is assumed that each process has its own processor. All of the models we use in this thesis are analogous to maximal parallelism, even though we sometimes use interleaving semantics.

Concurrency can be represented more explicitly by using sequences of sets of actions: a non-singleton set represents two or more events occurring simultaneously. This is a convenient semantics for synchronous systems. It is also a simple discrete time model that can be used for analyzing real-time systems [5]. Untimed asynchronous agents can be modeled by using sequences of non-empty sets of actions. Sequences of states can also be used to provide a similarly expressive model of concurrency. Here untimed systems are modeled using *stutter free* sequences or *stuttering closed* sets of sequences. Other models of concurrency include *Mazurkiewicz traces* [74] and *partially-ordered multisets* [83].

Models based on sequences of actions and sequences of states can be extended to real-time models in many different ways. Alur and Henzinger provide a good survey of these extensions,

as well as other real-time modeling issues [4].

There is a common feature of the models we have described so far in this section: agents are modeled by sets of elements, and each element represents a possible behavior of the agent. Any model with this feature can be handled using our notions of trace algebra and trace structure algebra, as long as the axioms of trace algebra (which are quite weak) are satisfied. The elements that represent behaviors, which we call *traces* (using the term quite broadly), become the carrier of an appropriate trace algebra. A *trace structure*, which represents an agent, contains a set of such traces. These trace structures form the carrier of a trace structure algebra, which has operations of parallel composition, projection and renaming on trace structures.

A long term goal of our research with trace algebra is to encode a large number of the existing models of concurrency as trace algebras and trace structure algebras. We believe that trace algebra can provide a kind of unifying theory, highlighting the important differences and similarities between these models. This thesis takes a first step in this direction by constructing conservative approximations between several real-time models.

Even though trace algebra is quite general, it cannot be used to adequately model branching time properties. In this situation, an agent is typically modeled with some sort of *labeled transition system*. If the states of the transition system are labeled, then it is called a *Kripke structure* [35]; if the edges (transitions) are labeled, it is called a synchronization tree [76]. Determining what features of trace structure algebras and conservative approximations can be extended to branching time semantics is an interesting research question, but it is beyond the scope of this thesis.

1.3.2 Describing Agents

After the semantics of agents are determined, it is still necessary to represent agents in human readable form (with a description language) and/or in machine readable form (with an appropriate data structure). As a simple example, assume an agent is modeled by a set of sequences of some sort. The set of sequences can be viewed as a formal language. If agents are finite state, then data structures based on finite automata or ω-automata can be used to represent agents. The verifier [16, 17] we use in chapter 5 uses automata to represent trace structures that consist of prefix-closed sets of finite sequences.

Input-output automata are a slight extension of conventional automata for representing finite and infinite sequences of actions [69]. They have been further extended to represent

timed behaviors using a continuous time model [68, 94]. Input-output automata are used in verification methods based on *refinement mappings* (see section 1.3.3). Verification based on language containment algorithms can be done with the timed automata of Alur and Dill [2, 41] and Lewis [64]. All of these techniques use the same underlying model of continuous time behaviors as is provided by the trace algebra C_C^{CTU} (see section 3.2); they just provide different ways of expressing agents. The verification methods we propose (see section 5.2) do not require directly representing continuous time agents; instead, we construct discrete time agents that are conservative approximations of the intended continuous time semantics.

A transition system can be used as branching time semantics (as above), or as a representation of linear time semantics if only its set of execution sequences are considered. Real-time extensions of transition systems include both continuous time [46] and discrete time [45, 81, 82].

Process algebras have also been used as the basis for real-time specification languages. Lee and Davidson [62] and Lee and Zwarico [63] have extended CSP with methods for specifying timeouts and delays associated with executing actions. Schneider has shown how extending CSP with a single *wait* operator makes it possible derive a large number of other standard timing operators [89]. Nicollin *et al.* [79, 80] have extended ACP with a unit delay operator. This operator can be used to express delays and timeouts of arbitrary duration. CCS has also been extended with operators for describing real-time processes [95, 96].

1.3.3 Specification and Verification

In its most general form, a specification is a set of agents; verification is the process of determining whether a given implementation is in the set of agents of the specification.

If there is some equivalence relation defined on the set of agent models, and the specification is an equivalence class, then the specification can be represented by one of the agents in the class. Several examples of this style of specification are based on various kinds of *observational equivalences* [76]. Hierarchical verification is simplified since both the implementation and the specification are given by a single agent.

A generalization of this is to use a preorder on agents rather than a equivalence relation. An agent then represents the set of all agents that are less than or equal to it according to this order. The preorder is often based on formal language containment [57]. The idea is that if behaviors are removed from an agent that satisfies some specification, then the resulting agent also satisfies the specification. Verification can either be done by hand (possibly with assistance from an automated theorem prover) using refinement mappings [68] or by language

containment algorithms on automata [32, 38, 57].

Most work in the literature on the automatic verification of real-time systems uses some sort of temporal logic as a specification language. These logics are usually extensions of existing qualitative temporal logics such as CTL [34] or PTL [66], which all suffer from well known limits in the expressiveness of propositional temporal logics [97]. A formula in a temporal logic serves as a specification. The set of agents represented by the specification is the set of agents that satisfies the formula.

An implementation can be represented by a formula in temporal logic (like the specification) or it can be represented by a transition system. If the implementation is represented by a formula f and the specification is represented by a formula g, then the implementation is correct if and only if the formula $f \wedge \neg g$ is not satisfiable; this can be checked using a *tableau* construction [33, 70]. *Model checking* is used to check whether a transition system satisfies a given temporal logic formula [26, 34, 35]. Hierarchical verification is difficult with model checking since the specification language is different from the languages used to describe implementations.

Ostroff [81, 82] extends linear temporal logic to include a global clock variable that can be used in forming propositions. The semantics is defined on a discrete time model, and algorithms are given for automatic model checking of formulas in the logic. The semantics and the algorithms are quite complicated however, and only small verification examples have been published. There are other examples of extending temporal logics with a discrete time model [3, 47, 54], but none of these methods have been implemented and tested on examples.

Methods for model checking a continuous real-time extension of CTL have been developed by Alur, Courcoubetis and Dill [1], and also independently by Lewis [65]. It appears likely that the exact modeling of continuous time reduces the efficiency of the model checking algorithms. Alur and Henzinger [4] give a survey of these and other real-time temporal logics.

Rather than using a temporal logic, Jahanian, Mok and Stuart use RTL (an extension of first order logic) to describe real-time systems and their specifications [51, 52]. If the specification is a theorem derivable from the formula representing the system, then the system is correct. The proof can be automated using either a first order theorem prover or a decision procedure for Presburger Arithmetic. System descriptions can also be written using the *event-action model* and then mechanically translated into RTL formulas.

1.3.4 Abstraction

Abstraction techniques are important for reducing the complexity of verification. We describe here some of the abstraction techniques that are closely related to the conservative approximations from continuous time to discrete time that we define later in the thesis.

Henzinger, Manna and Pnueli explore the relationship between verification results obtained with discrete time and continuous time models [47]. They show that for implementations given by time transition systems, and for specifications written in a large subset of *metric temporal logic*, properties hold in discrete time if and only if they hold in continuous time. This *exactness* result does not give the same amount of flexibility as conservative approximations do for devising abstractions. Also, their results appear to depend rather heavily on the particular behavior model that they used.

Kurshan *et al.* have verified several commercial communication systems and protocols [44] using powerful abstractions techniques based on homomorphisms on automata [57, 58]. The abstractions are closely related to our notion of a *conservative approximation induced by a homomorphism*. Our techniques for constructing domains of agent models and conservative approximations are significantly more general, but Kurshan *et al.* have gained considerable practical experience with their techniques.

Kurshan and McMillan [59] generalized homomorphisms on automata to develop a semi-algorithmic method for extracting finite-state models from an analog, circuit level model. This requires modeling continuous time, as well as continuous voltage and other physical parameters. The method can be applied directly to only small circuit components. However, hierarchical verification methods can be applied in order to verify larger circuits. Although the method can relate particular continuous and discrete models, it does not provide a relationship between entire domains of agent models like conservative approximations.

Reed, Roscoe and Schneider have defined an extensive hierarchy of timed models for CSP [84, 85, 86]. They show how abstractions within the hierarchy can be used to simplify correctness proofs [86, 89]. However, they do not provide mathematical tools, such as trace algebra and trace structure algebra, for simplifying extensions to the hierarchy. Also, in their models behaviors are either untimed or have real-valued time stamps; there are no intermediate discrete time models. The levels in the hierarchy are formed from various combinations of timed and untimed CSP traces, failures and stability values.

Chapter 2

Trace Algebra, Part I

This chapter describes some very general methods for constructing different models of concurrent systems, and for proving relationships between these models. The most important of these relationships is the concept of a *conservative approximation.* Informally, a model is a conservative approximation of a second model when the following condition is satisfied: if an implementation satisfies a specification in the first model, then the implementation also satisfies the specification in the second model. Conservative approximations are useful when the second model is accurate but difficult to use in proofs or with automatic verification tools, and the first model is an abstraction that simplifies verification.

The formal methods we describe are based on three kinds of inter-related algebras: concurrency algebra, trace algebra and trace structure algebra. Concurrency algebra is based on Dill's *circuit algebra* [38] and is a simple abstract algebra with three operations: parallel composition, projection, and renaming. The three operations must satisfy the axioms C1 through C9 (p. 24). The domain (or carrier) of a concurrency algebra is intended to represent a set of processes, or *agents*. Any set can be the domain of a concurrency algebra if interpretations for parallel composition, projection and renaming that satisfy C1 through C9 can be defined over the set. In this thesis, whenever we define an interpretation for these three operations, we always show that the interpretation forms a concurrency algebra, which gives evidence that the interpretation makes intuitive sense.

We often use a set of *trace structures* as the domain of a concurrency algebra. This special case of a concurrency algebra is a called a *trace structure algebra.* Each trace structure contains a set of *traces,* where each trace represents a behavior of the agent modeled by the trace structure. The kind of trace that is used is a parameter in our method. Any mathematical object that satisfies certain minimum requirements can be used as a trace. These requirements

are formalized as the axioms of *trace algebra*. A trace algebra has a set of traces as its domain, and defines the operations of projection and renaming (and possibly concatenation) on traces.

In summary, a *trace algebra* has a set of traces as its domain, and each trace is interpreted as an abstraction of a physical behavior. A sequence of actions is a standard example of a trace, but in trace algebra any mathematical object can used as a trace as long as certain axioms are satisfied. An agent is modeled by a *trace structure*, which contains a set of traces from some trace algebra, representing the set of possible behaviors of the agent. The operations of parallel composition, projection and renaming are defined over a domain of trace structures, forming a *trace structure algebra*. These operations satisfy the axioms of *concurrency algebra*, so a trace structure algebra is a special case of a concurrency algebra.

2.1 Concurrency Algebra

Concurrency algebras (which are based on Dill's *circuit algebra* [38]) have the following operations on agents: parallel composition, projection and renaming. These operations satisfy a set of axioms, which are intended to be consistent with the intuitive meaning of the operations.

Agents communicate through either shared actions or shared state variables. We use the term *signal* to refer to either an action or a state variable. We associate with each agent an *agent signature* (or just *signature*), which describes sets of input signals and output signals.

Definition 2.1. We use W to denote a set of *signals*. The set of *agent signatures* Γ *over* W is the set of ordered pairs (I, O) such that I and O are disjoint subsets of W. We use γ to denote agent signatures (often called just *signatures*).

In a signature (I, O) over W, the set W is usually infinite and the sets I and O are usually finite, but this is not required.

Definition 2.2. If $\gamma = (I, O)$ is a signature over W, then $A = I \cup O$ is the *alphabet of* γ. If A is the alphabet of some signature, then we call A an *alphabet*. Thus, an *alphabet over* W is any subset of W.

Note 2.3. When we mention a signature γ, we also implicitly define I and O so that $\gamma = (I, O)$. We also implicitly define A to be the alphabet of γ. If the name of the signature is decorated with primes and/or subscripts, those decorations carry over to the implicitly defined quantities. For example, mentioning a signature γ_1' implicitly defines I_1', O_1' and A_1'.

Note 2.4. If an object E has an agent signature associated with it, we implicitly define γ to be that signature. If the name of the object is decorated with primes and/or subscripts, those decorations carry over to the implicitly defined signature. For example, associating a signature with an object E_1' implicitly defines a signature γ_1'. This, as described in note 2.3, also implicitly defines I_1', O_1' and A_1'.

The renaming operation uses a *renaming function,* which is a bijection from one alphabet to another.

Definition 2.5. A function r with $dom(r) = A$ and $codom(r) = B$, where A and B are alphabets over W, is a *renaming function over* W if r is a bijection.

The parallel composition of two agents E and E' (written $E \parallel E'$) corresponds to, for example, joining two circuits or running two processes concurrently. In the resulting composition, E and E' communicate through shared signals. We require that no signal be an output of both E and E'. The agent $rename(r)(E)$ is formed from E by renaming the signals of E according to r. If B is a subset of the alphabet of E, then $proj(B)(E)$ has B as its alphabet; the remaining signals of E are not externally visible. We allow only outputs of E to be hidden, so B must contain all of the inputs of E. The three operations of concurrency algebra satisfy several identities. All of this is formalized in the following definition.

Definition 2.6. A *concurrency algebra over* W has a domain $\mathcal{D}$ of *agents,* and the operations of *parallel composition, projection* and *renaming,* denoted by $\parallel$, $proj(B)$ and $rename(r)$. Associated with each element of $\mathcal{D}$ is an agent signature from the set Γ of agent signatures over W. Let E and E' be elements of $\mathcal{D}$ (recall that this implicitly defines I, I', etc., see note 2.4). The signatures of $E \parallel E'$, $proj(B)(E)$ and $rename(r)(E)$ are given by the following rules.

- If $O \cap O' = \emptyset$, then $E \parallel E'$ is defined and its signature is

$$((I \cup I') - (O \cup O'),\ O \cup O').$$

- If $I \subseteq B \subseteq A$, then $proj(B)(E)$ is defined and its signature is $(I, O \cap B)$.
- If r is a renaming function with domain A, then $rename(r)(E)$ is defined and its signature is $(r(I), r(O))$, where r is naturally extended to sets.

The operations must satisfy the identities given below. In all of the identities, there is an implicit assumption that the left hand side of the equation is defined; in each case, if the left hand side is defined, then so is the right hand side.

C1. $(E \parallel E') \parallel E'' = E \parallel (E' \parallel E'')$.

C2. $E \parallel E' = E' \parallel E$.

C3. $rename(r)(rename(r')(E)) = rename(r \circ r')(E)$.

C4. $rename(r)(E \parallel E') = rename(r|_{A \to r(A)})(E) \parallel rename(r|_{A' \to r(A')})(E')$.

C5. $rename(id_A)(E) = E$.

C6. $proj(B)(proj(B')(E)) = proj(B)(E)$.

C7. $proj(A)(E) = E$.

C8. $proj(B)(E \parallel E') = proj(B \cap A)(E) \parallel proj(B \cap A')(E')$, if $(A \cap A') \subseteq B$.

C9. $proj(r(B))(rename(r)(E)) = rename(r|_{B \to r(B)})(proj(B)(E))$.

2.2 Trace Algebra

Several methods for verifying concurrent systems are based on checking for *language containment* or related properties [38, 43, 49, 57, 68]. In the simplest form of language containment-based verification, each agent is modeled by a formal language of finite (or possibly infinite) sequences. If agent T is a specification and T' is an implementation, then T' is said to satisfy T if the language of T' is a subset the language of T. The idea is that each sequence, sometimes called a *trace*, represents a behavior; an implementation satisfies a specification iff all the possible behaviors of the implementation are also possible behaviors of the specification.

The method we use in this thesis for verifying real-time properties is a generalization of the language containment method. Traces are not restricted to be sequences, but can be any mathematical object that has certain properties. In this section, these properties are formalized in the axioms of *trace algebra*, which is a kind of abstract algebra that has a set of traces as its domain. The next section describes *trace structure algebra*, which has as its domain a set of

trace structures, each containing a subset of the traces from a given trace algebra. The notion of one trace structure satisfying another is based on trace set containment.

Before giving the formal definitions of these concepts, let us describe a simple example of a trace algebra and a trace structure algebra. Let the set of traces over an alphabet A be A^∞, which is the set of finite and infinite sequences over A. A pair (γ, P) is a trace structure if γ is a signature and $P \subseteq A^\infty$, where A is the alphabet of γ.

We define the operations of parallel composition, projection and renaming on trace structures by first defining projection and renaming on individual traces. If $x \in A^\infty$ and $B \subseteq A$, then $proj(B)(x)$ is the string formed from x by removing all symbols not in B. If r is a renaming function over A, then $rename(r)(x)$ is the string formed from x by replacing every symbol a with $r(a)$.

Projection and renaming on trace structures are just the natural extensions of the corresponding operations on traces. In particular, if $T = ((I, O), P)$ is a trace structure, $I \subseteq B \subseteq A$ and r is a renaming function over A, then

$$\begin{aligned} proj(B)(T) &= ((I, O \cap B), proj(B)(P)), \\ rename(r)(T) &= ((r(I), r(O)), rename(r)(P)), \end{aligned}$$

where the operations of projection and renaming on traces are naturally extended to sets of traces. If $T = (\gamma, P)$ is equal to the parallel composition of T' and T'', then

$$P = \{x \in A^\infty : proj(A')(x) \in P' \wedge proj(A'')(x) \in P''\}.$$

Given our definition of projection on strings, this is a natural definition of parallel composition. Rem, van de Snepsheut and Udding's [87] definition of the set of traces resulting from parallel composition is almost identical to ours, except that it is restricted to finite length strings.

Looking at the above definitions more closely, we can see how these concepts can be generalized to unify many different kinds of models. Rather than always using strings in a formal language as the domain of traces, we can use any domain that has projection and renaming operations defined on it and that satisfies certain requirements. These requirements are formalized in the axioms of trace algebra. In each case, the operations on trace structures are defined exactly as above, in terms of the operations on individual traces. The resulting trace structure algebra satisfies the axioms of concurrency algebra because the underlying traces satisfy the axioms of trace algebra. The remainder of this chapter formalizes and proves these claims, and defines what it means for one trace structure algebra to be a conservative approximation of another.

We make a distinction between two different kinds of behaviors: *complete* behaviors and *partial* behaviors. A complete behavior has no endpoint. Since a complete behavior goes on forever, it does not make sense to talk about something happening "after" a complete behavior. A partial behavior has an endpoint; it can be a prefix of a complete behavior or of another partial behavior. Every complete behavior has partial behaviors that are prefixes of it; every partial behavior is a prefix of some complete behavior. The distinction between a complete behavior and a partial behavior has only to do with the length of the behavior (that is, whether or not it has an endpoint), not with what is happening during the behavior; whether an agent does anything, or what it does, is irrelevant.

Complete traces and *partial traces* are used to model complete and partial behaviors, respectively. A given object can be both a complete trace and a partial trace; what is being represented in a given case is determined from context. For example, a finite string can represent a complete behavior with a finite number of actions, or it can represent a partial behavior. The form of trace algebra we define here has only complete traces; it is intended to represent only complete behaviors. Trace algebra with partial traces will be defined in chapter 4. We use the symbol '$\mathcal{C}$' to denote trace algebras. Since we only consider here trace algebras with complete traces and without partial traces, we use a subscript 'C' (*e.g.*, '$\mathcal{C}_C$') to denote the trace algebras used in this chapter.

Definition 2.7. A *trace algebra* $\mathcal{C}_C$ *over* W is a triple $(\mathcal{B}_C, proj, rename)$. For every alphabet A over W, $\mathcal{B}_C(A)$ is a non-empty set, called the set of traces over A. Slightly abusing notation, we also write $\mathcal{B}_C$ as an abbreviation for

$$\bigcup \{\mathcal{B}_C(A) : A \text{ is an alphabet over } W\}.$$

For every alphabet B over W and every renaming function r over W, $proj(B)$ and $rename(r)$ are partial functions from $\mathcal{B}_C$ to $\mathcal{B}_C$. The following axioms T1 through T8 must also be satisfied. For all axioms that are equations, we assume that the left side of the equation is defined.

T1. $proj(B)(x)$ is defined iff there exists an alphabet A such that $x \in \mathcal{B}_C(A)$ and $B \subseteq A$. When defined, $proj(B)(x)$ is an element of $\mathcal{B}_C(B)$.

T2. $proj(B)(proj(B')(x)) = proj(B)(x)$.

T3. If $x \in \mathcal{B}_C(A)$, then $proj(A)(x) = x$.

T4. Let $x \in \mathcal{B}_C(A)$ and $x' \in \mathcal{B}_C(A')$ be such that $proj(A \cap A')(x) = proj(A \cap A')(x')$. For all A'' where $A \cup A' \subseteq A''$, there exists $x'' \in \mathcal{B}_C(A'')$ such that $x = proj(A)(x'')$ and $x' = proj(A')(x'')$.

T5. $rename(r)(x)$ is defined iff $x \in \mathcal{B}_C(dom(r))$. When defined, $rename(r)(x)$ is an element of $\mathcal{B}_C(codom(r))$.

T6. $rename(r)(rename(r')(x)) = rename(r \circ r')(x)$.

T7. If $x \in \mathcal{B}_C(A)$, then $rename(id_A)(x) = x$.

T8. $proj(r(B))(rename(r)(x)) = rename(r|_{B \to r(B)})(proj(B)(x))$.

T1 and T5 state when the operations on traces are defined. T2, T3, T6, T7 and T8 are natural properties corresponding to C6, C7, C3, C5 and C9, respectively. The remaining axiom, T4 is a kind of "diamond property", as illustrated in figure 2.1. As an example of applying T4, consider the case where traces are sequences. Let $A = \{a, b\}$, $A' = \{b, c\}$, $x = abab$ and $x' = bcb$. Clearly $proj(A \cap A')(x)$ and $proj(A \cap A')(x')$ are both equal to bb. Choosing $x'' = abacb$ demonstrates the T4 holds for this pair of sequences. Intuitively, T4 requires that if two traces x and x' are compatible on their shared signals (*i.e.*, $A \cap A'$), then there exists a trace x'' that corresponds to the synchronous composition of x and x'.

Note 2.8. We naturally extend the renaming and projection operations on traces to operations on sets of traces. For example, if $rename(r)(x)$ is defined for every x in X, then $rename(r)(X)$ is defined such that

$$rename(r)(X) = \{rename(r)(x) : x \in X\}.$$

2.2.1 Examples

As an example trace algebra, we formalize the trace algebra briefly described at the beginning of section 2.2, which we call $\mathcal{C}_C^I$. We always use the symbol '$\mathcal{C}$' to denote trace algebras, and the superscript 'I' is a mnemonic for an (untimed) interleaving model; the subscript 'C' indicates that there are only complete traces in the trace algebra (*i.e.*, a trace algebra without partial traces).

Definition 2.9. For a given set of signals W, the trace algebra $\mathcal{C}_C^I = (\mathcal{B}_C^I, proj^I, rename^I)$ over W is defined as follows:

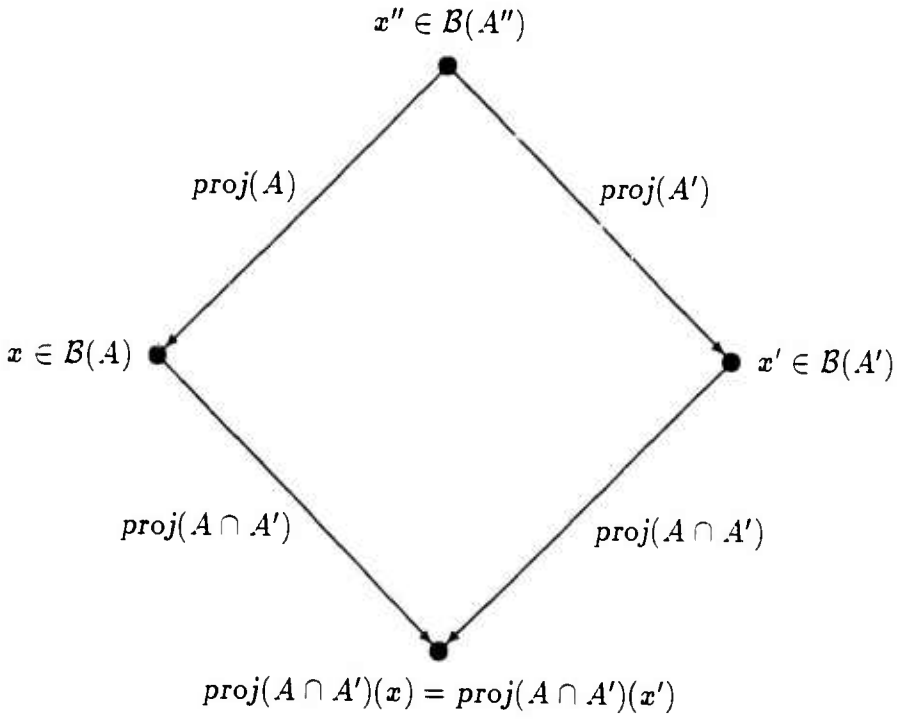


Figure 2.1: According to T4, if there exists an x and an x' that satisfy the lower half of the diamond, then there exists an x'' that satisfies the upper half, for any alphabet A'' such that $A \cup A' \subseteq A''$.

- For every alphabet A over W, the set $\mathcal{B}_C^I(A)$ of traces over A is A^∞.
- If $x \in \mathcal{B}_C^I(A)$ and $B \subseteq A$, then $proj^I(B)(x)$ is the sequence formed from x by removing every symbol a not in B. More formally, if $x' = proj^I(B)(x)$, then

  $$len(x') = |\{j \in \mathcal{N} : 0 \leq j < len(x) \wedge x(j) \in B\}|$$

  and $x'(k) = x(n)$ for all $k < len(x')$, where n is the unique integer such that $x(n) \in B$ and

  $$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge x(j) \in B\}|.$$

- If $x \in \mathcal{B}_C^I(A)$ and r is a renaming function over W with domain A, then $rename(r)(x) = \lambda n \in \mathcal{N}^+[r(x(n))]$.

Note 2.10. For the trace algebra $\mathcal{C}_C^I$ (and analogously for other trace algebras defined later) we often drop the superscript 'I' when writing $\mathcal{B}_C^I$, $proj^I$ and $rename^I$.

Trace algebra can be used to construct a large variety of behavior models. The trace algebra $\mathcal{C}_C^I$, for which $\mathcal{B}_C(A) = A^\infty$, is just one example. To provide more intuition about the range of possible trace algebras, we informally describe several examples.

The simplest possible trace algebra has exactly one trace; call it x_0. For any alphabet A, the set of traces over A is $\mathcal{B}_C(A) = \{x_0\}$. If B is an alphabet and r is a renaming function, then $proj(B)(x_0)$ and $rename(r)(x_0)$ are defined and are equal x_0. This trace algebra does not distinguish between any behaviors; all behaviors are represented by the same trace. For this reason it is not a useful trace algebra, but it does satisfy the necessary axioms.

A slightly more complicated trace algebra has $\mathcal{B}_C(A) = 2^A$. For any trace x, $proj(B)(x)$ is defined and is equal to $x \cap B$. On the other hand, $rename(r)(x)$ is defined iff $x \subseteq dom(r)$; when defined, it is equal to $r(x)$, where is r is naturally extended to sets. It is easy to show that this trace algebra satisfies T1 through T8; in particular, if x and x' satisfy the hypothesis of T4, then $x'' = x \cup x'$ is sufficient to show that T4 is satisfied. Traces in this trace algebra do not provide any information about actions occurring in sequence, only information about what actions occurred a non-zero number of times during a behavior. Alternatively, if $a \in x$, then this could be interpreted to mean that a occurred an odd number of times during the behavior represented by x.

Traces in the last two examples provide less information about a behavior than do traces in $\mathcal{C}_C^I$. As an example of a trace algebra that provides more information than $\mathcal{C}_C^I$, let $\mathcal{B}_C(A) = (2^A)^\omega$. For any trace x, $proj(B)(x)$ is defined and is formed from x by intersecting each element of the sequence with B. The function $rename(r)$ is the natural extension of r to sequences of sets. Unlike traces in $\mathcal{C}_C^I$, these traces can be interpreted as providing information about the time at which events occur. If x is such a trace, then $x(n)$ is the set of events that occurred at time n. The set $x(n)$ must be defined for all integers n; therefore, each trace x must be an infinite sequence. This trace algebra can be shown to be isomorphic to the synchronous time trace algebra $\mathcal{C}_C^{ST}$ (definition 3.6, p. 60).

A trace algebra that provides an intermediate amount of information between the last example and $\mathcal{C}_C^I$ can be constructed by letting $\mathcal{B}_C(A) = (2^A - \{\emptyset\})^\infty$. The renaming operation is the same as the last example, except that it is also extend to finite sequences. Projection is similar to the last example, except that after doing the intersection, any instances of the empty set that result must be removed from the sequence. Like $\mathcal{C}_C^I$, this trace algebra is untimed; however, it represent simultaneity explicitly, unlike interleaving semantics.

In chapter 3, we describe the continuous time trace algebra $\mathcal{C}_C^{CTU}$. There each trace over an alphabet A is an element of $2^{A \times \Re^{\geq}}$, where $\Re^{\geq}$ is the set of non-negative real numbers.

Each trace is a set of events; each event is an ordered pair of an action and a time stamp. An isomorphic trace algebra can be constructed by taking advantage of the natural bijection between $2^{A \times \Re^{\nearrow}}$ and $\Re^{\nearrow} \rightarrow 2^A$. If x is a trace in $\Re^{\nearrow} \rightarrow 2^A$, then $x(t)$ is the set of actions that occurred at time t.

All of the trace algebras we have described are action based, but trace algebra can also be used for state based models. For an agent with alphabet A, we interpret each $a \in A$ as a state variable. Let V be the set of values that can be taken by state variables. Then, each state is an element of $A \rightarrow V$. A trace algebra based on sequences of states would have $\mathcal{B}_C(A)$ equal to $(A \rightarrow V)^\omega$, which can also be written as $\mathcal{N}^{\nearrow} \rightarrow (A \rightarrow V)$.

For a continuous time, state based model, let $\mathcal{B}_C(A) = \Re^{\nearrow} \rightarrow (A \rightarrow V)$. If x is such a trace, then $x(t)$ is the state at time t. If V is the set of real numbers, then this trace algebra could be used as a circuit model that represents both continuous time and continuous voltage.

In section 2.3 we show how trace algebras can be used to construct *trace structure algebras*. We can then discuss how the above trace algebra examples, which provide different models of individual behaviors, lead to different models of agents.

2.2.2 Proofs

This section proves that $\mathcal{C}_C^I$ is trace algebra. It may be skipped on first reading.

Lemma 2.11. $\mathcal{C}_C^I$ is a trace algebra.

Proof. To show that $\mathcal{C}_C^I$ is a trace algebra, we must show that it satisfies T1 through T8.

T1, T3, T5, T6 and T7 are easy to show. All that remains is T2, T4 and T8.

Lemma 2.12. $\mathcal{C}_C^I$ satisfies T2.

Proof. Let $x \in \mathcal{B}_C(A)$ and $B \subseteq B' \subseteq A$. We must show that

$$proj(B)(proj(B')(x)) = proj(B)(x).$$

The proof can be divided into three cases depending on whether $proj(B)(x)$ and $proj(B')(x)$ are finite or infinite length strings (notice that it is impossible for $proj(B')(x)$ to be finite when $proj(B)(x)$ is infinite). We only consider the case where both are infinite, the other cases are analogous. In this case, x is of the form

$$x = y_0\, b_0\, y_1\, b_1 \cdots y_n\, b_n \cdots,$$

where $y_i \in (A - B)^*$ and $b_i \in B$. Thus,

$$proj(B)(x) = b_0\, b_1 \cdots b_n \cdots.$$

For all i, the trace y_i is of the form

$$y_i = z_{i,0}\, b'_{i,0}\, z_{i,1}\, b'_{i,1} \cdots z_{i,n_i-1}\, b'_{i,n_i-1}\, z_{i,n_i},$$

where $z_{i,j} \in (A - B')^*$ and $b_{i,j} \in B' - B$. Let

$$\begin{aligned} y'_i &= proj(B')(y_i) \\ &= b'_{i,0}\, b'_{i,1} \cdots b'_{i,n_i-1}, \end{aligned}$$

which is an element of $(B' - B)^*$. Clearly,

$$\begin{aligned} proj(B)(proj(B')(x)) &= proj(B)(y'_0\, b_0\, y'_1\, b_1 \cdots y'_n\, b_n \cdots) \\ &= b_0\, b_1 \cdots b_n \cdots \\ &= proj(B)(x). \end{aligned}$$

□

Lemma 2.13. $\mathcal{C}_C^I$ satisfies T4.

Proof. We consider the case where $proj(A \cap A')(x)$ and $proj(A \cap A')(x')$ are of infinite length; the finite case is similar. In this case x and x' are of the form

$$\begin{aligned} x &= x_0\, a_0\, x_1\, a_1 \cdots x_n\, a_n \cdots \\ x' &= x'_0\, a'_0\, x'_1\, a'_1 \cdots x'_n\, a'_n \cdots, \end{aligned}$$

where the a_i and a'_i are elements of $A \cap A'$, and $x_i \in (A - A')^*$ and $x'_i \in (A' - A)^*$. If we assume that

$$proj(A \cap A')(x) = proj(A \cap A')(x'),$$

then $a_i = a'_i$ for every i. An example of an x'' that satisfies T4 is

$$x'' = x_0\, x'_0\, a_0\, x_1\, x'_1\, a_1 \cdots x_n\, x'_n\, a_n \cdots,$$

since

$$\begin{aligned} proj(A)(x'') &= x_0\, a_0\, x_1\, a_1 \cdots x_n\, a_n \cdots \\ &= x \end{aligned}$$

and

$$\begin{aligned} proj(A')(x'') &= x'_0\, a_0\, x'_1\, a_1 \cdots x'_n\, a_n \cdots \\ &= x'. \end{aligned}$$

□

Lemma 2.14. $\mathcal{C}_C^I$ satisfies T8.

Proof. We consider the case where $proj(B)(x)$ is of infinite length; the finite case is similar. In this case, x is of the form

$$x = y_0\, b_0\, y_1\, b_1 \cdots y_n\, b_n \cdots,$$

where $y_i \in (A - B)^*$ and $b_i \in B$. Thus,

$$\begin{aligned} rename(r\,|_{B \to r(B)})(proj(B)(x)) &= rename(r\,|_{B \to r(B)})(b_0\, b_1 \cdots b_n \cdots) \\ &= r(b_0)\, r(b_1) \cdots r(b_n) \cdots. \end{aligned}$$

For all i, let

$$y'_i = rename(r)(y_i).$$

Clearly,

$$\begin{aligned} proj(r(B))(rename(r)(x)) &= proj(r(B))(y'_0\, r(b_0)\, y'_1\, r(b_1) \cdots y'_n\, r(b_n) \cdots) \\ &= r(b_0)\, r(b_1) \cdots r(b_n) \cdots \\ &= rename(r\,|_{B \to r(B)})(proj(B)(x)). \end{aligned}$$

□

□

2.3 Trace Structure Algebra

We are now ready to define the concept of a trace structure algebra. Trace structures are constructed from the traces of a trace algebra, and are used to represent agents. Here we consider trace structures that contain one set of traces, which represents the set of *possible* behaviors of an agent.

Definition 2.15. Let $\mathcal{C}_C = (\mathcal{B}_C, proj, rename)$ be a trace algebra over W. The set of *trace structures* over $\mathcal{C}_C$ is the set of ordered pairs (γ, P), where

- γ is a signature over W,
- A is the alphabet of γ, and
- P is a subset of $\mathcal{B}_C(A)$.

We call γ the *signature* and P the set of *possible traces* of a trace structure $T = (\gamma, P)$.

A trace structure (γ, P) represent an agent with signature γ; each trace in P represents a possible complete behavior of the agent.

Note 2.16. When we mention a trace structure T, we implicitly define γ to be its signature and P to be its set of possible traces. If the name of the trace structure is decorated with primes and/or subscripts, those decorations carry over to the implicitly defined quantities. For example, mentioning a trace structure T_1' implicitly defines a signature γ_1' and P_1'. This, as described in note 2.3, also implicitly defines I_1', O_1' and A_1'.

Definition 2.17. If $\mathcal{C}_C = (\mathcal{B}_C, proj, rename)$ is a trace algebra over W and $\mathcal{T}$ is a subset of the trace structures over $\mathcal{C}_C$, then $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ is a *trace structure algebra* iff the domain $\mathcal{T}$ is closed under the following operations on trace structures: parallel composition (def. 2.18), projection (def. 2.19) and renaming (def. 2.20).

We use the subscript C in $\mathcal{A}_C$ to denote a trace structure algebra that is built from a trace algebra $\mathcal{C}_C$ that has only complete traces (no partial traces). In chapter 4, we will define trace structure algebras that are constructed from trace algebras with both complete and partial traces.

To complete the definition of trace structure algebra, we need to define the operations on trace structures mentioned in definition 2.17.

Definition 2.18. If $O \cap O' = \emptyset$, then $T'' = T \parallel T'$ is defined and

$$\begin{aligned} \gamma'' &= ((I \cup I') - (O \cup O'),\ O \cup O') \\ P'' &= \{x \in \mathcal{B}_C(A'') : proj(A)(x) \in P \wedge proj(A')(x) \in P'\}. \end{aligned}$$

Definition 2.19. If $I \subseteq B \subseteq A$, then

$$proj(B)(T) = ((I, O \cap B), proj(B)(P)).$$

Definition 2.20. If r is a renaming function with domain A, then

$$rename(r)(T) = ((r(I), r(O)), rename(r)(P)).$$

It can be shown, using the axioms of trace algebra, that the operations of parallel composition, projection and renaming on trace structures form a concurrency algebra (see theorem 2.22).

We want to use trace structure algebras as the basis for a verification methodology, which requires defining what it means for an implementation to satisfy a specification when both are given by trace structures. Our notion of satisfaction is based on trace set containment: an implementation satisfies a specification iff it is *contained* by the specification.

Definition 2.21. We say $T \subseteq T'$ (read T is contained in T') iff $\gamma = \gamma'$ and $P \subseteq P'$.

The operations of parallel composition, renaming and projection are monotonic with respect to trace structure containment (see theorem 2.26). The monotonicity of parallel composition is important for using trace structure algebras as a basis for hierarchical verification techniques.

2.3.1 Examples

Let us consider how some of the example trace algebras discussed in section 2.2.1 can be used to construct trace structures, and how the different definitions of projection on traces lead to different notions of parallel composition of trace structures.

Consider trace structures over the trace algebra $\mathcal{C}_C^I$. The set of possible traces of a trace structure with alphabet A is a subset of $\mathcal{B}_C(A)$, which in this case is A^∞. Consider the trace structures

$$\begin{aligned} T &= (\{\{a,b\}, \emptyset), \{abab\}) \\ T' &= (\{\{b,c\}, \emptyset), \{bcb\}). \end{aligned}$$

By the definition of parallel composition in a trace structure algebra, the set of possible traces of $T'' = T \parallel T'$ is

$$\begin{aligned} P'' &= \{x \in \mathcal{B}_C(\{a,b,c\}) : proj(\{a,b\})(x) \in P \wedge proj(\{b,c\})(x) \in P'\} \\ &= \{abacb, abcab\}. \end{aligned}$$

This example illustrates how parallel composition results in nondeterminism in this model.

However, parallel composition does not lead to nondeterminism when the underlying trace algebra is the one with $\mathcal{B}_C(A) = (2^A)^\omega$ described in section 2.2.1. Let

$$\begin{aligned} T &= ((\{a,b\}, \emptyset), \{\langle \{a,b\}, \{a\}, \{b\} \rangle\}) \\ T' &= ((\{b,c\}, \emptyset), \{\langle \{b\}, \{c\}, \{b\} \rangle\}) \end{aligned}$$

Here the set of possible traces of $T'' = T \parallel T'$ is the singleton set

$$P'' = \{\langle \{a,b\}, \{a,c\}, \{b\} \rangle\}.$$

The relevant difference between this model and the interleaving model is that here each trace provides more information about the time of occurrence of events. As a result, the order of events is fully determined when "merging" together two local traces to form a global trace of a composition. Global traces are also fully determined in the cases where traces over an alphabet A are elements of $2^{A \times \Re^{\neq}}$, $(A \rightarrowtail V)^\omega$ or $\Re^{\neq} \rightarrowtail (A \rightarrowtail V)$.

Another case where parallel composition does lead to nondeterminism is the one described in section 2.2.1 where $\mathcal{B}_C(A) = (2^A - \{\emptyset\})^\infty$. In this case, for T and T' defined as above, the set of possible traces of $T'' = T \parallel T'$ is

$$\begin{aligned} P'' = \{&\langle \{a,b\}, \{a\}, \{c\}, \{b\} \rangle, \\ &\langle \{a,b\}, \{a,c\}, \{b\} \rangle, \\ &\langle \{a,b\}, \{c\}, \{a\}, \{b\} \rangle\}. \end{aligned}$$

2.3.2 Proofs

This section proves that trace structure algebras are concurrency algebras and that the operations on trace structures are monotonic with respect to trace structure containment.

Theorem 2.22. Trace structure algebras are concurrency algebras.

Proof. By definition, the domain $\mathcal{T}$ of trace structures is closed under projection, composition and renaming. We must show that C1 through C9 are also satisfied.

Lemma 2.23. Trace structure algebras satisfy C1.

Proof. Let $T_1 = (T \parallel T') \parallel T''$ and $T_2 = T \parallel (T' \parallel T'')$. Using T2 and definition 2.18, it is easy to show that both P_1 and P_2 are equal to

$$\{x \in \mathcal{B}_C(A_1) : proj(A)(x) \in S \wedge proj(A')(x) \in S' \wedge proj(A'')(x) \in S''\}.$$

□

C2 is obvious from the definition of parallel composition. C3 follows easily from T6 and the definition of *rename* on sets of traces and on trace structures.

Lemma 2.24. Trace structure algebras satisfy C4.

Proof. Let $T'' = rename(r)(T \parallel T')$. Then

$$\begin{aligned}
P'' &= rename(r)(\{x \in \mathcal{B}_C(A \cup A') : proj(A)(x) \in P \wedge proj(A')(x) \in P'\}) \\
&= \{rename(r)(x) \in \mathcal{B}_C(r(A \cup A')) : \\
&\qquad proj(A)(x) \in P \wedge proj(A')(x) \in P'\}) \\
&\quad \text{by T6 and T7} \\
&= \{rename(r)(x) \in \mathcal{B}_C(r(A \cup A')) : \\
&\qquad rename(r\mid_{A \to r(A)})(proj(A)(x)) \in rename(r\mid_{A \to r(A)})(P) \\
&\qquad\quad \wedge rename(r\mid_{A' \to r(A')})(proj(A')(x)) \in rename(r\mid_{A' \to r(A')})(P')\} \\
&\quad \text{by T8} \\
&= \{rename(r)(x) \in \mathcal{B}_C(r(A \cup A')) : \\
&\qquad proj(r(A))(rename(r)(x)) \in rename(r\mid_{A \to r(A)})(P) \\
&\qquad\quad \wedge proj(r(A'))(rename(r)(x)) \in rename(r\mid_{A' \to r(A')})(P')\} \\
&\quad \text{by T6 and T7} \\
&= \{y \in \mathcal{B}_C(r(A \cup A')) : proj(r\mid_{A \to r(A)}(A))(y) \in rename(r\mid_{A \to r(A)})(P) \\
&\qquad\quad \wedge proj(r\mid_{A' \to r(A')}(A'))(y) \in rename(r\mid_{A' \to r(A')})(P')\}.
\end{aligned}$$

Thus, P'' is equal to the set of possible traces of

$$rename(r\mid_{A \to r(A)})(T) \parallel rename(r\mid_{A' \to r(A')})(T').$$

□

C5 follows from T7, C6 follows from T2, and C7 follows from T3.

Lemma 2.25. Trace structure algebras satisfy C8.

Proof. Let $T_1 = proj(B)(T \parallel T')$ and $T_2 = proj(B \cap A)(T) \parallel proj(B \cap A')(T')$, where $A \cap A' \subseteq B \subseteq A \cup A'$. It is easy to check that T_1 and T_2 have the same signature; we must show that $P_1 = P_2$. Let $y \in \mathcal{B}_C(B)$, and assume

$$proj(B \cap A)(y) \in proj(B \cap A)(P).$$

Then,

$$\begin{aligned}
&proj(B \cap A)(y) \in proj(B \cap A)(P) \\
&\quad \Leftrightarrow \exists z \in P[proj(B \cap A)(y) = proj(B \cap A)(z)] \\
&\quad \text{by T4} \\
&\quad \Leftrightarrow \exists z \in P[\exists x \in \mathcal{B}_C(B \cup A)[y = proj(B)(x) \wedge z = proj(A)(x) \\
&\qquad \wedge proj(B \cap A)(y) = proj(B \cap A)(z)]] \\
&\quad \text{by substitution for } y \text{ and } z \\
&\quad \Leftrightarrow \exists z \in P[\exists x \in \mathcal{B}_C(B \cup A)[y = proj(B)(x) \wedge z = proj(A)(x) \\
&\qquad \wedge proj(B \cap A)(proj(P)(x)) = proj(B \cap A)(proj(A)(x))]] \\
&\quad \text{by T2} \\
&\quad \Leftrightarrow \exists z \in P[\exists x \in \mathcal{B}_C(B \cup A)[y = proj(B)(x) \wedge z = proj(A)(x)]] \\
&\quad \Leftrightarrow \exists x \in \mathcal{B}_C(B \cup A)[y = proj(B)(x) \wedge proj(A)(x) \in P].
\end{aligned}$$

Similarly,

$$\begin{aligned}
&proj(B \cap A')(y) \in proj(B \cap A')(P') \\
&\quad \Leftrightarrow \exists x' \in \mathcal{B}_C(B \cup A')[y = proj(B)(x') \wedge proj(A')(x') \in P'].
\end{aligned}$$

We use these facts to show

$$\begin{aligned}
P_2 &= \{y \in \mathcal{B}_C(B) : proj(B \cap A)(y) \in proj(B \cap A)(P) \\
&\qquad \wedge proj(B \cap A')(y) \in proj(B \cap A')(P')\} \\
&\quad \text{as shown above} \\
&= \{y \in \mathcal{B}_C(B) : \exists x \in \mathcal{B}_C(B \cup A)[\exists x' \in \mathcal{B}_C(B \cup A')[\\
&\qquad y = proj(B)(x) \wedge proj(A)(x) \in P \\
&\qquad \wedge y = proj(B)(x') \wedge proj(A')(x') \in P']]\} \\
&\quad \text{by T4, since } A \cap A' \subseteq B \subseteq A \cup A' \\
&= \{y \in \mathcal{B}_C(B) : \exists x \in \mathcal{B}_C(B \cup A)[\exists x' \in \mathcal{B}_C(B \cup A')[\exists x'' \in \mathcal{B}_C(A \cup A')[\\
&\qquad y = proj(B)(x) \wedge proj(A)(x) \in P \\
&\qquad \wedge y = proj(B)(x') \wedge proj(A')(x') \in P' \\
&\qquad \wedge x = proj(B \cup A)(x'') \wedge x' = proj(B \cup A')(x'')]]]\} \\
&\quad \text{by T2 and substitution for } x \text{ and } x' \\
&= \{y \in \mathcal{B}_C(B) : \exists x'' \in \mathcal{B}_C(A \cup A')[y = proj(B)(x'') \\
&\qquad \wedge proj(A)(x'') \in P \wedge proj(A')(x'') \in P']\} \\
&= P_1.
\end{aligned}$$

□

C9 follows easily from T8.

□

Theorem 2.26. Parallel composition, *rename* and *proj* are monotonic with respect to trace structure containment.

Proof. Let T and T' be arbitrary trace structures such that $T \subseteq T'$. The theorem follows from following propositions, all of which are easily proved:

- $T \parallel T'' \subseteq T' \parallel T''$,
- $proj(B)(T) \subseteq proj(B)(T')$,
- $rename(r)(T) \subseteq rename(r)(T')$.

□

2.3.3 Constructing Trace Structure Algebras

The definition of a trace structure algebra $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ requires that the set of trace structures $\mathcal{T}$ be closed under the operations on trace structures. This section proves three theorems that make it easier to prove closure, and shows how to use these theorems.

The first theorem states that if $\mathcal{T}$ is equal to the set of all trace structures over $\mathcal{C}_C$, then $\mathcal{T}$ is closed under the operations on trace structures, so $\mathcal{A}_C$ is a trace structure algebra. Recall that the alphabet of a trace structure need not be a finite set. The second theorem shows that the set of all trace structures with finite alphabets is closed under the operations on trace structures.

For the third theorem, let $(\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra, where $\mathcal{T}$ is some subset of the set of trace structures over $\mathcal{C}_C$. For every alphabet B, let $\mathcal{L}(B)$ be a class of sets of complete traces over B, that is, $\mathcal{L}(B) \subseteq 2^{\mathcal{B}c(B)}$. Assume that $\mathcal{L}$ is closed under intersection, renaming, projection and "inverse projection" (this is formalized below). Let $\mathcal{T}'$ be the set of trace structures $(\gamma, P) \in \mathcal{T}$ such that P is in $\mathcal{L}(A)$. Then $\mathcal{T}'$ is closed under the operations on trace structures, so $(\mathcal{C}_C, \mathcal{T}')$ is a trace structure algebra.

Let $\mathcal{T}^I$ be the set of all trace structures over $\mathcal{C}_C^I$. By the first theorem, $\mathcal{A}_C^I = (\mathcal{C}_C^I, \mathcal{T}^I)$ is a trace structure algebra. Let $\mathcal{T}^{IR}$ be the set of all trace structures (γ, P) over $\mathcal{C}_C^I$ for which γ has a finite alphabet and P is a mixed regular set of sequences (that is, P is the union of a regular set and an ω-regular set). By the second and third theorems, $\mathcal{A}_C^{IR} = (\mathcal{C}_C^I, \mathcal{T}^{IR})$ is also a trace structure algebra.

The remainder of this section formalizes and proves these results.

Theorem 2.27. If $\mathcal{C}_C$ is a trace algebra and $\mathcal{T}$ is the set of all of the trace structures over $\mathcal{C}_C$, then $\mathcal{T}$ is closed under the operations on trace structures, so $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ is a trace structure algebra.

Proof. The result of any operation on trace structures is always some trace structure T. Since $\mathcal{T}$ is the set of all trace structures, $T \in \mathcal{T}$. Therefore, by the definition of a trace structure algebra, $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ is a trace structure algebra.

□

Theorem 2.28. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra. Let $\mathcal{T}'$ be the set of trace structures $T \in \mathcal{T}$ such that the alphabet of T is a finite set. Then $\mathcal{A}'_C = (\mathcal{C}_C, \mathcal{T}')$ is a trace structure algebra.

Proof. It is easy to verify that the operations on trace structures produce trace structures with finite alphabets if the arguments to the operations have finite alphabets. This is sufficient show that $\mathcal{A}'_C$ is closed under the operations on trace structures.

□

Definition 2.29. Let $\mathcal{T}$ be a set of trace structure over some trace algebra $\mathcal{C}_C$. The set of *alphabets of* $\mathcal{T}$ is the set of alphabets A of a signature γ in the set

$$\{\gamma : \exists P[(\gamma, P) \in \mathcal{T}]\}.$$

Theorem 2.30. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra. For every alphabet B of $\mathcal{T}$, let $\mathcal{L}(B)$ be a subset of $2^{\mathcal{B}_C(B)}$. Let $\mathcal{T}'$ be the set of trace structures $T \in \mathcal{T}$ such that P is in $\mathcal{L}(A)$. Then $\mathcal{A}'_C = (\mathcal{C}_C, \mathcal{T}')$ is a trace structure algebra if the following requirements are satisfied for every alphabet B of $\mathcal{T}$.

L1. $\mathcal{L}(B)$ is closed under intersection.

L2. If $B' \subseteq B$ and $X \in \mathcal{L}(B)$, then $proj(B')(X) \in \mathcal{L}(B')$.

L3. If $B \subseteq B'$ and $X \in \mathcal{L}(B)$, then

$$\{x \in \mathcal{B}_C(B') : proj(B)(x) \in X\} \in \mathcal{L}(B').$$

L4. If r is a renaming function with domain B and $X \in \mathcal{L}(B)$, then $rename(r)(X) \in \mathcal{L}(r(B))$.

Proof. We must show that $\mathcal{A}'_C$ is closed under the operations on trace structures. To show that $\mathcal{T}'$ is closed under composition, let $T, T' \in \mathcal{T}'$ and let $T'' = T \parallel T'$. Then, P'' is in $\mathcal{L}(A'')$, since $\mathcal{L}(A)$ is closed under intersection (L1) and "inverse projection" (L3). Closure under projection and renaming follows easily from L2 and L4, respectively.

□

Definition 2.31. We define $\mathcal{A}^I_C$ to be the ordered pair $(\mathcal{C}^I_C, \mathcal{T}^I)$, where $\mathcal{T}^I$ is the set of all trace structures over $\mathcal{C}^I_C$. By theorem 2.27, $\mathcal{A}^I_C$ is a trace structure algebra.

Definition 2.32. We define $\mathcal{T}^{IR}$ to be the set of all trace structures $T = (\gamma, P)$ over $\mathcal{C}_C^I$ for which γ has a finite alphabet and P is a mixed regular set of sequences. Also, $\mathcal{A}_C^{IR}$ is the ordered pair $(\mathcal{C}_C^I, \mathcal{T}^{IR})$. By theorem 2.33 (below), $\mathcal{A}_C^{IR}$ is a trace structure algebra.

Theorem 2.33. $\mathcal{A}_C^{IR}$ is a trace structure algebra.

Proof. Let $\mathcal{T}'$ be the set of $T \in \mathcal{T}^I$ with a finite alphabet. By theorem 2.28, since $\mathcal{A}_C^I = (\mathcal{C}_C^I, \mathcal{T}^I)$ is a trace structure algebra, so is $(\mathcal{C}_C^I, \mathcal{T}')$. For all finite alphabets B of $\mathcal{A}^I$, let $\mathcal{L}(B)$ be the set of mixed regular languages over B. It is easy to verify that $\mathcal{L}(B)$ satisfies L1 through L4. Let

$$\mathcal{T}'' = \{T \in \mathcal{T}' : P \in \mathcal{L}(A)\}.$$

By theorem 2.30, since $\mathcal{A}_C^I = (\mathcal{C}_C^I, \mathcal{T}')$ is a trace structure algebra, so is $(\mathcal{C}_C^I, \mathcal{T}'')$. Notice that $\mathcal{T}''$ is equal to $\mathcal{T}^{IR}$. Therefore, $\mathcal{A}_C^{IR} = (\mathcal{C}_C^I, \mathcal{T}^{IR})$ is a trace structure algebra.

□

2.4 Conservative Approximations

In the next chapter we show that discrete time trace structures are a conservative approximation of continuous time trace structures. In preparation for that result, we define here what it means for one trace structure algebra to be a conservative approximation of another.

A conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ is an ordered pair $\Psi = (\Psi_l, \Psi_u)$, where Ψ_l and Ψ_u are functions from $\mathcal{T}$ to $\mathcal{T}'$. For a given trace structure T in $\mathcal{A}_C$, the trace structure $\Psi_l(T)$ is a kind of lower bound of T, while $\Psi_u(T)$ is an upper bound (relative to the '$\subseteq$' ordering on trace structures). Here we require that $\Psi_l(T)$ and $\Psi_u(T)$ have the same signature as T; it is also possible to allow conservative approximations that can change the signature of a trace structure, but that is beyond the scope of this thesis.

As an example, consider the verification problem

$$proj(A)(T_1 \parallel T_2) \subseteq T,$$

where T_1, T_2 and T are trace structures in $\mathcal{T}$. This corresponds to checking whether an implementation consisting of two components T_1 and T_2 (along with some internal signals

that are removed by the projection operation) satisfies the specification T. By definition, if Ψ is a conservative approximation, then showing

$$proj(A)(\Psi_u(T_1) \parallel \Psi_u(T_2)) \subseteq \Psi_l(T)$$

is sufficient to show that the original implementation satisfies its specification. Thus, the verification can be done in $\mathcal{A}'_C$, where it is presumably more efficient than in $\mathcal{A}_C$. A conservative approximation guarantees that doing the verification in this way will not lead to a false positive result, although false negatives are possible depending on how the approximation is chosen. The following definition formalizes the notion of a conservative approximation.

Definition 2.34. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{T}$ to $\mathcal{T}'$. We say $\Psi = (\Psi_l, \Psi_u)$ is a *conservative approximation from* $\mathcal{A}_C$ *to* $\mathcal{A}'_C$ iff the following conditions are satisfied.

- For all $T \in \mathcal{T}$, the signature of $\Psi_l(T)$ and $\Psi_u(T)$ is γ.
- Let E be an arbitrary expression potentially involving parallel composition, projection and renaming of trace structures in $\mathcal{T}$. Let E' be formed from E be replacing every instance of each trace structure T with $\Psi_u(T)$. If T_1 is a trace structure in $\mathcal{T}$, and $E' \subseteq \Psi_l(T_1)$, then $E \subseteq T_1$.

Usually a conservative approximation $\Psi = (\Psi_l, \Psi_u)$ has the additional property that $\Psi_l(T) \subseteq \Psi_u(T)$ for all T, but this is not required. Also, having Ψ_l and Ψ_u be monotonic (relative to the containment ordering on trace structures) is common but not required.

The simplest example of a conservative approximation is $\Psi = (\Psi_l, \Psi_u)$ is

$$\Psi_l(T) = (\gamma, \emptyset)$$
$$\Psi_u(T) = (\gamma, \mathcal{B}'_C(A)).$$

This definition of Ψ clearly satisfies the first condition of definition 2.34. To see that it satisfies the second condition, notice that the set of possible traces of E' and $\Psi_l(T_1)$ will be the universal set and the empty set, respectively; thus, it is never true that $E' \subseteq \Psi_l(T_1)$. This particular conservative approximation is not useful, however, because it always leads to a negative verification result; it cannot be used to show that an implementation satisfies a specification. In section 2.4.2, we will show how a conservative approximation can be constructed using a homomorphism from one trace algebra to another. We give a concrete example of such a conservative approximation in section 3.3.1.

The remainder of this section proves theorems that provide sufficient conditions for showing that some Ψ is a conservative approximation. The first theorem can be understood by recalling the example verification problem described above, and by considering the following chain of implications:

$$
\begin{aligned}
& proj(A)(\Psi_u(T_1) \parallel \Psi_u(T_2)) \subseteq \Psi_l(T) \\
& \quad \text{assuming } \Psi_u(T_1 \parallel T_2) \subseteq \Psi_u(T_1) \parallel \Psi_u(T_2) \\
& \Rightarrow \quad proj(A)(\Psi_u(T_1 \parallel T_2)) \subseteq \Psi_l(T) \\
& \quad \text{assuming } \Psi_u(proj(A)(T')) \subseteq proj(A)(\Psi_u(T')) \\
& \Rightarrow \quad \Psi_u(proj(A)(T_1 \parallel T_2)) \subseteq \Psi_l(T) \\
& \quad \text{assuming } \Psi_u(T') \subseteq \Psi_l(T) \text{ implies } T' \subseteq T \\
& \Rightarrow \quad proj(A)(T_1 \parallel T_2) \subseteq T.
\end{aligned}
$$

The theorem formalizes the above three assumptions (along with a fourth assumption for the renaming operation) and proves that they are sufficient to show that Ψ is a conservative approximation.

In addition, we show that if $\Psi' = (\Psi'_l, \Psi'_u)$ provides looser lower and upper bounds than a conservative approximation Ψ (*i.e.*, $\Psi'_l(T) \subseteq \Psi_l(T)$ and $\Psi_u(T) \subseteq \Psi'_u(T)$ for all T), then Ψ' is also a conservative approximation. Also, the functional composition of two conservative approximations yields another conservative approximation.

Theorem 2.35. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{T}$ to $\mathcal{T}'$. Assume that for all $T \in \mathcal{T}$, the signature of $\Psi_l(T)$ and $\Psi_u(T)$ is γ. If the following propositions A1 through A4 are satisfied for all trace structures T, T_1 and T_2 in $\mathcal{T}$, then Ψ is a conservative approximation.

A1. $\Psi_u(T_1 \parallel T_2) \subseteq \Psi_u(T_1) \parallel \Psi_u(T_2)$.

A2. $\Psi_u(proj(B)(T)) \subseteq proj(B)(\Psi_u(T))$.

A3. $\Psi_u(rename(r)(T)) \subseteq rename(r)(\Psi_u(T))$.

A4. If $\Psi_u(T_1) \subseteq \Psi_l(T_2)$, then $T_1 \subseteq T_2$.

Proof. Let E be an arbitrary expression potentially involving parallel composition, projection and renaming of trace structures in $\mathcal{T}$. Let E' be formed from E be replacing every instance of each trace structure T with $\Psi_u(T)$. Let T_1 be a trace structure in $\mathcal{T}$, and assume $E' \subseteq \Psi_l(T_1)$. We must show that $E \subseteq T_1$.

Using A1, A2 and A3, it is easy to prove by induction over the structure of E that $\Psi_u(E) \subseteq E'$. Therefore, $\Psi_u(E) \subseteq \Psi_l(T_1)$. By A4, $E \subseteq T_1$.

□

Theorem 2.36. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$. If $\Psi' = (\Psi'_l, \Psi'_u)$ is such that $\Psi'_l(T) \subseteq \Psi_l(T)$ and $\Psi_u(T) \subseteq \Psi'_u(T)$ for all $T \in \mathcal{T}$, then Ψ' is a conservative approximation.

Proof. Clearly, for all $T \in \mathcal{T}$, the signature of $\Psi'_l(T')$ and $\Psi'_u(T)$ is γ. Let E be an arbitrary expression potentially involving parallel composition, projection and renaming of trace structures in $\mathcal{T}$. Let E' be formed from E be replacing every instance of each trace structure T with $\Psi_u(T)$, and let E'' be similarly formed from E by using Ψ'_u. Let T_1 be a trace structure in $\mathcal{T}$, and assume $E'' \subseteq \Psi'_l(T_1)$. We must show that $E \subseteq T_1$.

Recall that by theorem 2.26, parallel composition, projection and renaming are monotonic with respect to trace structure containment. Thus, $E' \subseteq E''$, since $\Psi_u(T) \subseteq \Psi'_u(T)$ for every T_1. This implies $E' \subseteq \Psi_l(T_1)$, since $E'' \subseteq \Psi'_l(T_1)$ and $\Psi'_l(T_1) \subseteq \Psi_l(T_1)$. Therefore, $E \subseteq T_1$, since Ψ is a conservative approximation.

□

Theorem 2.37. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$, $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ and $\mathcal{A}''_C = (\mathcal{C}''_C, \mathcal{T}'')$ be trace structure algebras. Also, let $\Psi = (\Psi_l, \Psi_u)$ and $\Psi' = (\Psi'_l, \Psi'_u)$ be conservative approximations from $\mathcal{A}_C$ to $\mathcal{A}'_C$ and from $\mathcal{A}'_C$ to $\mathcal{A}''_C$, respectively. Then $\Psi'' = (\Psi''_l, \Psi''_u)$ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}''_C$, where

$$\begin{aligned} \Psi''_l(T) &= \Psi'_l(\Psi_l(T)) \\ \Psi''_u(T) &= \Psi'_u(\Psi_u(T)). \end{aligned}$$

Proof. Clearly, for all $T \in \mathcal{T}$, the signature of $\Psi'_l(T)$ and $\Psi'_u(T)$ is γ. Let E be an arbitrary expression potentially involving parallel composition, projection and renaming of trace structures in $\mathcal{T}$. Let E' be formed from E be replacing every instance of each trace structure T with $\Psi_u(T)$, and let E'' be similarly formed from E by using Ψ''_u. Let T_1 be a trace structure in $\mathcal{T}$, and assume $E'' \subseteq \Psi''_l(T_1)$. We must show that $E \subseteq T_1$.

By the definition of Ψ'', and since Ψ' is a conservative approximation, we know that $E' \subseteq \Psi'_l(T_1)$. Therefore, $E \subseteq T_1$, since Ψ is a conservative approximation.

□

2.4.1 Homomorphisms on Trace Algebras

We can define the notions of homomorphisms and isomorphisms between trace algebras. A homomorphism commutes with *rename* and *proj*; also, if x is a trace with alphabet A, then a homomorphism maps x to a trace with alphabet A. Thus, our definition of a homomorphism is quite standard. We will show in the next section how homomorphisms can be used to construct conservative approximations. An isomorphism is a homomorphism that is also a bijection. It is also possible to allow homomorphisms that can change the alphabet of a trace, but that is beyond the scope of this thesis.

Definition 2.38. Let $\mathcal{C}_C$ and $\mathcal{C}'_C$ be trace algebras. Let h be a function from $\mathcal{B}_C$ to $\mathcal{B}'_C$ such that for all alphabets A, if $x \in \mathcal{B}_C(A)$, then $h(x) \in \mathcal{B}'_C(A)$. The function h is a *homomorphism from* $\mathcal{C}_C$ *to* $\mathcal{C}'_C$ iff

$$\begin{aligned} h(\mathit{rename}(r)(x)) &= \mathit{rename}(r)(h(x)), \\ h(\mathit{proj}(B)(x)) &= \mathit{proj}(B)(h(x)). \end{aligned}$$

Chapter 3 has several examples of homomorphisms between trace algebras. Here is a simple example involving two of the trace algebras described in section 2.2.1. For all alphabets A, let h map traces in A^∞ to traces in 2^A such that

$$h(x) = \{a : \exists n\,[a = x(n)]\}.$$

It is easy to show that h is a homomorphism. Applying h to a trace abstracts away information about the order of events; all that remains is the set of actions that occurred one or more times.

Definition 2.39. A homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$ is an *isomorphism* iff it is a bijection. $\mathcal{C}_C$ are $\mathcal{C}'_C$ *isomorphic* iff there exists an isomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$.

Clearly if h is an isomorphism, then so is h^{-1}. Also, an isomorphism on trace algebras induces an isomorphism on trace structure algebras, as follows.

Corollary 2.40. Let h be an isomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras such that

$$\begin{aligned} (\gamma, P) \in \mathcal{T} &\Rightarrow (\gamma, h(P)) \in \mathcal{T}' \\ (\gamma, P') \in \mathcal{T}' &\Rightarrow \exists (\gamma, P) \in \mathcal{T}[P' = h(P)]. \end{aligned}$$

Then $\mathcal{A}_C$ and $\mathcal{A}'_C$ are isomorphic.

2.4.2 Approximations Induced by Homomorphisms

Let h be a trace algebra homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$, and let x and x' be traces in $\mathcal{C}_C$ and $\mathcal{C}'_C$, respectively, such that $h(x) = x'$. Intuitively, the trace x' is an abstraction of any trace y such that $h(y) = x'$. Thus, x' can be thought of as representing the set of all such y. Similarly, a set X' of traces in $\mathcal{C}'_C$ can be thought of as representing the largest set Y such that $h(Y) = X'$, where h is naturally extended to sets of traces. If $h(X) = X'$, then $X \subseteq Y$, so X' represents a kind of upper bound on the set X. This motivates using the function Ψ_u such that

$$\Psi_u(T) = (\gamma, h(P))$$

as the upper bound in a conservative approximation from a trace structure algebra over $\mathcal{C}_C$ to a trace structure algebra over $\mathcal{C}'_C$. A sufficient condition for a corresponding lower bound is: if $x \notin P$, then $h(x)$ is not in the set of possible traces of $\Psi_l(T)$. This leads to the definition

$$\Psi_l(T) = (\gamma, h(P) - h(\mathcal{B}_C(A) - P)).$$

The conservative approximation $\Psi = (\Psi_l, \Psi_u)$ is an example of a *conservative approximation induced by h*, which is formalized in the definition below using a slightly tighter lower bound for Ψ_l. Using this concept, if one proves that h is a homomorphism between two trace algebras (which is often quite easy), then one obtains a conservative approximation between trace structures with no additional effort. A conservative approximation induced by a homomorphism h is closely related to homomorphisms on ω-automata [57].

Definition 2.41. Let h be a homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$, and let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras. We naturally extend h to sets of traces. Assume Ψ_u and Ψ_l are functions from $\mathcal{T}$ to $\mathcal{T}'$ such that

$$\begin{aligned} \Psi_u(T) &\supseteq (\gamma, h(P)) \\ \Psi_l(T) &\subseteq (\gamma, h(P) - h(Y - P)), \end{aligned}$$

where

$$Y = \bigcup\{X \subseteq \mathcal{B}_C(A) : (\gamma, X) \in \mathcal{T} \wedge h(X) \subseteq h(P)\}.$$

By lemma 2.42 (below), $\Psi = (\Psi_l, \Psi_u)$ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$, which we call a *conservative approximation induced by h from* $\mathcal{A}_C$ *to* $\mathcal{A}'_C$. If the two set inequalities above are replaced by equalities, then Ψ is called the *tightest* conservative approximation induced by h from $\mathcal{A}_C$ to $\mathcal{A}'_C$.

Notice that $h(P) - h(\mathcal{B}_C(A) - P)$ is a subset of $h(P) - h(Y - P)$, so

$$\begin{aligned} \Psi_u(T) &= (\gamma, h(P)) \\ \Psi_l(T) &= (\gamma, h(P) - h(\mathcal{B}_C(A) - P)) \end{aligned}$$

(as described at the beginning of this section) is an example of a conservative approximation induced by h. This conservative approximation is independent of $\mathcal{T}$; the tightest conservative approximation induced by h depends on both h and $\mathcal{T}$.

Definition 2.41 defines both the class of conservative approximations induced by a homomorphism h and a distinguished approximation in that class, which we call the tightest conservative approximation induced by h. It is obvious that this distinguished approximation is in fact the tightest approximation within the class we defined. That is, if Ψ is the tightest conservative approximation induced by h and Ψ' is any conservative approximation in induced by h, then $\Psi'_l(T) \subseteq \Psi_l(T)$ and $\Psi_u(T) \subseteq \Psi'_u(T)$ for any trace structure T.

However, it is not immediately clear that class of approximations we defined includes all conservative approximations that might intuitively be "induced" by h. If there is a larger class of conservative approximations "induced" by h, then it might include an approximation that is tighter then the tightest one given in definition 2.41. We provide evidence that this is not the case in section 4.4, where we consider the *inverse* of a conservative approximation. This result depends on the particular set Y used in definition 2.41, and would not be true if we replaced

Y by a simpler expression such as $\mathcal{B}_C(A)$. With our current understanding, we cannot give any intuitive motivation for the definition of Y; it is simply the smallest set (which leads to the largest $\Psi_l(T)$) we could find that made the proof of lemma 2.42 go through.

It is straightforward to take the general notion of a conservative approximation induced by a homomorphism, and apply it to specific models. Simply construct trace algebras $\mathcal{C}$ and $\mathcal{C}'$, and a homomorphism h from $\mathcal{C}$ to $\mathcal{C}'$. Recall that these trace algebras act as models of individual behaviors. Using the results described so far in this chapter (without any additional proofs), one can construct the trace structure algebras $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ and $\mathcal{A}' = (\mathcal{C}', \mathcal{T}')$, and a conservative approximation Ψ induced by h (where $\mathcal{T}$ and $\mathcal{T}'$ are the sets of all trace structures over $\mathcal{C}$ and $\mathcal{C}'$, respectively). Thus, one need only construct two behavior models and a homomorphism between them to obtain two trace structure models along with a conservative approximation between the trace structure models.

The remainder of this section proves the claim made in definition 2.41: a conservative approximation induced by a homomorphism is in fact a conservative approximation.

Lemma 2.42. In definition 2.41, Ψ is a conservative approximation.

Proof. By definition 2.41, $\Psi = (\Psi_l, \Psi_u)$ is such that

$$\begin{aligned} \Psi_u(T) &\supseteq (\gamma, h(P)) \\ \Psi_l(T) &\subseteq (\gamma, h(P) - h(Y - P)), \end{aligned}$$

where

$$Y = \bigcup \{X \subseteq \mathcal{B}_C(A) : (\gamma, X) \in \mathcal{T} \wedge h(X) \subseteq h(P)\}.$$

By theorem 2.36, the current lemma is satisfied if Ψ is a conservative approximation when the two set inequalities above are replaced by equalities. Thus, we need only consider the case where

$$\begin{aligned} \Psi_u(T) &= (\gamma, h(P)) \\ \Psi_l(T) &= (\gamma, h(P) - h(Y - P)). \end{aligned}$$

By theorem 2.35, we can show that Ψ is a conservative approximation by showing that it satisfies A1 through A4.

Lemma 2.43. Ψ satisfies A1.

Proof. Let $T = T_1 \parallel T_2$; then

$$P = \{x \in \mathcal{B}_C(A) : proj(A_1)(x) \in P_1 \wedge proj(A_2)(x) \in P_2\}.$$

Let $T' = \Psi_u(T_1) \parallel \Psi_u(T_2)$; then

$$P' = \{x' \in \mathcal{B}'_C(A) : proj(A_1)(x') \in h(P_1) \wedge proj(A_2)(x') \in h(P_2)\}.$$

We must show that $h(P) \subseteq P'$.

$$\begin{aligned}
h(P) &= \{h(x) \in \mathcal{B}'_C(A) : proj(A_1)(x) \in P_1 \wedge proj(A_2)(x) \in P_2\} \\
&\subseteq \{h(x) \in \mathcal{B}'_C(A) : h(proj(A_1)(x)) \in h(P_1) \\
&\qquad \wedge h(proj(A_2)(x)) \in h(P_2)\} \\
&\quad \text{since } h \text{ is a homomorphism} \\
&= \{h(x) \in \mathcal{B}'_C(A) : proj(A_1)(h(x)) \in h(P_1) \\
&\qquad \wedge proj(A_2)(h(x)) \in h(P_2)\} \\
&\subseteq \{x' \in \mathcal{B}'_C(A) : proj(A_1)(x') \in h(P_1) \wedge proj(A_2)(x') \in h(P_2)\} \\
&= P'.
\end{aligned}$$

□

Lemma 2.44. Ψ satisfies A2.

Proof.

$$\begin{aligned}
h(proj(B)(P)) &= \{h(proj(B)(x)) : x \in P\} \\
&\quad \text{since } h \text{ is a homomorphism} \\
&= \{proj(B)(h(x)) : x \in P\} \\
&= proj(B)(\{h(x) : x \in P\}) \\
&= proj(B)(h(P)).
\end{aligned}$$

□

Lemma 2.45. Ψ satisfies A3.

Proof.

$$\begin{aligned} h(\mathit{rename}(r)(P)) &= \{h(\mathit{rename}(r)(x)) : x \in P\} \\ &\quad \text{since } h \text{ is a homomorphism} \\ &= \{\mathit{rename}(r)(h(x)) : x \in P\} \\ &= \mathit{rename}(r)(\{h(x) : x \in P\}) \\ &= \mathit{rename}(r)(h(P)). \end{aligned}$$

□

Lemma 2.46. Ψ satisfies A4.

Proof. Assume $\Psi_u(T_1) \subseteq \Psi_l(T_2)$. Then $A_1 = A_2$; let $A = A_1$. We must show that $P_1 \subseteq P_2$.

Let $x \in P_1$ and

$$Y = \bigcup\{X \subseteq \mathcal{B}_C(A) : (\gamma, X) \in \mathcal{T} \wedge h(X) \subseteq h(P_2)\}.$$

By the definition of Ψ, the assumption $\Psi_u(T_1) \subseteq \Psi_l(T_2)$ implies $h(P_1) \subseteq h(P_2) - h(Y - P_2)$. Thus, by the definition of Y, and since $(\gamma, P_1) \in \mathcal{T}$, we know $P_1 \subseteq Y$. Therefore, $x \in Y$.

We show that $P_1 \subseteq P_2$ with the following series of implications:

$$\begin{aligned} x \in P_1 &\Rightarrow h(x) \in h(P_1) \\ &\quad \text{since } h(P_1) \subseteq h(P_2) - h(Y - P_2) \\ &\Rightarrow h(x) \in h(P_2) - h(Y - P_2) \\ &\Rightarrow h(x) \notin h(Y - P_2) \\ &\quad \text{since } x \in Y - P_2 \text{ implies } h(x) \in h(Y - P_2) \\ &\Rightarrow x \notin Y - P_2 \\ &\quad \text{since } x \in Y \\ &\Rightarrow x \in P_2. \end{aligned}$$

□

□

2.5 Summary

It is worthwhile to summarize the results of this chapter and to described how they are applied and extended in the remainder of the thesis. We began by defining *concurrency algebra,* an abstract algebra in which each element of the domain represents an agent (def. 2.6, p. 23). Associated with each agent is a *signature,* which is a set of input symbols along with a (disjoint) set of output symbols. Each of these symbols might represent a wire in a circuit or message that can be sent between communicating processes, etc. The union of the inputs and the outputs is the *alphabet* of a signature. A concurrency algebra has three operations on agents: parallel composition, projection and renaming. These operations must satisfy axioms C1 through C9, the *axioms of concurrency algebra.* These axioms formalize certain minimum requirements that any agent model should be expected to satisfy.

Concurrency algebra includes no notion of what it means for an agent to satisfy a specification. We address this by using *trace set containment,* which is a generalization of standard verification techniques based on *language containment.* Each agent is represented by a *trace structure,* which is an ordered pair of a signature γ and a set P of *possible traces.* Each trace in P represents a possible behavior of the agent. Both implementations and specifications are represented by trace structures. One trace structure satisfies the specification given by another trace structure iff the set of possible traces of the first is contained in the set of possible traces of the second.

The above description of trace structures does not say what kinds of mathematical objects are used as traces. In normal language containment methods, a trace is a finite or infinite sequence, so a set of traces is a formal language. We want to be much more general than this, because we do not want our use of trace structures to limit the kinds of real-time models we can consider. On the other hand, we do not want to allow completely arbitrary traces because we want to have general theorems that are true of all trace structures (so the theorems do not have to be reproven every time a new class of trace structures is constructed).

We satisfy these constraints by using the idea of a *trace algebra.* A trace algebra (def. 4.20, p. 86) is an abstract algebra with a set of traces as its domain, where each trace is interpreted as an abstraction of a physical behavior. Traces are classified according to their alphabet. There are two operations in a trace algebra: projection and renaming. These operations must satisfy axioms T1 through T8, the *axioms of trace algebra.* Other than these axioms, no other restrictions are placed on what kinds of mathematical objects can be used as traces in a trace algebra.

Once trace algebra is formalized, it is possible to formalize trace structures. The set of *trace structures* (def. 2.15, p. 33) over a trace algebra $\mathcal{C}$ is the set of ordered pairs (γ, P), where γ is a signature and P is a subset of the traces of $\mathcal{C}$ with the same alphabet as γ. A *trace structure algebra* is an ordered pair $\mathcal{A} = (\mathcal{C}, \mathcal{T})$, where $\mathcal{C}$ is a trace algebra and $\mathcal{T}$ is a subset of the set of trace structures over $\mathcal{C}$. The operations of parallel composition, projection and renaming are defined on trace structures in $\mathcal{T}$ using the operations of projection and renaming on individual traces in $\mathcal{C}$ (def. 2.18, def. 2.19 and def. 2.20, p 33). The set of trace structures $\mathcal{T}$ must be closed under these operations. The axioms of trace algebra are quite weak, but they are strong enough to guarantee that the operations on trace structures satisfy the axioms of concurrency algebra. Thus, a trace structure algebra is a special case of a concurrency algebra.

Using these ideas to construct agent models only requires constructing a domain of traces, along with projection and renaming operations, and proving that they satisfy the axioms of trace algebra. A trace structure algebra, which is guaranteed to satisfy the axioms of concurrency algebra, can be constructed from the trace algebra without having to prove any additional theorems. Thus, our general results greatly simplify the task of constructing new agent models.

One of the uses of being able to easily build new process models is to study the relationships between models that can be efficiently mechanized and models that accurately represent physical reality. Ideally, correctness proofs (of trace set containment) in the efficient model would be logically equivalent to correctness proofs in the accurate model, but this is rarely the case. The best we can usually do is to have correctness in the efficient model imply correctness in the accurate model. This is formalized by using a *conservative approximation* from the accurate model to the efficient model (def. 2.34, p. 42). Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras. A conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$ is an ordered pair $\Psi = (\Psi_l, \Psi_u)$, where Ψ_l and Ψ_u are functions from $\mathcal{T}$ to $\mathcal{T}'$. For a given trace structure T in $\mathcal{A}_C$, the trace structure $\Psi_l(T)$ is a kind of lower bound of T, while $\Psi_u(T)$ is an upper bound (relative to trace set containment). By definition, if a verification problem in $\mathcal{C}_C$ is converted into a verification problem in $\mathcal{C}'_C$ by applying a conservative approximation Ψ, then a correctness proof in the latter problem implies a correctness result in the former problem.

A general method for constructing conservative approximations involves *homomorphisms on trace algebras* (def. 2.38, p. 45). A homomorphism from $\mathcal{C}$ to $\mathcal{C}'$ is just a function from the traces of $\mathcal{C}$ to the traces of $\mathcal{C}'$ that satisfies the standard homomorphism laws for the operations of trace algebra. A *conservative approximation induced by h* (def. 2.41, p. 46) is a conservative

approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$, for appropriate $\mathcal{T}$ and $\mathcal{T}'$.

We take advantage of these results in the next chapter, where we show that a continuous time model can be conservatively approximated by a discrete time model. We need only construct the appropriate trace algebras and homomorphisms; the trace structure algebras and the conservative approximations are obtained without any additional effort.

The conservative approximation defined in the chapter 3 maps to a discrete time model that represents simultaneity explicitly, which can make the model more expensive to automate. We would like to define a conservative approximation from this model to a discrete time model with interleaving semantics. Such an approximation cannot be induced from a homomorphism, so a new technique for constructing conservative approximations is needed. In chapter 4 we show how to use a *power set algebra over a trace algebra* (def. 4.1, p. 77), which is a trace algebra $\mathcal{C}$ where each trace in $\mathcal{C}$ is a set of traces in some other trace algebra $\mathcal{C}'$. The operations on traces in $\mathcal{C}$ are the natural extension to sets of the corresponding operations in $\mathcal{C}'$. For example, $\mathcal{C}'$ might have interleaved traces while a trace in $\mathcal{C}$ might be the set of interleavings of a trace with explicit simultaneity. Thus, $\mathcal{C}$ would be isomorphic to a more conventional representation of explicit simultaneity. The relationship between $\mathcal{C}$ and $\mathcal{C}'$ can be used to construct a conservative approximation from $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ to $\mathcal{A}' = (\mathcal{C}', \mathcal{T}')$ (def. 4.2, p. 78). This technique is used to complete the conservative approximation from continuous time to discrete time with interleaving semantics.

Chapter 3

Approximating Continuous Time

Methods for modeling and verifying real-time systems can be classified according to the type of timing model that is used. Continuous time allows more accurate modeling of physical reality. Discrete time models give an approximation to reality that can be automated more efficiently. This chapter develops several different trace structure algebras for modeling real-time systems, and describes conservative approximations from continuous time to discrete time.

3.1 Timing Models

In this chapter, we consider four different kinds of timing models:

- Continuous time,
- Quantized time with simultaneity,
- Quantized time with interleaving, and
- Synchronous time.

These models are informally described in this section; the formal definitions are given in the remainder of this chapter and in the next chapter. The classification is similar to that used by Alur and Dill [2], except that they did not differentiate the two quantized time models; they called them the *fictitious clock* model. They also used *discrete* to refer to what we call the synchronous model. We say a timing model is *discrete* if it is either synchronous or quantized.

For each of the four kinds of timing models, we can construct trace algebras with appropriate domains of traces. Using the results of the previous chapter, we can construct

corresponding trace structure algebras and conservative approximations between them. Thus, we obtain a hierarchy of domains of agent models at different levels of abstraction. In this section, we give an informal overview of the trace algebras and the conservative approximations that we use.

Continuous time is our most accurate and realistic timing model. The time of occurrence of each event is represented by a real number. As an example, consider the continuous time trace

$$x = \{(a, 0.2), (b, 2.3), (c, 2.8), (d, 2.8), (e, 5.3)\}.$$

The behavior is represented by a set of events; each event is an ordered pair designating an action and the time at which the action occurred. The order in which events occurred can be derived from the time stamps.

In a continuous time model, it is possible for an infinite number of events to occur in a finite period of time (Zeno's paradox). Such behaviors are not produced by the agents we wish to model, so we exclude them from the trace algebras we use.

The *synchronous time* model is the least accurate of the four types of models. In the synchronous time model, the time at which events occur is represented by integers, which can be derived by truncating the real numbered time stamps in the continuous time model. Thus, we can define a homomorphism h from continuous time traces to synchronous time traces such that

$$h(x) = \{(a, 0), (b, 2), (c, 2), (d, 2), (e, 5)\}.$$

Notice that in the synchronous time model, information about the order of occurrence of the b event and the c event is lost, as is information about the simultaneity of c and d. This is equivalent to assuming that all events that occur sometime during a given unit length period all occur simultaneously at some time point during that period. Since h is a homomorphism, there are conservative approximations induced by h from continuous time trace structures to synchronous time trace structures.

In some cases, we wish to truncate time stamps to integers, but also preserve information about the order and simultaneity of events. To do this, we begin by modeling continuous time behaviors with a different, but provably isomorphic, representation. For example, the example behavior x described above can be represented by the sequence

$$y = \langle(\{a\}, 0.2), (\{b\}, 2.3), (\{c, d\}, 2.8), (\{e\}, 5.3)\rangle.$$

Here a behavior is represented by a sequence of ordered pairs; each pair contains a non-empty set of actions (non-singleton sets represent simultaneous events) and a time stamp for the actions. The time stamps are real-valued, and must be (strictly) increasing. Notice that the order of events, and whether or not events are simultaneous, is represented more explicitly by the trace y than the trace x, even though they both represent the same behavior at the same level of abstraction. The isomorphism between these two kinds of continuous time traces depends on our assumption that only a finite number of events can occur in a finite period of time.

We can define a function h' that takes traces like y and truncates the time stamps:

$$h'(y) = \langle(\{a\},0),\ (\{b\},2),\ (\{c,d\},2),\ (\{x\},5)\rangle.$$

Information about the relative order of the the b event and the c event is preserved by h' because this information is represented explicitly in the trace y. The simultaneity of the c event and the d event is also reflected in $h'(y)$. The trace $h'(y)$ is a trace in a *quantized time with simultaneity* model, and h' is a homomorphism from continuous time to quantized time with simultaneity. In this model, a trace is a sequence of ordered pairs; each pair contains a non-empty set of actions and a time stamp for the actions. The time stamps are integer-valued, and must be (non-strictly) increasing. As with synchronous time, the homomorphism h' induces conservative approximations from continuous time trace structures to trace structures for quantized time with simultaneity. Quantized time models are of intermediate accuracy between the synchronous and continuous time models.

In *quantized time with interleaving,* simultaneity is modeled with nondeterminism. Thus, the continuous time behavior x is represented by two traces:

$$\begin{aligned} x' &= \langle(a,0),\ (b,2),\ (c,2),\ (d,2),\ (x,5)\rangle \\ x'' &= \langle(a,0),\ (b,2),\ (d,2),\ (c,2),\ (x,5)\rangle. \end{aligned}$$

It is possible to construct a conservative approximation from quantized time with simultaneity to quantized time with interleaving. However, it is not a conservative approximation induced by a homomorphism. It is an example of a *conservative approximation induced by a power set algebra* (def. 4.2, p. 78) and it depends on the way that traces with simultaneity can be represented by sets of interleaved traces.

The traces x' and x'' can be equivalently represented by infinite strings that include a special symbol φ. Each occurrence of φ represents the passage of one unit of time. Thus, the

traces x' and x'' are equivalent to

$$y' = a\varphi\varphi bcd\varphi\varphi\varphi x\varphi\varphi\cdots$$
$$y'' = a\varphi\varphi bdc\varphi\varphi\varphi x\varphi\varphi\cdots.$$

Each trace has an infinite number of φ to represent the passage of an unbounded amount of time (the normal interpretation of a complete trace). If we restrict our attention to safety properties, then only finite prefixes of these traces need be considered. The restriction to safety properties can be formalized in a general way using partial traces and a conservative approximation induced by a power set algebra (a complete trace is represented by the set of all partial traces that are prefixes of it). Thus, by a series of conservative approximations and isomorphisms between trace structure algebras, we go from a continuous time model to a quantized time with interleaving model that can be easily implemented using normal finite automata.

3.2 Modeling Continuous Time

We model continuous time behaviors with two different, but isomorphic, trace algebras: $\mathcal{C}_C^{CTU}$ ("continuous time unordered") and $\mathcal{C}_C^{CTO}$ ("continuous time ordered"). Having two representations simplifies the construction of conservative approximations from continuous time to discrete time. In particular, $\mathcal{C}_C^{CTU}$ is used in the mapping to synchronous time, and $\mathcal{C}_C^{CTO}$ is used in the mapping to quantized time with simultaneity.

In this section we describe $\mathcal{C}_C^{CTU}$, which uses traces consisting of a (possibly infinite) set of *events,* where each event is an ordered pair (a,t) in $A \times \Re^+$ that represents the occurrence of action a at time t. Only a finite number of actions are allowed in any finite period of time. The order of events is implicit and can be determined from their time stamps. For this reason, the model is called *unordered.*

Once $\mathcal{C}_C^{CTU}$ is shown to be a trace algebra, it is possible to construct the trace structure algebra $\mathcal{A}_C^{CTU} = (\mathcal{C}_C^{CTU}, \mathcal{T}^{CTU})$, where $\mathcal{T}^{CTU}$ is the set of all trace structures over $\mathcal{C}_C^{CTU}$. The construction makes use of the results of the previous chapter, and does not involve any additional proofs. This is an example of how constructing a trace algebra is all that is needed to construct a trace structure algebra which serves as a domain of agent models.

The remainder of the section formalizes the definition of $\mathcal{C}_C^{CTU}$, and proves that it is a trace algebra.

Note 3.1. The definition of a trace algebra is relative to a set of signals W (for example, see the definition of $\mathcal{C}_C^I$, def. 2.9, p. 27). In the sequel, the set of signals W will be implicit in the definitions of particular trace algebras. The phrase "all alphabets" is used to refer to all alphabets over W.

Definition 3.2. We define the trace algebra $\mathcal{C}_C^{CTU} = (\mathcal{B}_C, proj, rename)$ as follows. For all alphabets A, a trace x in $\mathcal{B}_C(A)$ is such that $x \subseteq A \times \Re^+$ and for any $t \in \Re^+$, there are only a finite number of $(a,t') \in x$ such that $t' \leq t$. If $x \in \mathcal{B}_C(A)$, then

$$\begin{aligned} proj(B)(x) &= \{(a,t) : (a,t) \in x \wedge a \in B\} \\ rename(r)(x) &= \{(r(a),t) : (a,t) \in x\}. \end{aligned}$$

Lemma 3.3. $\mathcal{C}_C^{CTU}$ is a trace algebra.

Proof. To show that $\mathcal{C}_C^{CTU}$ is a trace algebra, we must show that it satisfies T1 through T8. We only consider T4; the proofs for the remaining axioms are straightforward.

Lemma 3.4. $\mathcal{C}_C^{CTU}$ satisfies T4.

Proof. Let x and x' be traces in $\mathcal{B}_C(A)$ and $\mathcal{B}_C(A')$, respectively. Assume

$$proj(A \cap A')(x) = proj(A \cap A')(x'),$$

and let A'' be such that $A \cup A' \subseteq A''$. We must show that there exists $x'' \in \mathcal{B}_C(A'')$ such that $x = proj(A)(x'')$ and $x' = proj(A')(x'')$.

Let $x'' = (x \cup x')$. Notice that x'' is an element of $\mathcal{B}_C(A'')$, and

$$\begin{aligned} proj(A)(x'') &= \{(a,t) : (a,t) \in x'' \wedge a \in A\} \\ &\quad \text{since } x'' = x \cup x' \\ &= \{(a,t) : (a,t) \in x \wedge a \in A\} \cup \{(a,t) : (a,t) \in x' \wedge a \in A\} \\ &\quad \text{since } x \in \mathcal{B}_C(A) \text{ and } x' \in \mathcal{B}_C(A') \\ &= x \cup \{(a,t) : (a,t) \in x' \wedge a \in (A \cap A')\} \\ &\quad \text{since } proj(A \cap A')(x) = proj(A \cap A')(x') \\ &= x \cup \{(a,t) : (a,t) \in x \wedge a \in (A \cap A')\} \\ &= x. \end{aligned}$$

Similarly, $x' = proj(A')(x'')$. Therefore, x'' satisfies T4.

□

□

Definition 3.5. We define $\mathcal{A}_C^{CTU}$ to be the ordered pair $(\mathcal{C}_C^{CTU}, \mathcal{T}^{CTU})$, where $\mathcal{T}^{CTU}$ is the set of all trace structures over $\mathcal{C}_C^{CTU}$. By theorem 2.27, $\mathcal{A}_C^{CTU}$ is a trace structure algebra.

3.3 Modeling Synchronous Time

This section describes a trace algebra for modeling synchronous time, and shows how this model can be used to conservatively approximate continuous time. The trace algebra of synchronous time, $\mathcal{C}_C^{ST}$, is similar to $\mathcal{C}_C^{CTU}$ except that real-valued time stamps are replaced by integers. We define a homomorphism h from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{ST}$ that truncates the values of the time stamps for traces in $\mathcal{C}_C^{CTU}$. The homomorphism h allows us to construct a conservative approximation induced by h from trace structures over $\mathcal{C}_C^{CTU}$ to trace structures over $\mathcal{C}_C^{ST}$. This approximation is intended primarily as a simple example to illustrate mappings from continuous time. The conservative approximation used in the verification algorithms later in the thesis are based on quantized time rather than synchronous time.

One effect of the homomorphism h is that two continuous time events $(a, 2.2)$ and $(a, 2.7)$, for example, are both represented by a single synchronous time event $(a, 2)$. Thus, in addition to losing information about the exact time at which events occur, we also loose information about the number of events that occur in a given unit interval.

Although we do not provide the details here, it can be shown that $\mathcal{C}_C^{ST}$ is isomorphic to a trace algebra in which a trace over an alphabet A is an infinite sequence x over 2^A. In such a trace, if $a \in x(n)$, then action a occurred at time n. The representation of traces in $\mathcal{C}_C^{ST}$ that we have chosen simplifies the homomorphism (described below) from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{ST}$.

Definition 3.6. We define the trace algebra $\mathcal{C}_C^{ST}$ as follows. For all alphabets A, a trace x in $\mathcal{B}_C(A)$ is such that $x \subseteq A \times \mathcal{N}^+$. The definition of the operations on traces is identical to that of $\mathcal{C}_C^{CTU}$: if $x \in \mathcal{B}_C(A)$, then

$$\begin{aligned} proj(B)(x) &= \{(a,t) : (a,t) \in x \wedge a \in B\} \\ rename(r)(x) &= \{(r(a),t) : (a,t) \in x\}. \end{aligned}$$

Lemma 3.7. $\mathcal{C}_C^{ST}$ is a trace algebra.

Proof. The proof is analogous lemma 3.3, which showed that $\mathcal{C}_C^{CTU}$ is a trace algebra.

□

Definition 3.8. We define $\mathcal{A}_C^{ST}$ to be the ordered pair $(\mathcal{C}_C^{ST}, \mathcal{T}^{ST})$, where $\mathcal{T}^{ST}$ is the set of all trace structures over $\mathcal{C}_C^{ST}$. By theorem 2.27, $\mathcal{A}_C^{ST}$ is a trace structure algebra.

3.3.1 Approximating Continuous Time

In this section we describe the first of our conservative approximations from continuous time to discrete time. We construct the conservative approximation by first defining the homomorphism h from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{ST}$ such that

$$h(x) = \{(a, \lfloor t \rfloor) : (a,t) \in x\}$$

(see lemma 3.9 for a proof that h is a homomorphism). By lemma 2.42, h induces a conservative approximation from trace structures over $\mathcal{C}_C^{CTU}$ to trace structures over $\mathcal{C}_C^{ST}$. This is an example of how the results of the previous chapter simplify the task of constructing a conservative approximation between two domains of agent models.

The tightest conservative approximation $\Psi = (\Psi_l, \Psi_u)$ induced by h from $\mathcal{A}_C^{CTU}$ to $\mathcal{A}_C^{ST}$ has

$$\begin{aligned} \Psi_u(T) &= (\gamma, h(P)) \\ \Psi_l(T) &= (\gamma, h(P) - h(Y - P)), \end{aligned}$$

where

$$Y = \bigcup\{X \subseteq \mathcal{B}_C^{CTU}(A) : (\gamma, X) \in \mathcal{T}^{CTU} \wedge h(X) \subseteq h(P)\}.$$

Recall that $\mathcal{T}^{CTU}$ is the set of all trace structures over $\mathcal{C}_C^{CTU}$. As an example of applying Ψ, let $T = (\gamma, P)$ be the trace structure in $\mathcal{A}_C^{CTU}$ such that

$$\begin{aligned} A &= \{a\} \\ \gamma &= (\emptyset, A) \\ P &= \{\{(a,t)\} \in \mathcal{B}_C^{CTU}(A) : 0.5 \leq t < 2.5\}. \end{aligned}$$

This gives

$$\begin{aligned} Y &= \bigcup\{X \subseteq \mathcal{B}_C^{CTU}(A) : h(X) \subseteq h(P)\} \\ &= \bigcup\{X \subseteq \mathcal{B}_C^{CTU}(A) : h(X) \subseteq \{\{(a,0)\}, \{(a,1)\}, \{(a,2)\}\}\} \\ &= \bigcup\{X \subseteq \mathcal{B}_C^{CTU}(A) : X \subseteq \{\{(a,t)\} \in \mathcal{B}_C^{CTU}(A) : 0 \leq t < 3\}\} \\ &= \{\{(a,t)\} \in \mathcal{B}_C^{CTU}(A) : 0 \leq t < 3\} \end{aligned}$$

$$\begin{aligned} \Psi_u(T) &= (\gamma, h(P)) \\ &= (\gamma, \{\{(a,0)\}, \{(a,1)\}, \{(a,2)\}\}), \end{aligned}$$

$$\begin{aligned} \Psi_l(T) &= (\gamma, h(P) - h(Y - P)) \\ &= (\gamma, h(P) - h(\{\{(a,t)\} \in \mathcal{B}_C^{CTU}(A) : (0 \leq t < 0.5) \vee (2.5 \leq t < 3)\})) \\ &= (\gamma, h(P) - \{\{(a,0)\}, \{(a,2)\}\}) \\ &= (\gamma, \{\{(a,1)\}\}). \end{aligned}$$

Notice that $\Psi_l(T) \subseteq \Psi_u(T)$, as expected.

Let $\mathcal{T}$ be the set of all trace structures over $\mathcal{C}_C^{CTU}$ that have no events before time 0.5; that is,

$$\mathcal{T} = \{(\gamma, P) \in \mathcal{T}^{CTU} : \forall x \forall b \forall t[(x \in P \wedge (b,t) \in x) \Rightarrow t \geq 0.5]\}.$$

Let $\mathcal{A}_C = (\mathcal{C}_C^{CTU}, \mathcal{T})$; it can be shown that $\mathcal{A}_C$ is a trace structure algebra, although we do not give the details here. We can use $\mathcal{A}_C$ to demonstrate how the definition of a conservative approximation induced by a homomorphism depends on the set of trace structures in the trace structure algebra being approximated. Let h and T be as defined above (clearly $T \in \mathcal{T}$). Let $\Psi' = (\Psi_l', \Psi_u')$ be the tightest conservative approximation induced by h from $\mathcal{A}_C$ to $\mathcal{C}_C^{ST}$. Then $\Psi_u' = \Psi_u$ and

$$\begin{aligned} Y &= \bigcup\{X \subseteq \mathcal{B}_C^{CTU}(A) : (\gamma, X) \in \mathcal{T} \wedge h(X) \subseteq h(P)\} \\ &= \bigcup\{X \subseteq \mathcal{B}_C^{CTU}(A) : X \subseteq 2^{\{(a,t):0.5 \leq t\}} \wedge h(X) \subseteq \{\{(a,0)\}, \{(a,1)\}, \{(a,2)\}\}\} \\ &= \bigcup\{X \subseteq \mathcal{B}_C^{CTU}(A) : X \subseteq \{\{(a,t)\} \in \mathcal{B}_C^{CTU}(A) : 0.5 \leq t < 3\}\} \\ &= \{\{(a,t)\} \in \mathcal{B}_C^{CTU}(A) : 0.5 \leq t < 3\} \end{aligned}$$

$$\Psi_l'(T) = (\gamma, h(P) - h(Y - P))$$

$$
\begin{aligned}
&= (\gamma, h(P) - h(\{\{(a,t)\} \in B_C^{CTU}(A) : 2.5 \leq t < 3\})) \\
&= (\gamma, h(P) - \{\{(a,2)\}\}) \\
&= (\gamma, \{\{(a,0)\}, \{(a,1)\}\}).
\end{aligned}
$$

Notice that $\{(a,0)\}$ is a possible trace of $\Psi_l'(T)$ but not a possible trace of $\Psi_l(T)$. Thus, Ψ_l' gives a tighter bound than Ψ_l. This is a direct result of $\mathcal{T}$ being a proper subset of $\mathcal{T}^{CTU}$.

The remainder of this section proves that h is a homomorphism.

Lemma 3.9. The function h from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{ST}$ given by

$$h(x) = \{(a, \lfloor t \rfloor) : (a,t) \in x\}$$

is a homomorphism.

Proof. Clearly $h(x) \in B_C^{ST}(A)$. All that remains to be shown is that

$$
\begin{aligned}
h(proj(B)(x)) &= proj(B)(h(x)) \\
h(rename(r)(x)) &= rename(r)(h(x)).
\end{aligned}
$$

We consider *proj*; the *rename* case is also straightforward.

$$
\begin{aligned}
h(proj(B)(x)) &= h(\{(a,t) \in x : a \in B\}) \\
&= \{(a, \lfloor t \rfloor) : (a,t) \in x \wedge a \in B\} \\
&= proj(B)(\{(a, \lfloor t \rfloor) : (a,t) \in x\}) \\
&= proj(B)(h(x)).
\end{aligned}
$$

□

3.3.2 False Positive Example Revisited

In section 1.2, we described an example of how modeling a circuit in synchronous time can lead to a false positive verification result relative to a continuous time model. In section 3.3.1, we showed that a synchronous time model can be a conservative approximation of a continuous time model. To understand the relationship between these two results, it is helpful to analyze the false positive example more thoroughly.

The circuit behavior we used to demonstrate the false positive result is represented in $\mathcal{C}_C^{CTU}$ by the complete trace

$$x = \{(w, 0), (x3, 1.3), (x2, 1.9), (y2, 2.3), (x1, 2.5), (z, 3.3)\}.$$

Consider the exclusive-or gate driving the signal $y1$ (see fig. 1.1, p. 14). Notice that $A = \{x1, x2, y1\}$ is the alphabet of that gate. Let

$$\begin{aligned} y &= proj(A)(x) \\ &= \{(x2.1.9), (x1, 2.5)\}. \end{aligned}$$

The complete trace y represents the local behavior of the gate driving $y1$ during the global behavior represented by the trace x. Thus, if T is the trace structure over $\mathcal{C}_C^{CTU}$ that represents the gate, then $y \in P$. Notice that there is no transition of the signal $y1$ in the trace y. This is because of the continuous time inertial delay model, since the inputs $x1$ and $x2$ have non-equal values only during a period that is shorter than the gate's minimum delay of 1.

Let h be the homomorphism from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{ST}$ described in section 3.3.1, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation induced by h from trace structures over $\mathcal{C}_C^{CTU}$ to trace structures over $\mathcal{C}_C^{ST}$. If we were to use the synchronous time model to verify the circuit relative to some specification, then we would construct the trace structure $T' = \Psi_u(T)$, which is a conservative, synchronous time model of the gate driving $y1$. By the definition of Ψ_u, we know $h(P) \subseteq P'$, where h is naturally extended to sets of traces (if Ψ is the tightest conservative approximation induced by h, then $h(P) = P'$). Since $y \in P$, we have that $y' \in P'$, where

$$\begin{aligned} y' &= h(y) \\ &= \{(x2, 1), (x1, 2)\}. \end{aligned}$$

However, in the synchronous time model that we informally used in section 1.2, the only behavior of the gate that could result from that sequence of inputs $x1$ and $x2$ is

$$y'' = \{(x2, 1), (y1, 2), (x1, 2), (y1, 3)\}.$$

A conservative, synchronous time model of the gate driving $y1$ must contain both of the traces y' and y''. The fact that the informal model used in section 1.2 did not contain the trace y' is the reason for the false positive verification result described there.

Notice that the trace y'' allows continuous time behaviors where the first three transitions (in order) are $x2$, $x1$ and $y1$. This is more conservative than necessary; such behaviors are not

possible in the continuous time model of the gate. A tighter approximation can be obtained by using quantized time instead of synchronous time, since quantized time preserves information about the order of events.

3.4 Modeling Quantized Time with Simultaneity

In this section, we define a trace algebra $\mathcal{C}_C^{QTS}$ for quantized time with simultaneity. A homomorphism from continuous time traces to $\mathcal{C}_C^{QTS}$ is used to construct a corresponding conservative approximation.

A trace in $\mathcal{C}_C^{QTS}$ with alphabet A consists of two (possibly infinite) sequences of the same length. The first is a sequence over $2^A - \{\emptyset\}$ representing a sequence of sets of simultaneous events. The second sequence provides an integer-valued time stamp for each set of events; it is a non-strictly increasing sequence.

Defining the homomorphism from continuous time traces to $\mathcal{C}_C^{QTS}$ involves defining a second trace algebra, $\mathcal{C}_C^{CTO}$, for continuous time traces. A trace in $\mathcal{C}_C^{CTO}$ is similar to a trace in $\mathcal{C}_C^{QTS}$ except that the sequence of time stamps is a strictly increasing sequence of real numbers. There is a homomorphism from $\mathcal{C}_C^{CTO}$ to $\mathcal{C}_C^{QTS}$; it simply truncates the values of the time stamps in each trace. This implies that there is a homomorphism from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{QTS}$, since $\mathcal{C}_C^{CTU}$ is isomorphic to $\mathcal{C}_C^{CTO}$.

The remainder of this section formalizes the definition of $\mathcal{C}_C^{QTS}$ and proves that it is a trace algebra.

Definition 3.10. We define the trace algebra $\mathcal{C}_C^{QTS}$ as follows. For all alphabets A, a trace $x = (u, \tau)$ in $\mathcal{B}_C(A)$ is such that

- u is a (possibly infinite) sequence over $2^A - \{\emptyset\}$,
- τ is a (possibly infinite) sequence over $\mathcal{N}^+$,
- u and τ are the same length,
- $n_0 \leq n_1$ implies $\tau(n_0) \leq \tau(n_1)$ (increasing), and
- if τ has infinite length, then it is unbounded:

$$\forall t' \in \mathcal{N}^+ [\exists n \in \mathcal{N}^+ [t' < \tau(n)]].$$

Let $x = (u, \tau)$ be a trace over some alphabet A.

- $proj(B)(x) = (u', \tau')$, where u' is the sequence formed from u by removing every symbol a not in B, and τ' is formed from τ by removing the corresponding time stamps. More formally, $len(u')$ and $len(\tau')$ are both equal to

$$|\{j \in \mathcal{N} : 0 \leq j < len(u) \wedge u(j) \cap B \neq \emptyset\}|.$$

  Also,

$$\begin{aligned} u'(k) &= u(n) \cap B \\ \tau'(k) &= \tau(n), \end{aligned}$$

  where n is the unique integer such that $u(n) \cap B \neq \emptyset$ and

$$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge u(j) \cap B \neq \emptyset\}|.$$

- $rename(r)(x) = (\lambda n \in \mathcal{N}^+[r(u(n))], \tau)$.

Lemma 3.11. $\mathcal{C}_C^{QTS}$ is a trace algebra.

Proof. To show that $\mathcal{C}_C^{QTS}$ is a trace algebra, we must show that it satisfies T1 through T8. We only consider T4; the proofs for the remaining axioms are straightforward.

Lemma 3.12. $\mathcal{C}_C^{QTS}$ satisfies T4.

Proof. Let $x = (u, \tau)$ and $x' = (u', \tau')$ be traces in $\mathcal{B}_C(A)$ and $\mathcal{B}_C(A')$, respectively. Assume

$$proj(A \cap A')(x) = proj(A \cap A')(x'),$$

and let A'' be such that $A \cup A' \subseteq A''$. We must show that there exists $x'' \in \mathcal{B}_C(A'')$ such that $x = proj(A)(x'')$ and $x' = proj(A')(x'')$.

We defined $\mathcal{C}_C^{QTS}$ so that traces are a pair of sequences. However, in this case it is useful to also think of a trace as a sequence of pairs; the isomorphism is obvious. Thus, we write $x(n)$ to denote the pair $(u(n), \tau(n))$.

We must show how x and x' can be combined to form an appropriate x''. Our strategy is to first split the sequence x into in infinite number of subsequences y_n such that the time stamp of all of the pairs in y_n is n. We also form y'_n from x' in analogous fashion. We show how to combine each y_n and y'_n into y''_n such that

$$x'' = y''_0 y''_1 y''_2 \cdots$$

has the desired property.

For every non-negative integer n, let y_n be the sequence over $(2^A - \{\emptyset\}) \times \{n\}$ such that

$$len(y_n) = |\{j : \tau(j) = n\}|$$

and

$$y_n(j) = x(k + j),$$

where k is the smallest integer such that $\tau(k) = n$. Since τ is unbounded when it is of infinite length, each y_n is of finite length. Thus, $y_n \in ((2^A - \{\emptyset\}) \times \{n\})^*$. Notice that

$$x = y_0\, y_1\, y_2 \cdots.$$

We define y'_n analogously to y_n.

Each sequence y_n is of the form

$$y_n = v_0\, (a_0, n) \cdots v_{j-1}\, (a_{j-1}, n)\, v_j,$$

where each $v_i \in ((2^{(A-A')} - \{\emptyset\}) \times \{n\})^*$ and each $a_i \in 2^A - \{\emptyset\}$ such that $a_i \cap A' \neq \emptyset$. Similarly, each y'_n is of the form

$$y'_n = v'_0\, (a'_0, n) \cdots v'_{j'-1}\, (a'_{j'-1}, n)\, v_{j'},$$

where each $v'_i \in ((2^{(A'-A)} - \{\emptyset\}) \times \{n\})^*$ and each $a'_i \in 2^{A'} - \{\emptyset\}$ such that $a'_i \cap A \neq \emptyset$.

The assumption $proj(A \cap A')(x) = proj(A \cap A')(x')$ implies that $j = j'$ and

$$a_i \cap (A \cap A') = a'_i \cap (A \cap A')$$

for all $i < j$. Since $a_i \in 2^A - \{\emptyset\}$ and $a'_i \in 2^{A'} - \{\emptyset\}$, this implies

$$a_i \cap A' = a'_i \cap A.$$

Let

$$y''_n = v_0\, v'_0\, (a_0 \cup a'_0,\, n) \cdots v_{j-1}\, v'_{j-1}\, (a_{j-1} \cup a'_{j-1},\, n)\, v_j\, v'_j.$$

Notice that

$$\begin{aligned} &proj(A)(y''_n) \\ &\quad = proj(A)(v_0\, v'_0\, (a_0 \cup a'_0,\, n) \cdots v_{j-1}\, v'_{j-1}\, (a_{j-1} \cup a'_{j-1},\, n)\, v_j\, v'_j) \\ &\quad \text{since } v'_i \in ((2^{(A'-A)} - \{\emptyset\}) \times \{n\})^* \end{aligned}$$

$$
\begin{aligned}
&= proj(A)(v_0\,(a_0 \cup a'_0, n) \cdots v_{j-1}\,(a_{j-1} \cup a'_{j-1}, n)\,v_j) \\
&\text{since } a'_i \cap A \subseteq a_i \\
&= proj(A)(v_0\,(a_0, n) \cdots v_{j-1}\,(a_{j-1}, n)\,v_j) \\
&\text{since } v_i \in ((2^{(A-A')} - \{\emptyset\}) \times \{n\})^* \text{ and } a_i \in 2^A - \{\emptyset\} \\
&= v_0\,(a_0, n) \cdots v_{j-1}\,(a_{j-1}, n)\,v_j \\
&= y_n.
\end{aligned}
$$

Similarly, $proj(A')(y''_n) = y'_n$.

Let

$$x'' = y''_0\, y''_1\, y''_2 \cdots.$$

Then,

$$
\begin{aligned}
proj(A)(x'') &= proj(A)(y''_0\, y''_1\, y''_2 \cdots) \\
&= proj(A)(y''_0)\; proj(A)(y''_1)\; proj(A)(y''_2) \cdots \\
&= y_0\, y_1\, y_2 \cdots \\
&= x.
\end{aligned}
$$

Similarly, $x' = proj(A')(x'')$. Therefore, x'' satisfies T4.

□

□

Definition 3.13. We define $\mathcal{A}_C^{QTS}$ to be the ordered pair $(\mathcal{C}_C^{QTS}, \mathcal{T}^{QTS})$, where $\mathcal{T}^{QTS}$ is the set of all trace structures over $\mathcal{C}_C^{QTS}$. By theorem 2.27, $\mathcal{A}_C^{QTS}$ is a trace structure algebra.

3.4.1 Approximating Continuous Time

In this section we describe our next conservative approximation from continuous to discrete time. The first step is define another trace algebra, $\mathcal{C}_C^{CTO}$, for representing continuous time, and show that it is isomorphic to $\mathcal{C}_C^{CTU}$. A trace x in $\mathcal{C}_C^{CTO}$ is an ordered pair (u, τ) like $\mathcal{C}_C^{QTS}$ except τ is a strictly increasing sequence of real numbers rather than a sequence of integers. We define a homomorphism h from $\mathcal{C}_C^{CTO}$ to $\mathcal{C}_C^{QTS}$ such that if $x = (u, \tau)$ is a trace in $\mathcal{C}_C^{CTO}$, then $h(x) = (u, \tau')$, where $\tau'(n) = \lfloor \tau(n) \rfloor$. By lemma 2.42, h induces a conservative approximation from trace structures over $\mathcal{C}_C^{CTO}$ to trace structures over $\mathcal{C}_C^{QTS}$.

This conservative approximation is analogous to one described in section 3.3.1, which was induced by a homomorphism from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{ST}$.

The remainder of this section proves these claims. We begin by formally defining $\mathcal{C}_C^{CTO}$ and showing that it is a trace algebra and is isomorphic to $\mathcal{C}_C^{CTU}$.

Definition 3.14. We define the trace algebra $\mathcal{C}_C^{CTO}$ as follows. For all alphabets A, a trace $x = (u, \tau)$ in $\mathcal{B}_C(A)$ is such that

- u is a (possibly infinite) sequence over $2^A - \{\emptyset\}$,
- τ is a (possibly infinite) sequence over $\Re^{\neq}$,
- u and τ are the same length,
- $n_0 < n_1$ implies $\tau(n_0) < \tau(n_1)$ (strictly increasing), and
- if τ has infinite length, then it is unbounded:

$$\forall t' \in \Re^{\neq}[\exists n \in \mathcal{N}^{\neq}[t' < \tau(n)]].$$

Let $x = (u, \tau)$ be a trace over some alphabet A. The definitions of *proj* and *rename* are identical to those for $\mathcal{C}_C^{QTS}$:

- $proj(B)(x) = (u', \tau')$, where u' is the sequence formed from u by removing every symbol a not in B, and τ' is formed from τ by removing the corresponding time stamps. More formally, $len(u')$ and $len(\tau')$ are both equal to

$$|\{j \in \mathcal{N} : 0 \leq j < len(u) \wedge u(j) \cap B \neq \emptyset\}|.$$

Also,

$$\begin{aligned} u'(k) &= u(n) \cap B \\ \tau'(k) &= \tau(n), \end{aligned}$$

where n is the unique integer such that $u(n) \cap B \neq \emptyset$ and

$$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge u(j) \cap B \neq \emptyset\}|.$$

- $rename(r)(x) = (\lambda n \in \mathcal{N}^{\neq}[r(u(n))],\ \tau)$.

To show that $\mathcal{C}_C^{CTO}$ is a trace algebra, it is sufficient to show that there is a isomorphism h from $\mathcal{C}_C^{CTO}$ to $\mathcal{C}_C^{CTU}$. Since $\mathcal{C}_C^{CTU}$ is a trace algebra, this demonstrates that $\mathcal{C}_C^{CTO}$ is a trace algebra, as well as showing that it is isomorphic to $\mathcal{C}_C^{CTU}$.

Lemma 3.15. $\mathcal{C}_C^{CTO}$ is a trace algebra and is isomorphic to $\mathcal{C}_C^{CTU}$.

Proof. Let $x = (u, \tau)$ be a trace in $\mathcal{B}_C^{CTO}(A)$. We define h such that

$$h(x) = \{(a,t) : \exists n < len(u)[a \in u(n) \wedge t = \tau(n)]\}.$$

It can be shown that h is a surjection, since traces in $\mathcal{B}_C^{CTU}$ have only a finite number of actions during any finite period of time.

Also, it straightforward to show that h is an injection and for all $x' \in \mathcal{B}_C^{CTU}(A)$.

$$h^{-1}(x') = (u, \tau).$$

where u and τ are uniquely determined by the following constraints. First, the length of u and τ is

$$|\{t' : \exists a \in A[(a,t') \in x]\}|.$$

Second, if $n < len(u)$ then $\tau(n)$ is the unique real number such that $\exists a[(a,\tau(n)) \in x]$ and

$$n = |\{t' < \tau(n) : \exists a \in A[(a,t') \in x]\}|.$$

Third, if $n < len(u)$ then

$$u(n) = \{a : (a,\tau(n)) \in x\}.$$

It is also straightforward of show that h is a homomorphism. For example, the reader can easily verify that if $x = (u, \tau)$ is a trace in $\mathcal{B}_C^{CTO}(A)$, then both $proj(B)(h(x))$ and $h(proj(B)(x))$ are equal to

$$\{(a,t) : \exists n < len(u)[a \in (u(n) \cap B) \wedge t = \tau(n)]\}.$$

□

Lemma 3.16. Let h be the function from $\mathcal{C}_C^{CTO}$ to $\mathcal{C}_C^{QTS}$ such that if $x = (u, \tau)$ is an element of $\mathcal{B}_C^{CTO}(A)$, then

$$h(x) = (u, \tau'),$$

where

$$\tau'(n) = \lfloor \tau(n) \rfloor.$$

Then h is a homomorphism.

Proof. Clearly $h(x) \in \mathcal{B}_C^{QTS}(A)$. All that remains to be shown is that

$$\begin{aligned} h(proj(B)(x)) &= proj(B)(h(x)) \\ h(rename(r)(x)) &= rename(r)(h(x)). \end{aligned}$$

The reader can verify that both $h(proj(B)(x))$ and $proj(B)(h(x))$ are equal to (u'', τ''), where $len(u'')$ and $len(\tau'')$ are both equal to

$$|\{j \in \mathcal{N} : 0 \leq j < len(u) \wedge u(j) \cap B \neq \emptyset\}|$$

and

$$\begin{aligned} u''(k) &= u(n) \cap B \\ \tau''(k) &= \lfloor \tau(n) \rfloor, \end{aligned}$$

and n is the unique integer such that $u(n) \cap B \neq \emptyset$ and

$$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge u(j) \cap B \neq \emptyset\}|.$$

Also,

$$\begin{aligned} h(rename(r)(x)) &= h((\lambda n \in \mathcal{N}^+ [r(u(n))], \tau)) \\ &= (\lambda n \in \mathcal{N}^+ [r(u(n))], \lambda n \in \mathcal{N}^+ [\lfloor \tau(n) \rfloor]) \\ &= rename(r)((u, \lambda n \in \mathcal{N}^+ [\lfloor \tau(n) \rfloor])) \\ &= rename(r)(h(x)). \end{aligned}$$

□

3.4.2 False Positive Example Revisited

In section 3.3.2 we analyzed the false positive example of section 1.2 with a conservative, synchronous time model. In this section, we do a similar analysis with a quantized time model.

Recall that the continuous time trace

$$y = \{(x2, 1.9), (x1, 2.5)\}$$

represents the local behavior of the gate driving $y1$ in the false positive example. Let z' be the result of applying our homomorphism h from $\mathcal{C}_C^{CTU}$ to $\mathcal{C}_C^{QTS}$:

$$\begin{aligned} z' &= h(y) \\ &= \langle(\{x2\}, 1), (\{x1\}, 2)\rangle. \end{aligned}$$

A trace structure over $\mathcal{C}_C^{QTS}$ representing the gate should also contain the trace

$$z'' = \langle(\{x2\}, 1), (\{y1\}, 2), (\{x1\}, 2), (\{y1\}, 3)\rangle.$$

representing a behavior in which the time between the $x2$ and $x1$ events is greater than or equal to 1, such as

$$\{(x2, 1.9), (y1, 2.9), (x1, 2.95), (y1, 3.95)\}.$$

Recall that the synchronous time model of the gate needed to include the traces

$$\begin{aligned} y' &= \{(x2, 1), (x1, 2)\} \\ y'' &= \{(x2, 1), (y1, 2), (x1, 2), (y1, 3)\}. \end{aligned}$$

As stated earlier, the trace y'' allows continuous time behaviors where the first three transitions (in order) are $x2$, $x1$ and $y1$. This is more conservative than necessary because such behaviors are not possible in the continuous time model of the gate. Since the trace z'' does not allow such continuous time behaviors, it is a tighter approximation than y'' is.

3.5 Application to Automatic Verification

Let us consider how conservative approximations from continuous time trace structures to discrete time trace structures can be applied in automatic verification. One method for mechanically verifying that a implementation satisfies a continuous time specification is as follows.

First, construct data structures for each of the continuous time trace structures representing the specification and the components of the implementation. Then, algorithmically convert each of the continuous time trace structures to discrete time, and then decide the verification problem in discrete time.

We do not use this method, however, because we want to avoid having to construct a machine readable representation for any continuous time trace structures. Instead, we recommend a method involving a specification language with both continuous and discrete time semantics. The discrete time semantics must be shown to be a conservative approximation of the continuous time semantics, for any specification written in that language. The user writes descriptions of the specification and the components in this specification language, keeping the continuous time semantics in mind. The descriptions are translated into discrete time trace structures that are used to decide the verification problem. The result is a conservative approximation of the continuous time verification problem the user had in mind, but continuous time trace structures are never constructed. Implementing a specification language that can be used in this way is an area for future research.

Our results so far describe a conservative approximation from continuous time to quantized time with simultaneity. Using trace structure algebra techniques described in the next chapter (*conservative approximations induced by power set algebras*), we can extend this conservative approximation to quantized time with interleaving. The verification method described above can be used for the extended conservative approximation, as well.

Chapter 4

Trace Algebra, Part II

This chapter describes more advanced features of trace algebra such as partial traces and conservative approximations induced by powerset algebras over trace algebras. We use these features to extend the conservative approximations described in the previous chapter.

4.1 Power Set Algebras over Trace Algebras

We begin with an example to motivate power set algebras over trace algebras. Let $\mathcal{C}_C^S$ be the trace algebra given by:

- $\mathcal{B}_C^S(A) = (2^A - \{\emptyset\})^\infty$.
- $proj^S(B)(x) = x'$, where x' is the sequence formed by intersecting B with each element of the sequence x and then removing any instances of the empty set that result. More formally, $len(x')$ is equal to

  $$|\{j \in \mathcal{N} : 0 \leq j < len(x) \wedge x(j) \cap B \neq \emptyset\}|,$$

  and

  $$x'(k) = x(n) \cap B,$$

  where n is the unique integer such that $x(n) \cap B \neq \emptyset$ and

  $$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge x(j) \cap B \neq \emptyset\}|.$$

- $rename^S(r)(x) = \lambda n \in \mathcal{N}^+[r(x(n))]$.

This trace algebra was also described in section 2.2.1; it is an untimed behavior model with explicit simultaneity. The proof that $\mathcal{C}_C^S$ is a trace algebra is left as an exercise to the reader.

It is well known that a trace with explicit simultaneity can be represented by its set of interleavings. We use this fact to construct a trace algebra $\mathcal{C}_C^{SI}$ that is isomorphic to $\mathcal{C}_C^S$ and that is a *power set algebra over* $\mathcal{C}_C^I$. Each trace in $\mathcal{C}_C^{SI}$ is a set of traces from $\mathcal{C}_C^I$; this set of traces is the set of interleavings of some trace in $\mathcal{C}_C^S$. The operations of $\mathcal{C}_C^{SI}$ are the same as those in $\mathcal{C}_C^I$ except that they are naturally extended to sets. The main result of this section is the description of how to use power set algebras to construct conservative approximations from (for example) trace structures over $\mathcal{C}_C^S$ to trace structures over $\mathcal{C}_C^I$. The approximations are independent of the details of the trace algebras; they depend only on the fact that $\mathcal{C}_C^{SI}$ is isomorphic to $\mathcal{C}_C^S$ and is a power set algebra over $\mathcal{C}_C^I$.

The construction of $\mathcal{C}_C^{SI}$ involves the function *interleave* from traces in $\mathcal{C}_C^S$ to sets of traces in $\mathcal{C}_C^I$. Let x be a trace in $\mathcal{B}_C^S$. For all $n \in \mathcal{N}^+$ such that $n \leq len(x)$, let

$$l_n(x) = \sum_{k=0}^{n-1} |x(k)|.$$

We define *interleave*(x) to be the set of traces $x' \in \mathcal{B}_C^I$ such that for all n,

$$x(n) = \{x'(k) : l_n(x) \leq k < l_{n+1}(x)\}.$$

Intuitively, this is the set of traces that can be formed from x by constructing a permutation of each of the sets $x(n)$ and then concatenating the permutations together. Notice that for all alphabets A, if $x \in \mathcal{B}_C^S(A)$, then *interleave*$(x) \subseteq \mathcal{B}_C^I(A)$.

The function *interleave* is an injection. To see this, let x_0 and x_1 be distinct traces in $\mathcal{B}_C^S$. Since x_0 and x_1 are distinct, there exists a smallest n such that $x_0(n) \neq x_1(n)$. Also, there exists a b such that either

$$b \in x_0(n) - x_1(n)$$

or

$$b \in x_1(n) - x_0(n).$$

Consider the first case; the second case is analogous. Since $b \in x_0(n)$, there exists a trace $x' \in$ *interleave*(x_0) such that $x'(l_n(x_0)) = b$. Since $b \notin x_1(n)$, there does not exist a trace

$x' \in interleave(x_1)$ such that $x'(l_n(x_1)) = b$. Since n is the index of the first element on which x_0 and x_1 differ, $l_n(x_0) = l_n(x_1)$. Thus,

$$interleave(x_0) \neq interleave(x_1).$$

Therefore, *interleave* is an injection.

Since *interleave* is an injection, we can use it to construct a trace algebra $\mathcal{C}_C^{SI}$ that is isomorphic to $\mathcal{C}_C^S$ and such that each trace is a set of traces from $\mathcal{C}_C^I$:

$$\begin{aligned} \mathcal{B}_C^{SI}(A) &= \{y \subseteq \mathcal{B}_C^I(A) : \exists x \in \mathcal{B}_C^S(A)\,[y = interleave(x)]\} \\ proj^{SI}(B)(y) &= interleave(proj^S(B)(interleave^{-1}(y))) \\ rename^{SI}(r)(y) &= interleave(rename^S(r)(interleave^{-1}(y))). \end{aligned}$$

The reader can verify the following identities, which relate the operations of $\mathcal{C}_C^S$ to those of $\mathcal{C}_C^I$:

$$\begin{aligned} interleave(proj^S(B)(x)) &= \{proj^I(B)(x') : x' \in interleave(x)\} \\ interleave(rename^S(r)(x)) &= \{rename^I(r)(x') : x' \in interleave(x)\}. \end{aligned}$$

These are used to show that the operations of $\mathcal{C}_C^{SI}$ are just the natural extension to sets of the corresponding operations of $\mathcal{C}_C^I$:

$$\begin{aligned} proj^{SI}(B)(y) &= interleave(proj^S(B)(interleave^{-1}(y))) \\ &= \{proj^I(B)(x') : x' \in interleave(interleave^{-1}(y))\} \\ &= \{proj^I(B)(x') : x' \in y\}; \end{aligned}$$

similarly,

$$rename^{SI}(r)(y) = \{rename^I(r)(x') : x' \in y\}.$$

These properties of $\mathcal{C}^{SI}$ make it *a power set algebra over* $\mathcal{C}_C^I$, as defined below.

Definition 4.1. Let $\mathcal{C}_C = (\mathcal{B}_C, proj, rename)$ and $\mathcal{C}' = (\mathcal{B}'_C, proj', rename')$ be trace algebras. We say $\mathcal{C}_C$ is *a power set algebra over* $\mathcal{C}'_C$ iff

- $x \in \mathcal{B}_C(A)$ implies $x \subseteq \mathcal{B}'_C(A)$,
- $proj(B)(x) = \{proj'(B)(x') : x' \in x\}$,
- $rename(r)(x) = \{rename'(r)(x') : x' \in x\}$.

We want to construct a conservative approximation from trace structures over $\mathcal{C}_C^{SI}$ to trace structures over $\mathcal{C}_C^I$ (which, because of the isomorphism between $\mathcal{C}_C^S$ and $\mathcal{C}_C^{SI}$, allows us to construct a conservative approximation from trace structures over $\mathcal{C}_C^S$ to trace structures over $\mathcal{C}_C^I$). To do this, we need to find a way that a set of traces in $\mathcal{C}_C^I$ can be interpreted as representing or approximating a set of traces in $\mathcal{C}_C^S$.

Let X be a subset of $\mathcal{B}_C^{SI}(A)$ for some alphabet A. Since each trace x in $\mathcal{C}_C^{SI}$ is a set of traces in $\mathcal{C}_C^I$, the set $P' = \cup X$ is well-defined and is a subset of $\mathcal{B}_C^I(A)$. The set P' can be thought of as representing the largest set P of traces in $\mathcal{C}_C^{SI}$ such that $P' = \cup P$. This is a standard way of using a set of interleaved traces to (approximately) represent a set of traces with explicit simultaneity or partial order semantics. Notice that P is the largest set of traces in $\mathcal{C}_C^{SI}$ such that $P' = \cup P$ if and only if

$$x \in \mathcal{B}_C^{SI}(A) \wedge x \subseteq P' \quad \Leftrightarrow \quad x \in P.$$

So the above logical equivalence specifies when P' represents P exactly. For a conservative approximation, we do not need to represent P exactly, but we do need to construct P'_u and P'_l (subsets of $\mathcal{B}_C^I(A)$) that represent upper and lower bounds on P. The above requirement for exactness can be split into two parts to form requirements for such upper and lower bounds:

$$x \in \mathcal{B}_C^{SI}(A) \wedge x \subseteq P'_u \quad \Leftarrow \quad x \in P$$
$$x \in \mathcal{B}_C^{SI}(A) \wedge x \subseteq P'_l \quad \Rightarrow \quad x \in P$$

The requirement that $x \in \mathcal{B}_C^{SI}(A)$ is redundant in the reverse implication since $P \subseteq \mathcal{B}_C^{SI}(A)$. This leads to the following definition of a class of conservative approximations from (for example) trace structures over $\mathcal{C}_C^{SI}$ to trace structures over $\mathcal{C}_C^I$.

Definition 4.2. Let $\mathcal{C}'_C$ be a trace algebra and let $\mathcal{C}_C$ be a power set algebra over $\mathcal{C}'_C$. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras. Let Ψ_u and Ψ_l be functions from $\mathcal{T}$ to $\mathcal{T}'$ such that if $T'_u = \Psi_u(T)$ and $T'_l = \Psi_l(T)$, then

$$\gamma'_u = \gamma$$
$$\gamma'_l = \gamma$$
$$x \subseteq P'_u \quad \Leftarrow \quad x \in P, \tag{4.1}$$
$$x \in \mathcal{B}_C(A) \wedge x \subseteq P'_l \quad \Rightarrow \quad x \in P. \tag{4.2}$$

By lemma 4.3 (below), $\Psi = (\Psi_l, \Psi_u)$ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$. We call Ψ a *conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$ induced by the power set algebra*

C. If

$$\begin{aligned} P'_u &= \cup P, \\ P'_l &= \cup P - \cup\{x \in \mathcal{B}_C(A) - P : x \subseteq \cup P\} \end{aligned}$$

then the above constraints on P'_u and P'_l are clearly satisfied. In this case, we call Ψ the *standard conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$ induced by C.*

In chapter 2, we were able to characterize the tightest conservative approximation induced by a homomorphism. An obvious question is whether there always exists a tightest conservative approximation induced by a given power set algebra. The smallest (by inclusion ordering) P'_u that satisfies formula (4.1) is clearly $P'_u = \cup P$. However, in general, there is no largest P'_l that satisfies formula (4.2), so there is no tightest conservative approximation. Our definition of the *standard* conservative approximation induced by a power set algebra is a compromise that works well in many cases.

The remainder of this section proves that a conservative approximation induced by a conservative approximation is in fact a conservative approximation.

Lemma 4.3. A conservative approximation induced by a power set algebra is a conservative approximation.

Proof. Adopt the notation used in definition 4.2. By theorem 2.36 (which states that a conservative approximation remains conservative when "loosened"), the current lemma is satisfied if Ψ is a conservative approximation when P'_u is the smallest set satisfying formula (4.1). Thus, we may assume that

$$\Psi_u(T) = (\gamma, \cup P).$$

We use theorem 2.35 to show that Ψ is a conservative approximation by showing that Ψ satisfies A1 through A4.

Lemma 4.4. Ψ satisfies A1.

Proof. Let $T = T_1 \parallel T_2$; then

$$P = \{x \in \mathcal{B}_C(A) : proj(A_1)(x) \in P_1 \wedge proj(A_2)(x) \in P_2\}.$$

Also, let $T_1' = \Psi_u(T_1)$, $T_2' = \Psi_u(T_2)$ and $T' = T_1' \parallel T_2'$. We must show that $\cup P \subseteq P'$.

$$\begin{aligned}
x' \in \cup P \;&\Leftrightarrow\; \exists x \in \mathcal{B}_C(A)[x' \in x \wedge proj(A_1)(x) \in P_1 \wedge proj(A_2)(x) \in P_2] \\
&\quad \text{by the definition of } \Psi_u \\
&\Rightarrow\; \exists x \in \mathcal{B}_C(A)[x' \in x \wedge proj(A_1)(x) \subseteq P_1' \wedge proj(A_2)(x) \subseteq P_2'] \\
&\quad \text{by the definition of } proj \text{ on traces in } \mathcal{C}_C \\
&\Rightarrow\; proj(A_1)(x') \in P_1' \wedge proj(A_2)(x') \in P_2'] \\
&\Leftrightarrow\; x' \in P'.
\end{aligned}$$

□

Lemma 4.5. Ψ satisfies A2.

Proof.

$$\begin{aligned}
\cup proj(B)(P) \;&=\; \{x' : \exists x \in proj(B)(P)[x' \in x]\} \\
&\quad \text{by the definition of } proj(B) \text{ on sets of traces in } \mathcal{C}_C \\
&=\; \{x' : \exists y \in P[x' \in proj(B)(y)]\} \\
&\quad \text{since } \forall x' \exists y'[x' = proj(B)(y')], \text{ by T4} \\
&=\; \{proj(B)(y') : \exists y \in P[proj(B)(y') \in proj(B)(y)]\} \\
&\quad \text{by the definition of } proj(B) \text{ on traces in } \mathcal{C}_C \\
&=\; \{proj(B)(y') : \exists y \in P[y' \in y]\} \\
&\quad \text{by the definition of } proj(B) \text{ on sets of traces in } \mathcal{C}_C' \\
&=\; proj(B)(\{y' : \exists y \in P[y' \in y]\}) \\
&=\; proj(B)(\cup P).
\end{aligned}$$

□

Lemma 4.6. Ψ satisfies A3.

Proof.

$$\begin{aligned}
&\cup rename(r)(P) \\
&=\; \{x' : \exists x \in rename(r)(P)[x' \in x]\}
\end{aligned}$$

by the definition of $rename(r)$ on sets of traces in $\mathcal{C}_C$

$= \{x' : \exists y \in P[x' \in rename(r)(y)]\}$

since $rename(r)$ is a bijection in any trace algebra

$= \{rename(r)(y') : \exists y \in P[rename(r)(y') \in rename(r)(y)]\}$

by the definition of $rename(r)$ on traces in $\mathcal{C}_C$

$= \{rename(r)(y') : \exists y \in P[y' \in y]\}$

by the definition of $rename(r)$ on sets of traces in $\mathcal{C}'_C$

$= rename(r)(\{y' : \exists y \in P[y' \in y]\})$

$= rename(r)(\cup P)$.

□

Lemma 4.7. Ψ satisfies A4.

Proof. Assume $\Psi_u(T_1) \subseteq \Psi_l(T_2)$. Then $A_1 = A_2$; let $A = A_1$. Also, let $T'_1 = \Psi_u(T_1)$ and $T'_2 = \Psi_l(T_2)$. We must show that $P_1 \subseteq P_2$.

$x \in P_1$

by the definition of Ψ_u

$\Rightarrow \; x \subseteq P'_1$

since $P'_1 \subseteq P'_2$

$\Rightarrow \; x \subseteq P'_2$

by the definition of Ψ_l

$\Rightarrow \; x \in P_2$.

□

□

4.2 Quantized Time with Interleaving Semantics

In this section, we describe two different, but isomorphic, trace algebras for quantized time with interleaving. The first, $\mathcal{C}_C^{QTI}$, is quite similar to $\mathcal{C}_C^{QTS}$, except that for a trace $x = (u, \tau)$ the

sequence u is over A rather than $2^A - \{\emptyset\}$. It is used to construct a conservative approximation from quantized time with simultaneity to quantized time with interleaving, which extends the conservative approximation from continuous time. The second, $\mathcal{C}_C^{QTI\varphi}$, has traces that are sequences over $A \cup \{\varphi\}$, where φ is a special symbol that indicates the passage of a unit of time [16, 17, 18]. For example, the trace $\varphi\varphi b\varphi$ represents a behavior in which a b event has a time stamp of 2.

The remainder of this section formalizes these ideas.

Definition 4.8. We define the trace algebra $\mathcal{C}_C^{QTI}$ as follows. For all alphabets A, a trace $x = (u, \tau)$ in $\mathcal{B}_C(A)$ is such that

- u is a (possibly infinite) sequence over A,
- τ is a (possibly infinite) sequence over $\mathcal{N}^+$,
- u and τ are the same length,
- $n_0 \leq n_1$ implies $\tau(n_0) \leq \tau(n_1)$ (increasing), and
- if τ has infinite length, then it is unbounded,

$$\forall t \in \mathcal{N}^+[\exists n \in \mathcal{N}^+[t < \tau(n)]].$$

Let $x = (u, \tau)$ be a trace over some alphabet A.

- $proj(B)(x) = (u', \tau')$, where u' is the sequence formed from u by removing every symbol a not in B, and τ' is formed from τ by removing the corresponding time stamps. More formally, $len(u')$ and $len(\tau')$ are both equal to

  $$|\{j \in \mathcal{N} : 0 \leq j < len(u) \wedge u(j) \in B\}|.$$

  Also,

  $$\begin{aligned} u'(k) &= u(n) \\ \tau'(k) &= \tau(n), \end{aligned}$$

  and n is the unique integer such that $u(n) \in B$ and

  $$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge u(j) \in B\}|$$

- $rename(r)(x) = (\lambda n \in \mathcal{N}^+[r(u(n))],\ \tau)$.

Lemma 4.9. $\mathcal{C}_C^{QTI}$ is a trace algebra.

Proof. The proof is analogous to lemma 3.11, which show that C_C^{QTS} is a trace algebra.

□

Definition 4.10. We define the trace algebra $C_C^{QTI\varphi}$ as follows:

- The set $\mathcal{B}_C(A)$ of traces over an alphabet A is the set of $x \in (A \cup \{\varphi\})^\omega$ such that φ appears infinitely often in x (we assume $\varphi \notin W$, see note 3.1, p. 59).
- If $x \in \mathcal{B}_C(A)$ and $B \subseteq A$, then $proj(B)(x)$ is the sequence formed from x by removing every symbol a not in $B \cup \{\varphi\}$. More formally,

$$proj^{QTI\varphi}(B)(x) = proj^I(B \cup \{\varphi\})(x).$$

- If $x \in \mathcal{B}_C(A)$ and r is a renaming function with domain A, then $rename(r)(x)$ is the sequence formed from x by replacing every $a \in A$ with $r(a)$.

Lemma 4.11. $C_C^{QTI\varphi}$ is a trace algebra.

Proof. The proof is a slight modification of the proof that C_C^I is a trace algebra (lemma 4.33), and is left as an exercise for the reader.

□

Definition 4.12. We define $\mathcal{A}_C^{QTI}$ to be the ordered pair $(C_C^{QTI}, \mathcal{T}^{QTI})$, where $\mathcal{T}^{QTI}$ is the set of all trace structures over C_C^{QTI}. By theorem 2.27, $\mathcal{A}_C^{QTI}$ is a trace structure algebra. $\mathcal{A}_C^{QTI\varphi}$ is similarly defined.

Lemma 4.13. $C_C^{QTI\varphi}$ is isomorphic to C_C^{QTI}.

Proof. It is sufficient to show that there is a bijection h from $C_C^{QTI\varphi}$ to C_C^{QTI} that satisfies the requirements of a homomorphism.

Let x be a trace in $\mathcal{B}_C^{QTI\varphi}(A)$. We define h such that $h(x) = (u, \tau)$, where

$$\begin{aligned} u(k) &= x(n) \\ \tau(k) &= n - k, \end{aligned}$$

and n is the unique integer such that $x(n) \neq \varphi$ and

$$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge x(j) \neq \varphi\}|.$$

It is straightforward, but tedious, to show that h is an bijection and that it satisfies the requirements for being a homomorphism. The proof is left as an exercise for the reader.

□

Corollary 4.14. $\mathcal{A}_C^{QTI\varphi}$ is isomorphic to $\mathcal{A}_C^{QTI}$.

4.2.1 Approximating Continuous Time

In this section we complete the conservative approximation from $\mathcal{A}_C^{CTU}$ to $\mathcal{A}_C^{QTI}$. Since we have already constructed a conservative approximation from $\mathcal{A}_C^{CTU}$ to $\mathcal{A}_C^{QTS}$, it is only necessary to go from $\mathcal{A}_C^{QTS}$ to $\mathcal{A}_C^{QTI}$. There exists a trace algebra $\mathcal{C}_C^{QTSI}$ that is a power set algebra over $\mathcal{C}_C^{QTI}$ and is isomorphic to $\mathcal{A}_C^{QTS}$. This allows us to construct a conservative approximation from $\mathcal{A}_C^{QTS}$ to $\mathcal{A}_C^{QTI}$. We define the set of *interleavings* of a trace in $\mathcal{C}_C^{QTS}$ to be a set of traces in $\mathcal{C}_C^{QTI}$. Each trace in $\mathcal{C}_C^{QTSI}$ is the set of interleavings of a trace in $\mathcal{C}_C^{QTS}$.

The remainder of this section proves these claims.

Definition 4.15. Let $x = (u, \tau)$ be a trace in $\mathcal{B}_C^{QTS}(A)$. For all $n \in \mathcal{N}^+$ such that $n \leq len(x)$,

$$l_n = \sum_{k=0}^{n-1} |u(k)|.$$

We define $interleave(x) \subseteq \mathcal{B}_C^{QTI}(A)$ to be the set of traces $x' = (u', \tau')$ such that for all n,

$$u(n) = \{u'(k) : l_n \leq k < l_{n+1}\}$$

and

$$\forall k[l_n \leq k < l_{n+1} \Rightarrow \tau(n) = \tau'(k)].$$

Definition 4.16. We define the trace algebra $\mathcal{C}_C^{QTSI}$ as follows. For all alphabets A,

$$\mathcal{B}_C^{QTSI}(A) = \{interleave(x) : x \in \mathcal{B}_C^{QTS}(A)\}.$$

The operations on traces of $\mathcal{C}_C^{QTSI}$ are the natural extension of the operations of $\mathcal{C}_C^{QTI}$ to the sets of $\mathcal{C}_C^{QTI}$ traces in $\mathcal{B}_C^{QTSI}$.

Corollary 4.17. $\mathcal{C}_C^{QTSI}$ is a power set algebra over $\mathcal{C}_C^{QTI}$.

Lemma 4.18. $\mathcal{C}_C^{QTSI}$ is a trace algebra and it is isomorphic to $\mathcal{C}_C^{QTS}$.

Proof. Since $\mathcal{C}_C^{QTS}$ is a trace algebra, it is sufficient to show that there is a bijection h from $\mathcal{C}_C^{QTS}$ to $\mathcal{C}_C^{QTSI}$ that satisfies the requirements of a homomorphism. This demonstrates that $\mathcal{C}_C^{QTSI}$ is a trace algebra, as well as showing that it is isomorphic to $\mathcal{C}_C^{QTS}$.

Let $x = (u, \tau)$ be a trace in $\mathcal{B}_C^{QTS}(A)$. The obvious candidate for h is

$$h(x) = interleave(x).$$

It is straightforward, but tedious, to show that h is an bijection and that it satisfies the requirements for being a strong homomorphism. The proof is left as an exercise for the reader.

□

Theorem 4.19. There is a conservative approximation (up to isomorphism) from $\mathcal{A}_C^{QTS}$ to $\mathcal{A}_C^{QTI}$.

Proof. Since $\mathcal{C}^{QTSI}$ is a power set algebra over $\mathcal{C}^{QTI}$, definition 4.2 can be used to construct a conservative approximation from trace structures over $\mathcal{C}^{QTSI}$ to trace structures over $\mathcal{C}^{QTI}$. This can be used to construct a conservative approximation from $\mathcal{A}^{QTS}$ to $\mathcal{A}^{QTI}$ since $\mathcal{C}^{QTS}$ is isomorphic to $\mathcal{C}^{QTSI}$.

□

4.3 Partial Traces

Recall the distinction described in section 2.2 between two different kinds of behaviors: *complete* behaviors and *partial* behaviors. A complete behavior has no endpoint; a partial behavior has an endpoint and can be a prefix of a complete behavior or of another partial behavior. *Complete traces* and *partial traces* are used to model complete and partial behaviors, respectively. So far we have only considered trace algebras and trace structure algebras that contain complete traces but no partial traces. In the next several sections we extend these algebras to include partial traces.

4.3.1 Trace Algebra with Partial Traces

A trace algebra C with partial traces includes, for each alphabet, a set of complete traces and a set of partial traces. In addition, a concatenation operation "$\cdot$" is included that takes a partial trace as its first argument and a partial or complete trace as its second argument. Besides the axioms T1 through T8 for trace algebra without partial traces, trace algebra with partial traces must satisfy axioms T9 through T19, which state requirements on the concatenation operation and on the effects of projection and renaming on partial traces.

Definition 4.20. A *trace algebra with partial traces C over W* is a 5-tuple

$$(\mathcal{B}_C, \mathcal{B}_P, proj, rename, \cdot).$$

For every alphabet A over W, $\mathcal{B}_C(A)$ and $\mathcal{B}_P(A)$ are non-empty sets, called the set of complete traces and partial traces over A, respectively. Notice that $\mathcal{B}_C(A)$ and $\mathcal{B}_P(A)$ are not necessarily disjoint. Slightly abusing notation, we also write $\mathcal{B}_C$ and $\mathcal{B}_P$ as abbreviations:

$$\begin{aligned} \mathcal{B}_C &= \bigcup \{\mathcal{B}_C(A) : A \text{ is an alphabet}\} \\ \mathcal{B}_P &= \bigcup \{\mathcal{B}_P(A) : A \text{ is an alphabet}\}. \end{aligned}$$

We also write $\mathcal{B}(A)$ for $\mathcal{B}_C(A) \cup \mathcal{B}_P(A)$ and $\mathcal{B}$ for $\mathcal{B}_C \cup \mathcal{B}_P$. For every alphabet B over W and every renaming function r over W, $proj(B)$ and $rename(r)$ are partial functions from $\mathcal{B}$ to $\mathcal{B}$. The *concatenation* operation '$\cdot$' is a partial function from $\mathcal{B}_P \times \mathcal{B}$ to $\mathcal{B}$. The axioms T1 through T8 (see definition 2.7, p. 26) must be satisfied, with each instance of $\mathcal{B}_C$ replaced by $\mathcal{B}$ in the statement of these axioms. The following axioms T9 through T19 must also be satisfied.

T9. For every alphabet A, if $x \in \mathcal{B}_P(A)$ and $y \in \mathcal{B}(A)$, then $x \cdot y$ is defined and is an element of $\mathcal{B}(A)$. If there is no alphabet A such that $x \in \mathcal{B}_P(A)$ and $y \in \mathcal{B}(A)$, then $x \cdot y$ is undefined.

T10. If $x \cdot y$ is defined and is an element of $\mathcal{B}(A)$, then

$$\begin{aligned} y \in \mathcal{B}_C(A) &\Leftrightarrow x \cdot y \in \mathcal{B}_C(A) \\ y \in \mathcal{B}_P(A) &\Leftrightarrow x \cdot y \in \mathcal{B}_P(A). \end{aligned}$$

T11. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

T12. If $x \cdot y = x \cdot y'$, then $y = y'$.

T13. For every alphabet A, there exists a distinguished element ϵ_A of $\mathcal{B}_P(A)$ such that $x \cdot \epsilon_A = x$ for all $x \in \mathcal{B}_P(A)$ and $\epsilon_A \cdot y = y$ for all $y \in \mathcal{B}(A)$. Also, if $x \cdot y = \epsilon_A$, then $x = \epsilon_A$ and $y = \epsilon_A$.

T14. If $x \cdot y = x' \cdot y'$, then there exists $z, z' \in \mathcal{B}_P$ and $z'' \in \mathcal{B}$ such that $x \cdot z = x' \cdot z'$ and $z \cdot z'' = y$.

T15. If $z, z' \in \mathcal{B}(A)$ and $z \neq z'$, then there exists x and y such that

$$x \cdot y = z \wedge \forall y'[x \cdot y' \neq z'] \quad \text{or} \quad x \cdot y = z' \wedge \forall y'[x \cdot y' \neq z].$$

T16. If $x \in \mathcal{B}(A)$ and $proj(B)(x)$ is defined, then

$$x \in \mathcal{B}_C(A) \Leftrightarrow proj(B)(x) \in \mathcal{B}_C(B)$$
$$x \in \mathcal{B}_P(A) \Rightarrow proj(B)(x) \in \mathcal{B}_P(B).$$

T17. For all x, y and z', $x \cdot y = proj(B)(z')$ iff there exists x' and y' such that $x = proj(B)(x')$, $y = proj(B)(y')$ and $x' \cdot y' = z'$.

T18. If $rename(r)(x)$ is defined, then

$$x \in \mathcal{B}_C(dom(r)) \Leftrightarrow rename(r)(x) \in \mathcal{B}_C(codom(r))$$
$$x \in \mathcal{B}_P(dom(r)) \Leftrightarrow rename(r)(x) \in \mathcal{B}_P(codom(r)).$$

T19. $rename(r)(x \cdot y) = rename(r)(x) \cdot rename(r)(y)$.

T9 states when concatenation is defined. T10, T16 and T18 state when the results of an operation are partial or complete traces. Notice that if $proj(B)(x)$ is a partial trace, then x need not be a partial trace. For example, this can happen x is an infinite sequence and $B = \emptyset$.

The trace $x \cdot y$ represents the execution of x followed by the execution of y. Given this interpretation of concatenation, it is clear that concatenation should be associative, as required by T11. T12 states that if two behaviors differ for some suffix, then they are different behaviors. For every alphabet A, T13 requires the existence of a trace ϵ_A that is analogous to the empty string for formal languages.

Note 4.21. We often write ϵ instead of ϵ_A when A is clear from context or ϵ_A is independent of A.

T14 says that if x and x' are both prefixes of some trace w, then there exists some partial trace w' that is a prefix of w such that x and x' are both prefixes of w'. T15 says that for any two distinct traces z and z', there must exist a trace x such that x is a prefix of z but not of z', or a prefix of z' but not of z.

The reverse implication of T17 is equivalent to requiring that projection distribute over concatenation. The forward implication can be interpreted as follows. Assume the trace $proj(B)(z')$ can be split into the pieces x followed by y, *i.e.*, $x \cdot y = proj(B)(z')$. Then the trace z' can be split into pieces x' and y' such that $x = proj(B)(x')$ and $y = proj(B)(y')$.

It is natural for renaming to distribute over concatenation, as required by T19.

Note 4.22. We naturally extend the concatenation operation on traces to an operation on sets of traces.

As an example trace algebra with partial traces, we construct $\mathcal{C}^I$, which is an extension of the trace algebra (without partial traces) $\mathcal{C}^I_C$ formalized in definition 2.9. As with $\mathcal{C}^I_C$, the superscript I is a mnemonic for an (untimed) interleaving model. The proof that $\mathcal{C}^I$ is a trace algebra is delayed until lemma 4.33 (p. 91).

Definition 4.23. We define the trace algebra with partial traces $\mathcal{C}^I$ as follows:

- The set $\mathcal{B}^I_C(A)$ of complete traces over an alphabet A is A^∞ (notice that this definition of $\mathcal{B}^I_C(A)$ is consistent with the definition of $\mathcal{B}^I_C(A)$ given for $\mathcal{C}^I_C$).
- The set $\mathcal{B}^I_P(A)$ of partial traces over an alphabet A is A^*.
- The projection and renaming operations are the same as in $\mathcal{C}^I_C$.
- The concatenation operation is standard concatenation of sequences.

Similarly, $\mathcal{C}^{QTI\varphi}_C$ can be extended to include partial traces. Notice in the definition below that the set of partial traces over an alphabet A is not $(A \cup \{\varphi\})^*$; non-empty sequences must end with φ. This is related to the fact that partial traces in a discrete time model must represent a time period that is an integer number of time units long.

Definition 4.24. We define the trace algebra with partial traces $\mathcal{C}^{QTI\varphi}$ as follows:

- The set $\mathcal{B}_C(A)$ of traces over an alphabet A is the set of $x \in (A \cup \{\varphi\})^\omega$ such that φ appears infinitely often in x.

- The set $\mathcal{B}_P(A)$ of partial traces over an alphabet A is

  $$\epsilon + (A \cup \{\varphi\})^*(\{\varphi\}),$$

  that is, the empty sequence and all finite sequences over $A \cup \{\varphi\}$ that end with φ.
- If $x \in \mathcal{B}(A)$ and $B \subseteq A$, then $proj(B)(x)$ is the sequence formed from x by removing every symbol a not in $B \cup \{\varphi\}$. More formally,

  $$proj^{QTI\varphi}(B)(x) = proj^{I}(B \cup \{\varphi\})(x).$$
- If $x \in \mathcal{B}(A)$ and r is a renaming function with domain A, then $rename(r)(x)$ is the sequence formed from x by replacing every $a \in A$ with $r(a)$.
- The concatenation operation is standard concatenation of sequences.

Given a trace algebra with complete traces, there are several related trace algebras that we can define, as follows.

Definition 4.25. Given a trace algebra $\mathcal{C} = (\mathcal{B}_C, \mathcal{B}_P, proj, rename, \cdot)$, we use the subscripts C, P and PC to denote the trace algebras

$$\begin{aligned} \mathcal{C}_C &= (\mathcal{B}_C, proj, rename) \\ \mathcal{C}_P &= (\mathcal{B}_P, \mathcal{B}_P, proj, rename, \cdot) \\ \mathcal{C}_{PC} &= (\mathcal{B}_P, proj, rename). \end{aligned}$$

Lemma 4.37 (p. 92) proves that $\mathcal{C}_C$, $\mathcal{C}_P$ and $\mathcal{C}_{PC}$ satisfy the appropriate axioms of trace algebra.

Typically, $\mathcal{C}_C$ is used when only complete traces are of interest. We have already seen an example of this notation with the trace algebras $\mathcal{C}^I$ and $\mathcal{C}_C^I$. The algebras $\mathcal{C}_P$ and $\mathcal{C}_{PC}$ are used when restricting to safety properties; using only traces in $\mathcal{B}_P$ is analogous to using only finite sequences, without infinite sequences, to represent behaviors.

We can use the concatenation operation to define *suffixes* and *prefixes*.

Definition 4.26. Let $x \in \mathcal{B}_P(A)$ and $Z \subseteq \mathcal{B}(A)$. The functions $suf(x, Z)$, $pref(Z)$ and $suf(Z)$ are given by

$$\begin{aligned} suf(x, Z) &= \{y \in \mathcal{B}(A) : x \cdot y \in Z\} \\ pref(Z) &= \{x \in \mathcal{B}_P(A) : suf(x, Z) \neq \emptyset\} \\ suf(Z) &= \bigcup_{x \in pref(Z)} suf(x, Z). \end{aligned}$$

Definition 4.27. We say X is *prefix-closed* iff $pref(X) \subseteq X$.

Note 4.28. If $x \in \mathcal{B}$, we sometimes write $pref(x)$ to denote $pref(\{x\})$. Similarly for $suf(x)$.

The remainder of this section proves the claims made above, and proves some additional results about trace algebras with partial traces. It may be skipped on first reading.

We begin with some simply corollaries that follow immediately from the axioms of trace algebra.

Corollary 4.29.

1. By T17, $proj(B)(x) \cdot proj(B)(y) = proj(B)(x \cdot y)$.
2. By T13 and T15 (with $z' = \epsilon_A$), if $z \in \mathcal{B}(A)$ is not equal to ϵ_A, then there exists x and y such that $x \cdot y = z$ and $x \neq \epsilon_A$.

We also prove some simple corollaries related to suffixes and prefixes.

Corollary 4.30.

$$\begin{aligned} suf(x, X \cup Y) &= suf(x, X) \cup suf(x, Y), \\ pref(X \cup Y) &= pref(X) \cup pref(Y), \\ suf(X \cup Y) &= suf(X) \cup suf(Y). \end{aligned}$$

Corollary 4.31. If $X \subseteq Y$, then

$$\begin{aligned} suf(x, X) &\subseteq suf(x, Y), \\ pref(X) &\subseteq pref(Y), \\ suf(X) &\subseteq suf(Y). \end{aligned}$$

Corollary 4.32.

1. By T13, if $Z \neq \emptyset$, then $\epsilon \in pref(Z)$.
2. By T13, $(Z \cap \mathcal{B}_P) \subseteq pref(Z)$.
3. By item 1, $Z \subseteq suf(Z)$.
4. By T11, $suf(x, suf(y, X)) = suf(y \cdot x, X)$.

5. By T6, T7 and T19, $rename(r)(suf(x,X)) = suf(rename(r)(x), rename(r)(X))$.

We must also prove our claim that $\mathcal{C}^I$ is a trace algebra.

Lemma 4.33. $\mathcal{C}^I$ is a trace algebra with partial traces.

Proof. To show that $\mathcal{C}^I$ is a trace algebra with partial traces, we must show that it satisfies T1 through T19. Axioms T1 through T8 hold since $\mathcal{C}^I_C$ is a trace algebra without partial traces and $\mathcal{B}^I = \mathcal{B}^I_C$. T9, T10, T16 and T18 are easy to verify. T11 and T12 are a basic properties of concatenation of finite and infinite sequences. The ϵ_A of T13 is just the empty sequence ϵ, for every alphabet A. T19 is also easy to show. All that remains is T14, T15 and T17.

Lemma 4.34. $\mathcal{C}^I$ satisfies T14.

Proof. Assume $x \cdot y = x' \cdot y'$; we must show that there exists partial traces z and z' such that $x \cdot z = x' \cdot z'$. There is no loss of generality in assuming that $len(x') \leq len(x)$. Let $z = \epsilon$. By our assumptions, the length of y' must be at least $len(x) - len(x')$. Let z' be the prefix of y' that has length $len(x) - len(x')$. Both z' and z are of finite length, so they are partial traces. It is easy to check that $x' \cdot z' = x \cdot z$. Let $z'' = y$; clearly $z \cdot z'' = y$.

□

Lemma 4.35. $\mathcal{C}^I$ satisfies T15.

Proof. Assume z and z' are distinct elements of $\mathcal{B}(A)$. If z and z' have different lengths, then there is no loss of generality in assuming that z' is the shorter of the two. In this case, z' must have finite length, say n, and z must have a length of at least $n+1$. T15 is satisfied by letting x be the length $n+1$ prefix of z.

If z and z' have the same length n, then there must exist a $k < n$ such that $z(k) \neq z'(k)$. T15 is satisfied by letting x be the length $k+1$ prefix of either z or z'.

□

Lemma 4.36. $\mathcal{C}^I$ satisfies T17.

Proof. The reverse implication of T17 is equivalent to

$$proj(B)(x' \cdot y') = proj(B)(x') \cdot proj(B)(y'),$$

which follows easily from the definition of *proj*.

To prove the forward implication, let $B \subseteq A$. We consider the case where z' is a finite length sequence; the generalization to the infinite case is straightforward. There is no loss of generality in assuming that

$$z' = x'_0 b_0 x'_1 b_1 \cdots x'_{n-1} b_{n-1} x'_n,$$

where $b_i \in B$ and $x'_i \in (A - B)^*$. If $x \cdot y = proj(B)(z')$, then there must exist a k such that

$$x = b_0 \cdots b_{k-1} \quad \text{and} \quad y = b_k \cdots b_{n-1}.$$

Therefore, it is sufficient to let x' and y' be such that

$$x' = x'_0 b_0 \cdots x'_{k-1} b_{k-1}$$

$$y' = x'_k b_k \cdots x'_{n-1} b_{n-1} x'_n.$$

□

□

Next we prove the claim made in definition 4.25 about the existence of trace algebras $\mathcal{C}_C$, $\mathcal{C}_P$ and $\mathcal{C}_{PC}$, given a trace algebra with complete traces $\mathcal{C}$.

Lemma 4.37. If $\mathcal{C} = (\mathcal{B}_C, \mathcal{B}_P, proj, rename, \cdot)$ is a trace algebra with partial traces, then

$$\begin{aligned} \mathcal{C}_C &= (\mathcal{B}_C, proj, rename) \\ \mathcal{C}_P &= (\mathcal{B}_P, \mathcal{B}_P, proj, rename, \cdot) \\ \mathcal{C}_{PC} &= (\mathcal{B}_P, proj, rename), \end{aligned}$$

are trace algebras.

Proof.

Lemma 4.38. $\mathcal{C}_C$ is a trace algebra without partial traces.

Proof. Since $\mathcal{C}$ satisfies T16 and T18, $\mathcal{C}_C$ is closed under projection and renaming. We must show that $\mathcal{C}_C$ satisfies T1 through T8. Except for T4, all of these axioms obviously remain true when traces are removed from the domain. To prove T4, let $x \in \mathcal{B}_C(A)$ and $x' \in \mathcal{B}_C(A')$ be such that $proj(A \cap A')(x) = proj(A \cap A')(x')$, and let A'' be an alphabet such that $A \cup A' \subseteq A''$. Since $\mathcal{C}$ satisfies T4, there exists $x'' \in \mathcal{B}_C(A'') \cup \mathcal{B}_P(A'')$ such that $x = proj(A)(x'')$ and $x' = proj(A')(x'')$. We must show that $x'' \in \mathcal{B}_C(A'')$, which is true since $x \in \mathcal{B}_C(A)$, $x = proj(A)(x'')$ and $\mathcal{C}$ satisfies T16.

□

Next we must show that $\mathcal{C}_P$ is a trace algebra with partial traces. Since $\mathcal{C}$ satisfies T10, T16 and T18, $\mathcal{C}_P$ is closed under concatenation, projection and renaming. We must show that $\mathcal{C}_P$ satisfies T1 through T19. The next lemma shows that $\mathcal{C}_P$ satisfies T4: the remaining axioms are handled in a later lemma.

Lemma 4.39. $\mathcal{C}_P$ satisfies T4.

Proof. To prove T4, let $x_0 \in \mathcal{B}_P(A)$ and $x_0' \in \mathcal{B}_P(A')$ be such that $proj(A \cap A')(x_0) = proj(A \cap A')(x_0')$, and let A'' be an alphabet such that $A \cup A' \subseteq A''$. We must show that there exists $x_0'' \in \mathcal{B}_P(A'')$ such that $x_0 = proj(A)(x_0'')$ and $x_0' = proj(A')(x_0'')$.

Since $\mathcal{C}$ satisfies T4, there exists $w'' \in \mathcal{B}(A'')$ such that $x_0 = proj(A)(w'')$ and $x_0' = proj(A')(w'')$.

Since $\mathcal{C}$ satisfies T13, $x_0 \cdot \epsilon_A = proj(A)(w'')$ and $x_0' \cdot \epsilon_A = proj(A')(w'')$. Since $\mathcal{C}$ satisfies T17, there exists $x_1 \in \mathcal{B}_P(A'')$ and $y_1 \in \mathcal{B}_C(A'') \cup \mathcal{B}_P(A'')$ such that $x_0 = proj(A)(x_1)$, $\epsilon_A = proj(A)(y_1)$ and $x_1 \cdot y_1 = w''$. Similarly, there exists $x_1' \in \mathcal{B}_P(A'')$ and $y_1' \in \mathcal{B}_C(A'') \cup \mathcal{B}_P(A'')$ such that $x_0' = proj(A')(x_1')$, $\epsilon_{A'} = proj(A')(y_1')$ and $x_1' \cdot y_1' = w''$.

Notice that $x_1 \cdot y_1 = x_1' \cdot y_1'$. Since $\mathcal{C}$ satisfies T14, there exists $z_1, z_1' \in \mathcal{B}_P(A'')$ and $z'' \in \mathcal{B}(A)$ such that $x_1 \cdot z_1 = x_1' \cdot z_1'$ and $z_1 \cdot z_1'' = y_1$. Notice,

$$
\begin{aligned}
z_1 \cdot z_1'' = y_1 \quad &\Leftrightarrow \quad x_1 \cdot (z_1 \cdot z_1'') = x_1 \cdot y_1 \\
&\text{since } \mathcal{C} \text{ satisfies T11} \\
&\Leftrightarrow \quad (x_1 \cdot z_1) \cdot z_1'' = x_1 \cdot y_1 \\
&\text{since } x_1 \cdot y_1 = x_1' \cdot y_1' \text{ and } x_1 \cdot z_1 = x_1' \cdot z_1'
\end{aligned}
$$

$$\Leftrightarrow \quad (x_1' \cdot z_1') \cdot z_1'' = x_1' \cdot y_1'$$

since $\mathcal{C}$ satisfies T11

$$\Leftrightarrow \quad x_1' \cdot (z_1' \cdot z_1'') = x_1' \cdot y_1'$$

since $\mathcal{C}$ satisfies T12

$$\Leftrightarrow \quad z_1' \cdot z_1'' = y_1'$$

Also,

$$z_1 \cdot z_1'' = y_1 \quad \Rightarrow \quad proj(A)(z_1 \cdot z_1'') = proj(A)(y_1)$$

since $proj(A)(y_1) = \epsilon_A$

$$\Leftrightarrow \quad proj(A)(z_1 \cdot z_1'') = \epsilon_A$$

since $\mathcal{C}$ satisfies T17

$$\Leftrightarrow \quad proj(A)(z_1) \cdot proj(A)(z_1'') = \epsilon_A$$

since $\mathcal{C}$ satisfies T13

$$\Leftrightarrow \quad proj(A)(z_1) = \epsilon_A.$$

Similarly, $proj(A')(z_1') = \epsilon_{A'}$.

Let $x_0'' = x_1 \cdot z_1$; notice that $x'' \in \mathcal{B}_P(A'')$.

$$proj(A)(x_0'') = proj(A)(x_1 \cdot z_1)$$

since $\mathcal{C}$ satisfies T17

$$= proj(A)(x_1) \cdot proj(A)(z_1))$$

since $x_0 = proj(A)(x_1)$ and $\epsilon_A = proj(A)(z_1)$

$$= x_0 \cdot \epsilon_A$$

since $\mathcal{C}$ satisfies T13

$$= x_0.$$

Similarly, $proj(A')(x_0'') = x_0'$. Therefore, x_0'' has the properties needed to show that $\mathcal{C}_P$ satisfies T4, since $x_0'' \in \mathcal{B}_P(A'')$, $x_0 = proj(A)(x_0'')$ and $x_0' = proj(A')(x_0'')$.

□

Lemma 4.40. $\mathcal{C}_P$ is a trace algebra with partial traces.

Proof. As mentioned earlier, $\mathcal{C}_P$ is closed under concatenation, projection and renaming. We must show that $\mathcal{C}_P$ satisfies T1 through T19. Clearly T1, T2 and T3 remain true when traces are removed from the domain. The previous lemma showed that $\mathcal{C}_P$ satisfies T4.

Clearly T5, T6, T7 and T8 remain true when traces are removed from the domain. To prove T9, we must show that for all x, y in $\mathcal{B}_P$, $x \cdot y$ is defined iff there exists an alphabet such that $x \in \mathcal{B}_P(A)$ and $y \in \mathcal{B}_P(A)$. This follows since $\mathcal{C}$ satisfies T9. Since $\mathcal{C}$ satisfies T9 and T10, both sides of both iff's in T10 for $\mathcal{C}_P$ are identically true, so T10 holds.

Clearly T11, T12, T13 and T14 remain true. Since $\mathcal{C}$ satisfies T10, the x and y in T15 must be elements of $\mathcal{B}_P$, so T15 remains true.

Since $\mathcal{C}$ satisfies T1 and T16, both sides of the iff and the implication in T16 for $\mathcal{C}_P$ are identically true, so T16 holds. Since $\mathcal{C}$ satisfies T10, the x' and y' in T17 must be an elements of $\mathcal{B}_P$, so T17 remains true.

Since $\mathcal{C}$ satisfies T5 and T18, both sides of both iff's in T18 for $\mathcal{C}_P$ are identically true, so T18 holds. Clearly T19 remains true.

□

Lemma 4.41. $\mathcal{C}_{PC}$ is a trace algebra without partial traces.

Proof. Let $\mathcal{C}' = \mathcal{C}_P$. By the previous two lemmas, $\mathcal{C}'$ and $\mathcal{C}'_C$ are trace algebras. Therefore, $\mathcal{C}_{PC}$ is a trace algebra, since $\mathcal{C}_{PC} = \mathcal{C}'_C$.

□

□

The final result of this section shows that traces can be characterized by their set of prefixes. We will use this result when restricting models to represent only safety properties.

Theorem 4.42. For some trace algebra $\mathcal{C} = (\mathcal{B}_C, \mathcal{B}_P, proj, rename, \cdot)$ and some alphabet A, let z and z' be elements of $\mathcal{B}(A)$. Then

$$z = z' \quad \Leftrightarrow \quad pref(z) = pref(z').$$

Proof. The forward implication is obvious. To prove the reverse implication, assume that z and z' are distinct elements of $\mathcal{B}(A)$. By T15, there exists x and y such that

$$x \cdot y = z \wedge \forall y'[x \cdot y' \neq z'] \quad \text{or} \quad x \cdot y = z' \wedge \forall y'[x \cdot y' \neq z].$$

Therefore,

$$x \in \mathit{pref}(z) \wedge x \notin \mathit{pref}(z') \quad \text{or} \quad x \in \mathit{pref}(z') \wedge x \notin \mathit{pref}(z).$$

□

4.3.2 Restricting to Safety Properties

It is common to restrict a verification technique to handle only safety properties, since this can be computationally more efficient than handling full liveness properties. If traces are sequences, then this is just a matter of restricting to prefix-closed trace structures with only finite sequences. We generalize this idea to arbitrary traces, as follows.

Definition 4.43. Given a trace algebra $\mathcal{C} = (\mathcal{B}_C, \mathcal{B}_P, \mathit{proj}, \mathit{rename}, \cdot)$, we use the subscript PC to denote the trace structure algebra,

$$\mathcal{A}_{PC} = (\mathcal{C}_{PC}, \mathcal{T}),$$

where $\mathcal{T}$ is the set of all prefix-closed trace structures over $\mathcal{C}_{PC}$ (see def. 4.25, p. 89). A trace structure $T = (\gamma, P)$ over $\mathcal{C}_{PC}$ is prefix-closed iff $\mathit{pref}(P) \subseteq P$. Lemma 4.44 proves that $\mathcal{A}_{PC}$ is a trace structure algebra.

For an arbitrary trace algebra $\mathcal{C}$ with partial traces, it is possible to construct a conservative approximation from trace structures over $\mathcal{C}_C$ to $\mathcal{A}_{PC}$. We do this by using an isomorphism based on identifying a single trace in $\mathcal{C}_C$ with its set of prefixes (each prefix is a trace in $\mathcal{C}_{PC}$). The result is a power set algebra over $\mathcal{C}_{PC}$ that is isomorphic to $\mathcal{C}_C$, which can be used to construct a conservative approximation induced by a power set algebra. The approximation is only useful for verification if the specification does not include any liveness properties; otherwise a false negative will result (assuming the implementation satisfies its specification).

The remainder of this section proves these claims.

Lemma 4.44. If $\mathcal{C}$ is a trace algebra with partial traces, then $\mathcal{A}_{PC}$ (as in def. 4.43) is a trace structure algebra.

Proof. Let $\mathcal{T}'$ be the set of all trace structures over $\mathcal{C}_{PC}$. By theorem 2.27, $(\mathcal{C}_{PC}, \mathcal{T}')$ is a trace structure algebra.

For all alphabets B, let $\mathcal{L}(B)$ be the class of all prefix-closed sets of traces of $\mathcal{B}_P(B)$. Let

$$\mathcal{T}'' = \{T \in \mathcal{T}' : P \in \mathcal{L}(A)\}.$$

It is easy to check that $\mathcal{L}(B)$ satisfies L1 through L4. Thus, by theorem 2.30. since $(\mathcal{C}_{PC}, \mathcal{T}')$ is a trace structure algebra, so is $(\mathcal{C}_{PC}, \mathcal{T}'')$. Notice that $\mathcal{T}''$ is equal to $\mathcal{T}$. Therefore, $\mathcal{A}_{PC} = (\mathcal{C}_{PC}, \mathcal{T})$ is a trace structure algebra.

□

Definition 4.45. Given a trace algebra $\mathcal{C} = (\mathcal{B}_C, \mathcal{B}_P, proj, rename, \cdot)$, we use the subscript C/P to denote the trace algebra,

$$\mathcal{C}_{C/P} = (\mathcal{B}_{C/P}, proj, rename),$$

where

$$\mathcal{B}_{C/P}(A) = \{pref(x) : x \in \mathcal{B}_C(A)\},$$

and *proj* and *rename* are naturally extended to sets of traces. Lemma 4.46 (below) proves that $\mathcal{C}_{C/P}$ is a trace algebra and is isomorphic to $\mathcal{C}_C$.

Notice that $\mathcal{C}_{C/P}$ is a power set algebra (definition 4.1) over $\mathcal{C}_{PC}$.

Lemma 4.46. If $\mathcal{C}$ is a trace algebra with partial traces, then $\mathcal{C}_{C/P}$ (as in def. 4.45) is a trace algebra. Also, $\lambda x[pref(\{x\})]$ is an isomorphism from $\mathcal{C}_C$ to $\mathcal{C}_{C/P}$.

Proof. By theorem 4.42, the function $\lambda x[pref(\{x\})]$ is an alphabet preserving bijection from $\mathcal{B}_C$ to $\mathcal{B}_{C/P}$. All that remains is to show that it satisfies the homomorphism laws for *proj* and *rename*.

$$\begin{aligned} & pref(\{proj(B)(z')\}) \\ &= \{x : \exists y[x \cdot y = proj(B)(z')]\} \end{aligned}$$

since $\mathcal{C}$ satisfies T17

$= \{x : \exists y[\exists x', y'[$

$x' \cdot y' = z' \wedge x = proj(B)(x') \wedge y = proj(B)(y')]]\}$

since $\exists y[y = proj(B)(y')]$ for all $y' \in \mathcal{B}_C(B)$

$= \{x : \exists x', y'[x' \cdot y' = z' \wedge x = proj(B)(x')]\}$

by definition of the natural extension of $proj(B)$

$= proj(B)(\{x' : \exists y'[x' \cdot y' = z']\})$

$= proj(B)(pref(\{z'\}))$.

Also,

$pref(\{rename(r)(z')\})$

$= \{x : \exists y[x \cdot y = rename(r)(z')]\}$

since $\mathcal{C}$ satisfies T6 and T7

$= \{x : \exists y[rename(r^{-1})(x \cdot y) = z']\}$

since $\mathcal{C}$ satisfies T19

$= \{x : \exists y[rename(r^{-1})(x) \cdot rename(r^{-1})(y) = z']\}$

since $\mathcal{C}$ satisfies T6 and T7

$= \{rename(r)(x') : \exists y'[x' \cdot y' = z']\}$

$= rename(r)(pref(\{z'\}))$.

□

Now we can construct a conservative approximation from trace structures over $\mathcal{C}_C$ to trace structures over $\mathcal{C}_{PC}$ by using $\mathcal{C}_{C/P}$, which is isomorphic to $\mathcal{C}_C$ and is a power set algebra over $\mathcal{C}_{PC}$. The upper bound $\Psi_u(T)$ is simply the result of composing the isomorphism with the upper bound of the standard conservative approximation induced by $\mathcal{C}_{C/P}$. The lower bound $\Psi_l(T)$ is equal to $\Psi_u(T)$ when T has no liveness properties; otherwise, it is equal to the empty trace structure.

Theorem 4.47. Let $\mathcal{C}$ be a trace algebra with partial traces, and let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T}_C)$ be a trace structure algebra, where $\mathcal{T}_C$ is the set of all trace structures over $\mathcal{C}_C$. Let Ψ_u and Ψ_l be functions from trace structures $T = (\gamma, P)$ in $\mathcal{A}_C$ to trace structures in $\mathcal{A}_{PC}$ such

that $\Psi_u(T) = (\gamma, P'_u)$ and $\Psi_l(T) = (\gamma, P'_l)$, where

$$\begin{aligned} P'_u &= pref(P) \\ P'_l &= \begin{cases} pref(P), & \text{if } [x \in \mathcal{B}_C(A) \wedge pref(\{x\}) \subseteq pref(P)] \Rightarrow x \in P; \\ \emptyset & \text{otherwise.} \end{cases} \end{aligned}$$

Then $\Psi = (\Psi_l, \Psi_u)$ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}_{PC}$.

Proof. Let $\mathcal{A}_{C/P} = (\mathcal{C}_{C/P}, \mathcal{T}_{C/P})$ be a trace structure algebra, where $\mathcal{T}_{C/P}$ is the set of all trace structures over $\mathcal{C}_{C/P}$. By lemma 4.46, $\mathcal{C}_{C/P}$ is isomorphic to $\mathcal{C}_C$, so by corollary 2.40, $\mathcal{A}_C$ is isomorphic to $\mathcal{A}_{C/P}$. The isomorphism from $\mathcal{A}_C$ to $\mathcal{A}_{C/P}$ is the function H such that

$$H((\gamma, P)) = (\gamma, \{pref(\{x\}) : x \in P\}).$$

We show that Ψ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}_{PC}$ by first constructing a conservative approximation Ψ' from $\mathcal{A}_{C/P}$ to $\mathcal{A}_{PC}$, and then showing that Ψ is equal to Ψ' composed with H.

Let Ψ'_u and Ψ'_l be functions from trace structures $T = (\gamma, P)$ in $\mathcal{A}_{C/P}$ to trace structures in $\mathcal{A}_{PC}$ such that $\Psi_u(T) = (\gamma, P'_u)$ and $\Psi_l(T) = (\gamma, P'_l)$, where

$$\begin{aligned} P'_u &= \cup P \\ P'_l &= \begin{cases} \cup P, & \text{if } [x \in \mathcal{B}_C(A) \wedge x \subseteq \cup P] \Rightarrow x \in P; \\ \emptyset & \text{otherwise.} \end{cases} \end{aligned}$$

It is easy to check that $\Psi' = (\Psi'_l, \Psi'_u)$ is a conservative approximation from $\mathcal{A}_{C/P}$ to $\mathcal{A}_{PC}$ induced by the power set algebra $\mathcal{C}_{C/P}$. It is also easy to check that

$$\begin{aligned} \Psi_u(T) &= \Psi'_u(H(T)) \\ \Psi_l(T) &= \Psi'_l(H(T)), \end{aligned}$$

where H is the isomorphism from $\mathcal{A}_C$ to $\mathcal{A}_{C/P}$ described above. Thus, Ψ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}_{PC}$.

□

4.3.3 Trace Structure Algebra with Partial Traces

If T is a trace structure and $x \in pref(P)$, then x represents a partial behavior that is a prefix of some complete behavior of T. After T executes x, we might say that T has changed to a different state. It is often useful to think of each state of an agent as being a different agent [76]. With this in mind, we might say that T becomes a different agent after executing x. We define the function *suf* on trace structures so that we can write $suf(x, T)$ to denote the agent that T becomes as a result of executing x.

Definition 4.48. If $\mathcal{C} = (\mathcal{B}_C, \mathcal{B}_P, proj, rename, \cdot)$ is a trace algebra with partial traces and $\mathcal{T}$ is a subset of the trace structures of $\mathcal{C}_C$ (recall that $\mathcal{C}_C = (\mathcal{B}_C, proj, rename)$, as described in definition 4.25), then $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ is a *trace structure algebra with partial traces* iff the domain $\mathcal{T}$ is closed under the following operations on trace structures: parallel composition (def. 2.18), renaming (def. 2.20), projection (def. 2.19) and suffixing (def. 4.49).

For trace structure algebras with partial traces, the operations of parallel composition, renaming and projection on trace structures are defined exactly the same as they were for trace structure algebras without partial traces. Thus, they form a concurrency algebra.

Definition 4.49. $suf(x, T) = (\gamma, suf(x, P))$, where $x \in pref(P)$.

The operation of suffixing is clearly monotonic with respect to trace structure containment. Also, the following propositions involving suffixing are satisfied:

$$suf(\epsilon_A, T) = T$$

$$suf(x.\, suf(y, T)) = suf(y \cdot x, T)$$

$$suf(x, T \parallel T') = suf(proj(A)(x), T) \parallel suf(proj(A')(x), T')$$

$$proj(B)(suf(x, T)) \subseteq suf(proj(B)(x), proj(B)(T))$$

$$rename(r)(suf(x, T)) = suf(rename(r)(x), rename(r)(T)).$$

The remainder of this section proves these results.

Theorem 4.50. If T and T' are trace structures, then

$$suf(\epsilon_A, T) = T$$

$$suf(x, suf(y, T) = suf(y \cdot x, T)$$

$$suf(x, T \parallel T') = suf(proj(A)(x), T) \parallel suf(proj(A')(x), T')$$

$$proj(B)(suf(x, T)) \subseteq suf(proj(B)(x), proj(B)(T))$$

$$rename(r)(suf(x, T)) = suf(rename(r)(x), rename(r)(T)).$$

In all of the relationships, there is an implicit assumption that the left hand side of the equation or inequality is defined.

Proof. The first identity follows easily from T13 and the second follows from corollary 4.32.4. The remaining propositions are proved in the following lemmas.

Lemma 4.51. If $x \in pref(P \cap P')$, then

$$suf(x, T \parallel T') = suf(proj(A)(x), T) \parallel suf(proj(A')(x), T').$$

Proof. Let $T_1 = suf(x, T \parallel T')$ and let

$$T_2 = suf(proj(A)(x), T) \parallel suf(proj(A')(x), T').$$

We must show that $P_1 = P_2$.

$$\begin{aligned} P_1 &= suf(x, \{y \in \mathcal{B}_C(A_1) : proj(A)(y) \in P \wedge proj(A')(y) \in P'\}) \\ &= \{z \in \mathcal{B}_C(A_1) : proj(A)(x \cdot z) \in P \wedge proj(A')(x \cdot z) \in P'\} \\ &\quad \text{by corollary 4.29.1} \\ &= \{z \in \mathcal{B}_C(A_1) : proj(A)(x) \cdot proj(A)(z) \in P \\ &\qquad \wedge proj(A')(x) \cdot proj(A')(z) \in P'\} \\ &= \{z \in \mathcal{B}_C(A_1) : proj(A)(z) \in suf(proj(A)(x), P) \\ &\qquad \wedge proj(A')(z) \in suf(proj(A')(x), P')\} \\ &= P_2. \end{aligned}$$

□

Lemma 4.52. If $x \in pref(P)$ and $B \subseteq A$, then

$$proj(B)(suf(x,T)) \subseteq suf(proj(B)(x), proj(B)(T)).$$

Proof. Let $T_1 = proj(B)(suf(x,T))$ and let

$$T_2 = suf(proj(B)(x), proj(B)(T)).$$

We must show that $P_1 \subseteq P_2$.

$$\begin{aligned}
P_1 &= proj(B)(suf(x,P)) \\
&= proj(B)(\{y : x \cdot y \in P\}) \\
&= \{proj(B)(y) : x \cdot y \in P\} \\
&\subseteq \{proj(B)(y) : proj(B)(x \cdot y) \in proj(B)(P)\} \\
&\quad \text{by corollary 4.29.1} \\
&= \{proj(B)(y) : proj(B)(x) \cdot proj(B)(y) \in proj(B)(P)\} \\
&= \{proj(B)(y) : proj(B)(x) \cdot proj(B)(y) \in proj(B)(P)\} \\
&\subseteq \{y' : proj(B)(x) \cdot y' \in proj(B)(P)\} \\
&= suf(proj(B)(x), proj(B)(P)) \\
&= P_2.
\end{aligned}$$

□

Lemma 4.53. If $x \in pref(P)$, then

$$rename(r)(suf(x,T)) = suf(rename(r)(x), rename(r)(T)).$$

Proof. Let $T_1 = rename(r)(suf(x,T))$ and let

$$T_2 = suf(rename(r)(x), rename(r)(T)).$$

We must show that $P_1 = P_2$.

$$\begin{aligned}
P_1 &= rename(r)(suf(x,P)) \\
&= \{rename(r)(y) : x \cdot y \in P\} \\
&\quad \text{by T6 and T7} \\
&= \{rename(r)(y) : rename(r)(x \cdot y) \in rename(r)(P)\}
\end{aligned}$$

by T19

$= \{rename(r)(y) : rename(r)(x) \cdot rename(r)(y) \in rename(r)(P)\}$

by T6 and T7

$= \{z : rename(r)(x) \cdot z \in rename(r)(P)\}$

$= suf(rename(r)(x), rename(r)(P))$

$= P_2.$

□

□

4.3.4 Constructing Trace Structure Algebras with Partial Traces

The definition of a trace structure algebra with partial traces $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ requires that the set of trace structures $\mathcal{T}$ be closed under the operations on trace structures, including suffixing. This section proves three theorems that make it easier to prove closure, and shows how to use these theorems. The theorems are straightforward extensions of analogous results already proved for trace structure algebras without partial traces (section 2.3.3, p. 39).

The first theorem states that if $\mathcal{T}$ is equal to the set of all trace structures over $\mathcal{C}$, then $\mathcal{T}$ is closed under the operations on trace structures, so $\mathcal{A}$ is a trace structure algebra with partial traces; which is analogous to theorem 2.27. Recall that the alphabet of a trace structure need not be a finite set. The second theorem shows that trace structures with finite alphabets are closed under the operations on trace structures; which is analogous to theorem 2.28.

For the third theorem, let $(\mathcal{C}, \mathcal{T})$ be a trace structure algebra with partial traces, where $\mathcal{T}$ is some subset of the set of traces structures over $\mathcal{C}_C$. For every alphabet B, let $\mathcal{L}(B)$ be a class of sets of complete traces over B, that is, $\mathcal{L}(B) \subseteq 2^{\mathcal{B}c(B)}$. Assume that $\mathcal{L}$ is closed under intersection, renaming, projection, inverse projection and suffixing by prefixes (this is formalized below). Let $\mathcal{T}'$ be the set of trace structures $(\gamma, P) \in \mathcal{T}$ such that P is in $\mathcal{L}(A)$. Then $\mathcal{T}'$ is closed under the operations on trace structures, so $(\mathcal{C}, \mathcal{T}')$ is a trace structure algebra with partial traces. This is analogous to theorem 2.30.

Recall that $\mathcal{T}^I$ is the set of all trace structures over $\mathcal{C}_C^I$. By the first theorem, $\mathcal{A}^I = (\mathcal{C}^I, \mathcal{T}^I)$ is a trace structure algebra with partial traces. Recall that $\mathcal{T}^{IR}$ is the set of all trace structures (γ, P) over $\mathcal{C}_C^I$ for which γ has a finite alphabet and P is a mixed regular set of sequences (that

is, P is the union of a regular set and an ω-regular set). By the second and third theorems, $\mathcal{A}^{IR} = (\mathcal{C}^I, \mathcal{T}^{IR})$ is also a trace structure algebra with partial traces.

The remainder of this section formalizes these results.

Theorem 4.54. If $\mathcal{C}$ is a trace algebra and $\mathcal{T}$ is the set of all of the trace structures over $\mathcal{C}$, then $\mathcal{T}$ is closed under the operations on trace structures (parallel composition, projection, renaming and suffixing), so $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ is a trace structure algebra with partial traces.

Proof. Simple extension of theorem 2.27 (p. 39).

□

Theorem 4.55. Let $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ be a trace structure algebra with partial traces. Let $\mathcal{T}'$ be the set of trace structures $T \in \mathcal{T}$ such that the alphabet of T is a finite set. Then $\mathcal{A}' = (\mathcal{C}, \mathcal{T}')$ is a trace structure algebra with partial traces.

Proof. Simple extension of theorem 2.28 (p. 39).

□

Theorem 4.56. Let $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ be a trace structure algebra with partial traces. For every alphabet B of $\mathcal{T}$, let $\mathcal{L}(B)$ be a subset of $2^{Bc(B)}$. Let $\mathcal{T}'$ be the set of trace structures $T \in \mathcal{T}$ such that P is in $\mathcal{L}(A)$. Then $\mathcal{A}' = (\mathcal{C}, \mathcal{T}')$ is a trace structure algebra with partial traces if L1 through L5 are satisfied for every alphabet B of $\mathcal{T}$ (L1 through L4 are given on p. 40).

L5. If $X \in \mathcal{L}(B)$ and $x \in pref(X)$, then $suf(x, X) \in \mathcal{L}(B)$.

Proof. Simple extension of theorem 2.30 (p. 40).

□

Definition 4.57. We define $\mathcal{A}^I$ to be the ordered pair $(\mathcal{C}^I, \mathcal{T}^I)$; recall that $\mathcal{T}^I$ is the set of all trace structures over $\mathcal{C}^I_C$ (definition 2.31). By theorem 4.54, $\mathcal{A}^I$ is a trace structure algebra.

Definition 4.58. Recall that $\mathcal{T}^{IR}$ is the set of all trace structures $T = (\gamma, P)$ over $\mathcal{C}^I_C$ for which γ has a finite alphabet and P is a mixed regular set of sequences (definition 2.32). We define $\mathcal{A}^{IR}$ to be the ordered pair $(\mathcal{C}^I, \mathcal{T}^{IR})$. Showing that $\mathcal{A}^{IR}$ is a trace structure algebra with partial traces is a simple extension of the proof that $\mathcal{A}^{IR}_C$ is a trace structure algebra without partial traces (theorem 2.33).

4.4 Inverses of Conservative Approximations

Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. Let $T \in \mathcal{T}$ and $T' \in \mathcal{T}'$ be such that $T' = \Psi_u(T)$. As we have discussed, T' represents a kind of upper bound on T. It is natural to ask whether there is a trace structure in $\mathcal{T}$ that is represented exactly by T' rather than just being bounded by T'. If no trace structure in $\mathcal{T}$ can be represented exactly, then Ψ is abstracting away too much information to be of much use. If every trace structure in $\mathcal{T}$ can be represented exactly, then Ψ_l and Ψ_u are equal and are isomorphisms from $\mathcal{A}_C$ to $\mathcal{A}'_C$. These extreme cases illustrate that the amount of abstraction in Ψ is related to what trace structures T are represented exactly by $\Psi_u(T)$ and $\Psi_l(T)$.

To formalize what it means to be represented exactly in this context, we define the inverse of the conservative approximation Ψ. Normal notions of the inverse of a function are not adequate for this purpose, since Ψ is a pair of functions. We handle this by only considering those $T \in \mathcal{T}$ for which $\Psi_l(T)$ and $\Psi_u(T)$ have the same value, call it T'. Intuitively, T' represents T exactly in this case; the key property of the inverse of Ψ (written Ψ_{inv}) is that $\Psi_{inv}(T') = T$. If $\Psi_l(T) \neq \Psi_u(T)$, then T is not represented exactly in $\mathcal{A}'_C$. In this case, T is not in the image of Ψ_{inv}. Characterizing when $\Psi_{inv}(T')$ is defined (and what its value is) helps to show what trace structures in $\mathcal{T}$ can be represented exactly (not just conservatively) by trace structures in $\mathcal{T}'$. The remainder of this section formalizes the idea of the inverse of a conservative approximation, and characterizes the inverse of the tightest conservative approximation induced by a homomorphism h.

Lemma 4.59. Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. For every $T' \in \mathcal{T}'$, there is at most one $T \in \mathcal{T}$ such that $\Psi_l(T) = T'$ and $\Psi_u(T) = T'$.

Proof. The proof is by contradiction. Assume there exists two distinct T_1 and T_2 in $\mathcal{T}$ such that $\Psi_l(T_1)$, $\Psi_u(T_1)$, $\Psi_l(T_2)$ and $\Psi_u(T_2)$ are all equal to T'. This implies $\Psi_u(T_1) \subseteq \Psi_l(T_2)$ and $\Psi_u(T_2) \subseteq \Psi_l(T_1)$. Thus, by the definition of a conservative approximation, $T_1 \subseteq T_2$ and $T_2 \subseteq T_1$. Therefore, $T_1 = T_2$, which is a contradiction.

□

Definition 4.60. Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. Let $\mathcal{T}_1$ be the set of $T \in \mathcal{T}$ such that $\Psi_l(T) = \Psi_u(T)$. Let $\mathcal{T}'_1$ be the

image of $\mathcal{T}_1$ under Ψ_l. The *inverse of* Ψ is the partial function Ψ_{inv} with domain $\mathcal{T}'$ and codomain $\mathcal{T}$ that is defined for all $T' \in \mathcal{T}_1'$ so that $\Psi_{inv}(T') = T$, where T is the unique (by lemma 4.59 and the definition of $\mathcal{T}_1'$) trace structure such that $\Psi_l(T) = T'$ and $\Psi_u(T) = T'$.

Theorem 4.61. Let h be a trace algebra homomorphism from $\mathcal{C}_C$ to $\mathcal{C}_C'$, and let $\Psi = (\Psi_l, \Psi_u)$ be the tightest conservative approximation induced by h from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}_C' = (\mathcal{C}_C', \mathcal{T}')$. If $T' \in \mathcal{T}'$ is such that the set

$$Z = \{X \subseteq \mathcal{B}_C(A') : (\gamma', X) \in \mathcal{T} \wedge h(X) \subseteq P'\},$$

contains a unique maximal (by inclusion) element P for which $P' = h(P)$, then $\Psi_{inv}(T') = (\gamma', P)$; otherwise, $\Psi_{inv}(T')$ is undefined.

Proof. Let $T \in \mathcal{T}$ have the same signature γ as T', and let

$$Y = \bigcup\{X \subseteq \mathcal{B}_C(A) : (\gamma, X) \in \mathcal{T} \wedge h(X) \subseteq h(P)\}.$$

Notice that $P \subseteq Y$, since $T \in \mathcal{T}$. Consider the following sequence of logical equivalences:

$\Psi_{inv}(T') = T$

by the definition of the inverse of Ψ

$\Leftrightarrow \quad \Psi_u(T) = T' \ \wedge \ \Psi_l(T) = T'$

by the definition of Ψ

$\Leftrightarrow \quad h(P) = P' \ \wedge \ Y - P = \emptyset$

since $P \subseteq Y$

$\Leftrightarrow \quad h(P) = P' \ \wedge \ P = Y$

by the definition of Y and Z

$\Leftrightarrow \quad h(P) = P' \ \wedge \ P = \cup Z$

since $T \in \mathcal{T}$

$\Leftrightarrow \quad h(P) = P' \ \wedge \ P = \cup Z \ \wedge \ P \in Z,$

which is true iff Z contains a unique maximal element P for which $P' = h(P)$. The reverse implication of this equivalence implies the theorem for the case when $\Psi_{inv}(T')$ is

defined. By the forward implication, if Z does not contain a unique maximal element P for which $P' = h(P)$, then there does not exist $T \in \mathcal{T}$ such that $\Psi_{inv}(T') = T$, which implies that $\Psi_{inv}(T')$ is undefined.

□

The above theorem completely characterizes the inverse of any tightest conservative approximation induced by a homomorphism h. The final theorem of this section specializes this result to trace structures algebras that are *closed under finite and infinite unions,* a property enjoyed by many of the trace structure algebras we consider. This specialization results in a simpler characterization of when Ψ_{inv} is defined. In particular, $\Psi_{inv}(T')$ is defined iff there exists a $T \in \mathcal{T}$ such that $\Psi_u(T) = T'$. This is a strong result. Clearly the existence of such a T is a necessary condition for the inverse of any conservative approximation to be defined on T'; when $\mathcal{T}$ is closed under finite and infinite unions, and Ψ is the tightest conservative approximation induced by a homomorphism, it is also a sufficient condition.

Definition 4.62. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra. We say $\mathcal{A}_C$ is *closed under finite (infinite) unions* iff for every signature γ the set

$$\{P \subseteq \mathcal{B}_C(A) : (\gamma, P) \in \mathcal{T}\}$$

is closed under finite (infinite) unions.

Theorem 4.63. Let h be a trace algebra homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$, and let $\Psi = (\Psi_l, \Psi_u)$ be the tightest conservative approximation induced by h from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. Assume $\mathcal{A}_C$ is closed under finite and infinite unions. If $T' \in \mathcal{T}'$ is such that $\Psi_u(T) = T'$ for some $T \in \mathcal{T}$, then

$$\Psi_{inv}(T') = \bigcup\{X \subseteq \mathcal{B}_C(A') : (\gamma', X) \in \mathcal{T} \wedge h(X) \subseteq P'\};$$

otherwise, $\Psi_{inv}(T')$ is undefined.

Proof. By theorem 4.61, $\Psi_{inv}(T')$ is defined iff the set

$$Z = \{X \subseteq \mathcal{B}_C(A') : (\gamma', X) \in \mathcal{T} \wedge h(X) \subseteq P'\},$$

contains a unique maximal element P for which $P' = h(P)$. Since $\mathcal{A}_C$ is closed under finite and infinite unions, Z contains $\cup Z$, so this condition is equivalent to simply requiring that Z contain some element P for which $P' = h(P)$. By the definition of Ψ_u, this is equivalent to there exists $T \in \mathcal{T}$ such that $\Psi_u(T) = T'$. Also, when $\Psi_{inv}(T')$ is defined, it is clearly equal to $\cup Z$.

□

Chapter 5

Delay Models

Trace algebras and trace structure algebras are very general mathematical tools for constructing domains of agents models. Conservative approximations provide a general method for proving relationships between different domains of process models. However, developing domains of agent models is only part of the task of modeling and specifying real-time systems: it is also necessary to choose specific agents models to represent specifications and system components.

Finding a correct formal specification is known to often be quite difficult. However, the problem of finding good component models has received relatively little attention. For speed-dependent asynchronous circuits, finding good component models (often called gate models, in this case) is surprisingly subtle.

In this chapter, we consider several different delay models for verifying speed-dependent asynchronous circuits. From each delay model we produce a gate model by feeding the output of an ideal (delay-free) gate into a delay element of the appropriate type. The delay models are used in the verification two asynchronous FIFO queue circuits: the first was designed by Seitz [92] and the second was synthesized using the method of Lavagno *et al.* [61].

The automatic verifier that we use is our extension of Dill's trace theory verifier [38] that allows for the use of trace structures over the discrete time trace algebra $C_{PC}^{QTI\varphi}$ (in the verifier, trace structures actually consist of two sets of traces, a *success set* and a *failure set,* but that difference does not concern us here).

Together with the conservative approximations described earlier, the verifier can be used to prove correctness relative to the continuous time trace structure algebra $\mathcal{A}_C^{CTU}$. Although this verifier was first described in 1989 [16, 17], it still appears to be the state of the art in automatic verification of speed-dependent asynchronous circuits.

$$\begin{array}{lll} (y = \beta) \wedge (z = \beta) & \xrightarrow{y} & y := \neg\beta \\ (y = \beta) \wedge (z = \neg\beta) & \xrightarrow{y} & \textit{failure} \\ (y = \beta) \wedge (z = \neg\beta) & \xrightarrow{z} & z := \beta \end{array}$$

Figure 5.1: Delay insensitive buffer, the meta-variable β ranges over $\{0, 1\}$.

5.1 Hazard-Failure Delay Model

We begin by considering the trace structure modeling a speed-independent buffer with input y and output z. The buffer is described using a production rule notation (see figure 5.1) somewhat reminiscent of the notation used by Martin [71, 73]. The firing of a production rule is an instantaneous (atomic) event. It is possible for more than one production rule to fire simultaneously; however, we will only consider non-simultaneous firings here. Since the buffer in figure 5.1 is untimed, we can interpret its production rules as representing a set of traces in C^I; since its input is y and its output is z, the traces are elements of $\mathcal{B}^I_C(\{y, z\})$. Recall that $\mathcal{B}^I_C(\{y, z\})$ is equal to $(y + z)^*$.

A trace is in the set of traces represented by a set of production rules if and only if it corresponds to a *run* of the production rules. Consider a run of the production rules in figure 5.1. In the initial state, with y and z both equal to 0, only the first rule is firable. Since the first production rule is labeled with y (the symbol above the arrow), the trace of the run begins with y. If the second production rule firing is a y transition, then the trace of the run begins with yy, and the buffer goes into *failure mode*. Once in failure mode, any trace is possible. Thus, for example, the buffer includes all of the traces in $yy(y + z)^*$. We call this delay model the *speed-independent hazard-failure model,* because any hazard puts the buffer into failure mode (for our purposes, a hazard is two consecutive transitions on the input of a buffer, without an intervening output transition). The term *failure* is borrowed from Dill [38]. Dill used two sets of traces in each trace structure, a *failure set* and a *success set;* for simplicity, we just use one set of traces.

If the second production rule firing is a z transition, then the trace begins with yz. Continuing in this way, one can build up the trace corresponding to a particular run of the production rules. The set of traces represented by the production rules is equal to the set of traces that can be built up in this manner.

We can also interpret the production rules in figure 5.1 over the continuous time trace

algebra $\mathcal{C}^{CTU}$. Recall that each trace in $\mathcal{B}_{\mathcal{C}}^{CTU}(\{y,x\})$ is a subset of $\{y,z\} \times \Re^{\neq}$, where $\Re^{\neq}$ is the set of non-negative real numbers. In the initial state, with y and z both equal to 0, only the first rule is firable and it can fire at any time t'. Since the first production rule is labeled with y the trace of the run contains the event (y,t'). Assume the next production rule firing occurs at time t''. If the second production rule firing is a y transition, then the trace of the run contains the event (y,t''), and the buffer goes into *failure mode*. Thus, for example, the buffer includes all of the traces of the form

$$\{(y,t'),(y,t'')\} \cup x,$$

where x is a subset of

$$\{y,z\} \times \{t \in \Re^{\neq} : t > t''\}.$$

If the second production rule firing is a z transition, then the trace contains the event (z,t''). Continuing in this way, one can build up the trace corresponding to a particular run of the production rules.

The next step is to generalize the model of the buffer to include a lower bound Δ_{min} and an upper bound Δ_{max} on its delay. We do this by including *clocks* in the production rules to record the passage of time (see figure 5.2). The clock t in figure 5.2 is treated as a real numbered value when used in the precondition of a production rule. A clock can either be *running* or *stopped*. When stopped, its value is zero; when running, its value increases automatically and continuously with the passage of time. All clocks are initially stopped. The operation *restart*(t) sets the value of t to zero and starts the clock running, regardless of whether it was already running. Thus, if a clock is running, then its value represents the amount of time since it was last restarted. The operation *reset*(t) sets t to zero and stops it. A production rule with *disallow* as its right side has a special meaning: the precondition must never be allowed to be true. This can lead to complicated backtracking in general, but here *disallow* is only used to enforce upper bounds on the response time of a delay element.

Consider a run of the production rules in figure 5.2. In the initial state, with y and z both equal to 0 and t stopped, only the first rule is firable and it can fire at any time t'. Since the first production rule is labeled with y the trace of the run contains the evert (y,t'). When the rule fires, it restarts the clock t. Thus, until t is reset or restarted again, its value reflects the amount of time since the y transition. Assume the next production rule firing occurs at time t''. If $\Delta_{max} < t'' - t'$, then the precondition of rule 4 becomes true, but this is specifically disallowed. Thus, we know that $t'' \leq t' + \Delta_{max}$. If the second production rule firing is a y

$$
\begin{array}{lcl}
(y = \beta) \wedge (z = \beta) & \xrightarrow{y} & y := \neg\beta;\ restart(t) \\
(y = \beta) \wedge (z = \neg\beta) & \xrightarrow{y} & failure \\
 & & \\
(t \geq \Delta_{min}) \wedge (y = \beta) \wedge (z = \neg\beta) & \xrightarrow{z} & z := \beta;\ reset(t) \\
(t > \Delta_{max}) \wedge (y = \beta) \wedge (z = \neg\beta) & \longrightarrow & disallow
\end{array}
$$

Figure 5.2: Binary hazard-failure delay, the meta-variable β ranges over $\{0,1\}$.

transition, then the trace of the run contains the event (y, t''), and the delay element goes into failure mode. If the second production rule firing is a z transition, then $t'' \geq t' + \Delta_{min}$, and the trace contains the event (z, t''). In this case, the clock t is reset (set to zero and stopped) because there is no need to keep track of the passage of time when the delay element is in a quiescent state. Continuing in this way, one can build up the trace corresponding to a particular run of the production rules.

5.2 Approximating Continuous Time

We can also interpret production rules as representing trace structures over $\mathcal{C}_{PC}^{QTI\varphi}$. Recall that for a given alphabet A, the set traces $\mathcal{B}_P^{QTI\varphi}(A)$ of $\mathcal{C}_{PC}^{QTI\varphi}$ over alphabet A is

$$\epsilon + (A \cup \{\varphi\})^*(\{\varphi\}).$$

Earlier chapters have described a class of conservative approximations from traces structures over $\mathcal{C}_C^{CTU}$ to prefix-closed trace structures over $\mathcal{C}_{PC}^{QTI\varphi}$ (via $\mathcal{C}_C^{CTO}$, $\mathcal{C}_C^{QTS}$, $\mathcal{C}_C^{QTSI}$, $\mathcal{C}_C^{QTI}$, $\mathcal{C}_C^{QTI\varphi}$ and $\mathcal{C}_{C/P}^{QTI\varphi}$). Let Ψ be the tightest of these conservative approximations. Let T be the trace structure over $\mathcal{C}_C^{CTU}$ represented by the production rules in figure 5.2 with $\Delta_{min} = 2$ and $\Delta_{max} = 3$. It can be shown that the trace structure $T' = \Psi_u(T)$ is represented by the automata in figure 5.3.

The proof of this result is quite tedious and will not be presented here. This tedium can be avoided by showing the following more general results. Although we feel we have a good understanding of how to prove these more general results, they remain as future work. First, define two different formal semantics for the production rule language. The first semantics would be in terms of trace structures over $\mathcal{C}_C^{CTU}$ (continuous time), the second in terms of trace structures over $\mathcal{C}_{PC}^{QTI\varphi}$ (discrete time). Second, prove that for any set of syntactically well-formed set of production rules, the semantics over $\mathcal{C}_{PC}^{QTI\varphi}$ is a conservative approximation

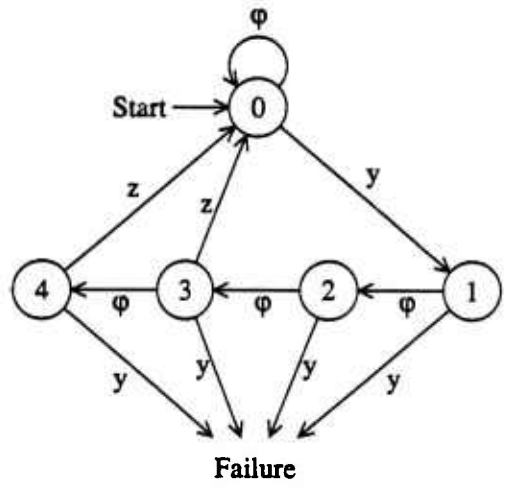


Figure 5.3: Automata that accepts the set $P' \subseteq \mathcal{B}_C^{QTI\varphi}$ of a buffer with minimum delay of 2 and maximum delay of 3.

of the semantics over $\mathcal{C}_C^{CTU}$ (this actually requires having three different semantics: T, $\Psi_l(T)$ and $\Psi_u(T)$, where T is the trace structure giving the continuous time semantics and $\Psi = (\Psi_l, \Psi_u)$ is the appropriate conservative approximation). It follows from these results that if an implementation of a "production rule compiler" satisfies the discrete time semantics, then it provides a conservative approximation of the continuous time semantics. In such an implementation (and the one used for the verification examples in this chapter), finite automata can be used to represent trace structures over $\mathcal{C}_{PC}^{QTI\varphi}$.

Applying the conservative approximation Ψ described above is not the only potential source of false negatives when using discrete time models. Let T_0 and T_0' be continuous time and discrete time models of a hazard-failure delay element with input y_0, output z_0, $\Delta_{min} = 1$ and $\Delta_{max} = 1$. We define T_1 and T_1' similarly except that they have input y_1 and output z_1. In the agent $T_0 \parallel T_1$, if a y_0 transition precede a y_1 transition, then the resulting z_0 transition is guaranteed to precede the resulting z_1 transition. However, the following trace is possible in $T_0' \parallel T_1'$:

$$y_0 y_1 \varphi z_1 z_0.$$

To see this, notice that

$$\begin{aligned} proj(\{y_0, z_0\})(y_0 y_1 \varphi z_1 z_0) &= y_0 \varphi z_0 \\ &\in P_0' \end{aligned}$$

$$proj(\{y_1, z_1\})(y_0 y_1 \varphi z_1 z_0) = y_1 \varphi z_1 \in P_1'.$$

5.3 Seitz Queue Element

In this section we analyze the self-timed queue element in figure 5.4. It is based on a circuit described by Seitz [92]. Seitz's original circuit does not have the two inverters between the E and G nodes shown in figure 5.4, and it also includes an initialization signal. Seitz's circuit is not speed-independent, but was intended to work under the more liberal 3/2 rule, which states that the total delay through any 3 gates is greater than the delay through any 2 gates. The control signals use 2-phase handshaking.

Seitz's original circuit was analyzed by Browne and Mishra *et al.* [9, 77]. They were not able to model the 3/2 rule, so the circuit was analyzed under a unit delay model. The unit delay model is more liberal (less conservative) than the 3/2 rule, so any bug discovered under the unit delay model is also a bug under the 3/2 rule. They discovered a bug, and proposed a modification to the circuit. Their modified circuit differed from the one in figure 5.4 by the absence of the two inverters between the E and G nodes mentioned above, and the addition of two more inverters, for a total of five, between the *AckOut* and E nodes. This circuit satisfied their specifications, but even in the unit delay model at least one bug remained that was not caught by their specifications. To see the bug, assume the circuit is in a quiescent state with the queue full (*Full1* is high and *Full0* is low) and there is a *ReqIn* pending. Assume an *AckOut* is received, and that there are no other input changes until the circuit is stable. The queue should become momentarily empty, and then become full again before the circuit stabilizes. But it is possible for the A signal to not remain high long enough to properly set the flip-flop, so the circuit can stabilize with the queue empty. We refer to this bug as the "dropped bit" bug.

Our analysis shows that the circuit in figure 5.4 is correct (up to safety properties) in a unit delay model, and is also correct in some timing models that are more conservative than the unit delay model. The circuit is not correct, however, in a model as conservative as the 3/2 rule.

Before giving the details of our analysis of the queue circuit, we should describe some of the limitations of the component model that was used. We started by modeling each gate with an ideal (delay-free) gate followed by hazard-failure delay element as described in figure 5.2. The

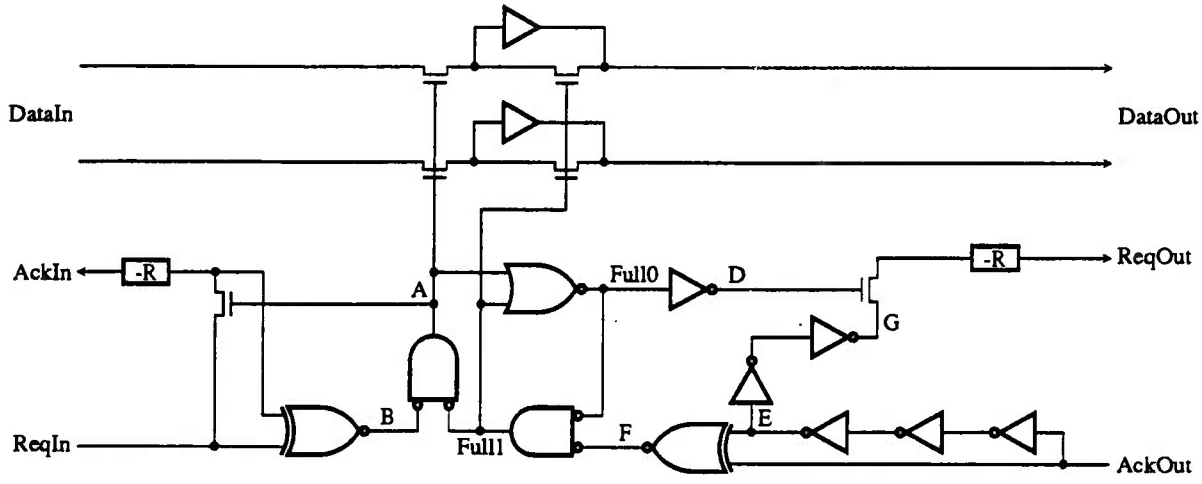


Figure 5.4: Queue element.

same values of Δ_{min} and Δ_{max} are used for each gate in the circuit. Nodes with indeterminate voltages are not modeled. So, it cannot be verified that an initialization signal works correctly, the verification is simply started with all nodes at the proper initial voltages. Also, the verifier cannot model transistors as switches, so the pass transistors in the circuit must be modeled as latches. The negative resistors are simply modeled as buffers with delay. For correct circuit operation, it is necessary that the delay be at least 3 gate delays for the negative resistor in the input section, and at least 2 gate delays for the negative resistor in the output section. These delays could be reduced if assumptions are made about the minimum response time of the environment. We remove the buffers in the data part since we cannot model their role in the circuit, which is to convert a dynamic storage node to a static storage node. Only one bit of the data path was modeled.

If any gate of the circuit goes into failure mode, then the resulting erratic transitions of the gate's output will eventually propagate to the interface of the circuit, causing it to not satisfy its specification. The gate driving *Full0* goes into failure mode, regardless of the values of Δ_{min} and Δ_{max}, in the following situation. The queue is full, and a *ReqIn* is pending, so *A*, *B* and *Full0* are low and *Full1* is high. As a result of an *AckOut* transition, *Full1* can go low. At Δ_{min} time units later, *A* can go high before *Full0* goes high, causing a hazard. This hazard puts the gate driving *Full0* into failure mode. We can use a more liberal model of the gate by assuming that it would fire between Δ_{min} and Δ_{max} time units after *Full1* goes low, even in the above scenario. Thus, we modify the trace structure modeling this gate so a trace

in which the gate is firable for Δ_{min} time units is not a failure, and the gate fires within the Δ_{max} time unit maximum delay, even if there is a hazard. This means that *Full0* can go high and then go low $2\Delta_{min} - \Delta_{max}$ time units later; thus, the buffer driving D is modeled so that it is not a failure whenever it is firable for at least $2\Delta_{min} - \Delta_{max}$ time units, even if there is a hazard. Thus, the model of the buffer driving D is slightly more liberal than the model of the gate driving *Full0*. The other gates could also be modeled similarly, but it is not necessary in order to verify the correctness of the circuit.

We used the verifier to determine for what values of Δ_{min} and Δ_{max} is the circuit correct. The circuit was originally claimed to be correct under the 3/2 rule, which states that the total delay through any 3 gates is greater than the delay through any 2 gates. This is not quite the same as saying the circuit is correct when $\Delta_{max} = 3$ and $\Delta_{min} = 2$, since that would allow the total delay through any 3 gates to be greater than *or equal to* the delay through any 2 gates. Nonetheless, we can show that the circuit is incorrect under the 3/2 rule; the verifier finds a variant of the "dropped bit" bug (described above) in the circuit when assuming that $\Delta_{max} = 6$ and $\Delta_{min} = 5$, which is a more optimistic assumption than the 3/2 rule. The verifier shows this bug by producing an error trace that puts the gate driving *Full0* into failure mode.

The circuit is correct as modeled when $\Delta_{max} = 7$ and $\Delta_{min} = 6$. The automatic verifier checked this by examining 8753 states in about 5 minutes on a Sun 3/60.

In an earlier description of this circuit [17], we reported that the circuit was correct for $\Delta_{max} = 6$ and $\Delta_{min} = 5$. That analysis was based on a discrete time model that differs slightly from the discrete time model used here (see figure 5.3). The difference is that the b transition from state 1 returns to state 0, rather than going in to failure mode. Thus, a hazard shorter than one clock tick (in discrete time) is ignored, which gives a more optimistic model. The intention was that this model would compensate for the extra conservativeness in the discrete time model caused by possible reordering of events between clock ticks (see the end of section 5.2). Now we understand that this discrete time model does not correspond to any continuous time model, and should be avoided. It does appear, however, that the circuit works correctly whenever

$$\frac{\Delta_{min}}{\Delta_{max}} > \frac{5}{6},$$

based on examining the error trace produced when $\Delta_{max} = 6$ and $\Delta_{min} = 5$. Also, the verifier shows that the circuit is correct for $\Delta_{max} = 13$ and $\Delta_{min} = 11$. For this model, the verifier examined 44,906 states in about 28 minutes.

$$
\begin{array}{lcl}
(y = \beta) \wedge (z = \beta) & \xrightarrow{y} & y := \neg\beta;\ restart(t) \\
(y = \beta) \wedge (z = \neg\beta) & \xrightarrow{y} & y := \neg\beta;\ reset(t) \\
 & & \\
(t \geq \Delta_{min}) \wedge (y = \beta) \wedge (z = \neg\beta) & \xrightarrow{z} & z := \beta;\ reset(t) \\
(t > \Delta_{max}) \wedge (y = \beta) \wedge (z = \neg\beta) & \longrightarrow & disallow
\end{array}
$$

Figure 5.5: Binary inertial delay, the meta-variable β ranges over $\{0, 1\}$.

5.4 Binary Inertial Delay

The hazard-failure model can be overly conservative in many situations. A common alternative is the inertial delay model [12, 91, 90]. Our formal model of a binary inertial delay element with input y and output z is given in figure 5.5, using production rules. Consider a run of the production rules in figure 5.5 In the initial state, with y and z both equal to 0 and t stopped, only the first rule is firable and it can fire at any time t'. Since the first production rule is labeled with y (the symbol above the arrow), the trace of the run contains the event (y, t'). When the rule fires, it restarts the clock t. Thus, until t is reset or restarted again, its value reflects the amount of time since the y transition. Assume the next production rule firing occurs at time t''. If $\Delta_{max} < t'' - t'$, then the precondition of rule 4 becomes true, but this is specifically disallowed. Thus, we know that $t'' \leq t' + \Delta_{max}$. If the second production rule firing is a y transition, then the trace of the run contains the event (y, t''). If the second production rule firing is a z transition, then $t'' \geq t' + \Delta_{min}$, and the trace contains the event (z, t''). In both cases, the clock t is reset (set to zero and stopped) because there is no need to keep track of the passage of time when the delay element is in a quiescent state. Continuing in this way, one can build up the trace corresponding to a particular run of the production rules. The set of traces represented by the production rules is equal to the set of traces that can be built up in this manner.

The distinctive feature of the production rule description of inertial delay is rule 2. It specifies that if two consecutive y transitions occur without a z transition in between, then the state of the delay element is the same as if no transitions occurred. Thus, a hazard is treated as if nothing happened. As an extreme example, consider a signal that transitions every t' time units, where t' is slightly less then Δ_{min}. If this signal is input to an inertial delay element, then the output is constant, which is clearly overly optimistic.

5.5 Binary Chaos Delay

In the binary chaos delay model a delay element goes into a special mode, called *chaos mode,* when there is a hazard on its input. When in chaos mode, the output of the delay element can transition unpredictably, which conservatively models the unpredictability of an actual gate responding to a hazard. In this sense, chaos mode is like failure mode. The difference is that chaos delay allows the delay element to leave chaos mode if its input does not transition for a period of length Δ_{max}, in which case the delay element enters a quiescent state with its output equal to its input.

A circuit can work properly even if one of its gates enters chaos mode, as long as the random outputs of the gate are not allowed to propagate to the interface of the circuit. If the hazard-failure delay model were used when verifying such a circuit, a false negative would result. There are examples of this happening in practice. The synthesis techniques of Lavagno *et al.* [61] can produce circuits that are correct under the chaos delay model but incorrect under the hazard-failure delay model [60].

The term "chaos" is borrowed from Josephs and Udding [53], who used a *chaos process* to represent the response of a delay-insensitive process to a hazard. In their model, however, it is impossible for a component to ever leave chaos mode.

The production rules for the chaos delay model have an extra boolean state variable c which is equal to 1 if and only if the delay element is in chaos mode (see figure 5.6). Rule 2 is the major difference between inertial delay and chaos delay; it requires that the delay element go into chaos mode in response to a hazard. The clock t is restarted in order to record the amount time that must pass before the delay element can exit chaos mode. In rule 3, the clock is restarted again if another input transition occurs.

Rules 4 and 5 control the minimum and maximum response time of the delay element when there are no hazards (i.e., not in chaos mode). Rule 6 allows the output to transition unpredictably in chaos mode. In rule 7, chaos mode can be exited if sufficient time has passed and the values of the input and output are equal. Rule 8 requires that chaos mode must be exited after sufficient time. This forces the output to become equal to the input before more than Δ_{max} time has passed.

$$
\begin{array}{rrl}
(y=\beta) \wedge\ (z=\beta)\ \wedge (c=0) & \xrightarrow{y} & y := \neg\beta;\ restart(t) \\
(y=\beta) \wedge (z=\neg\beta) \wedge (c=0) & \xrightarrow{y} & y := \neg\beta;\ c := 1;\ restart(t) \\
(y=\beta) \wedge \qquad\qquad (c=1) & \xrightarrow{y} & y := \neg\beta;\ restart(t)
\end{array}
$$

$$
\begin{array}{rrl}
(t \geq \Delta_{min}) \wedge (y=\beta) \wedge (z=\neg\beta) \wedge (c=0) & \xrightarrow{z} & z := \beta;\ reset(t) \\
(t > \Delta_{max}) \wedge (y=\beta) \wedge (z=\neg\beta) \wedge (c=0) & \longrightarrow & disallow
\end{array}
$$

$$
\begin{array}{rrl}
(z=\beta)\ \wedge (c=1) & \xrightarrow{z} & z := \neg\beta \\
(t=\Delta_{max}) \wedge (y=\beta) \wedge\ (z=\beta)\ \wedge (c=1) & \longrightarrow & c := 0;\ reset(t) \\
(t > \Delta_{max}) \wedge \qquad\qquad (c=1) & \longrightarrow & disallow
\end{array}
$$

Figure 5.6: Binary chaos delay. The meta-variable β ranges over $\{0, 1\}$.

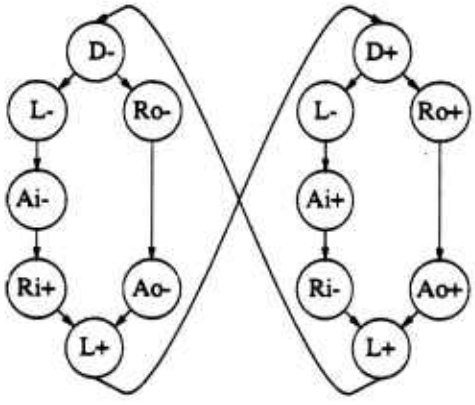


Figure 5.7: STG specification for a FIFO controller.

5.6 FIFO Controller

We compared the binary inertial delay and binary chaos delay models by verifying a speed-dependent FIFO controller circuit. The specification for the FIFO controller, which is due to Chu [30], is given as a Signal Transition Graph (STG) in figure 5.7. The automatic verifier that was used is based on an extension of Dill's trace theory that allows for the modeling of real-time properties [17]. It uses a discrete time model that is a provably conservative approximation of a continuous time model. As a result, if a circuit is verified correct under this discrete time model, then it is guaranteed to be correct under the continuous time model.

In figure 5.8, the circuit that was checked is described using a LISP-like language that can be read by the automatic verifier. For each gate, the first argument is the input(s), the second argument is the output, and the third argument gives the minimum and maximum

```
; Initially Ri=0, Ao=0, D=0, Ro=0, Ai=0,
;   L=0, W1=1, W2=1, W3=1, W4=0, W5=1,
;   W6=1, W7=1, W8=1, W9=1
(compose
    (buffer D Ro)
    (inverter L W1 (4 7))
    (orgate (-W1 -D) W2 (8 12))
    (inverter D W3 (4 7))
    (orgate (-W3 -W1) W4 (8 12))
    (orgate (-Ai -W4) W5 (8 12))
    (orgate (-W2 -W5) Ai (8 12))
    (inverter Ao W6 (4 6))
    (orgate (-W3 -W6 -Ri) W7 (14 21))
    (inverter Ri W8 (4 6))
    (orgate (-D -W8 -Ao) W9 (14 21))
    (orgate (-W7 -W9) L (8 12)))
```

Figure 5.8: Implementation of FIFO controller.

delays of the gate. If there is no third argument, then the gate has unbounded delay (i.e., is a speed-independent gate). Negated inputs are denoted by a minus sign. The circuit is based on a design synthesized using the method of Lavagno *et al.* [61]. It was intentionally synthesized to have an error, in order to test the gate models used with the verifier [60].

We checked the circuit under the inertial delay model and the chaos delay model. In both cases, gates are modeled as an ideal (delay free) gate whose output feeds a delay element of the appropriate type. Under the inertial delay model the circuit is correct. The verifier checked this by examining 6,450 states in less than 190 seconds of CPU time on a Sun 3/60.

Under the chaos delay model the circuit does not satisfy the specification. The counter-example trace returned by the verifier is

$$\begin{aligned} &\text{Ri}+\ \varphi^4\ \text{W8}-\ \varphi^{10}\ \text{W7}-\ \varphi^8\ \text{L}+\ \text{D}+\ \varphi^4 \\ &\text{W1}-\ \text{W2}-\ \text{W2}+\ \text{Ai}+, \end{aligned} \tag{5.1}$$

which represents a possible behavior of the circuit that is not consistent with the specification. The symbol φ in this trace gives information about the times at which transitions occur. Assume the trace begins at time 0, and let T be the basic unit of time. If a transition occurs between the nth and $(n+1)$th φ in the trace, then the transition occurs between times nT and $(n+1)T$. Superscripts are used to indicate multiple occurrences of φ. Thus, the transition

of W8 in the trace occurs between times $4T$ and $5T$. The key event in the trace is the final transition of W1, which causes a hazard on the gate driving W2. This hazard is ignored in the inertial delay model, but in the chaos delay model it puts the gate into chaos mode, resulting in two consecutive transitions of W2. This puts the gate driving Ai into chaos mode, causing an Ai transition earlier than is allowed by the specification. This is an illustration of how the inertial delay model can lead to false positive verification results.

5.7 A Less Conservative Model

Although the chaos delay model is not as conservative as failing on all hazards, it may still be overly conservative. This is illustrated in the counter-example trace (5.1). The length of the hazard in the trace is 4 time units (the time between the D+ and W1− transitions), which is half the minimum delay of the relevant gate. Depending on how the gate is implemented, a pulse this short might be reliably filtered out. Also, once the hazard occurs, the output of the gate (W2) immediately becomes unpredictable. In practice, the output would remain stable until Δ_{min} time units after the first transition in the hazard (D+, in this case).

Both of these issues are addressed in the model described in figure 5.9. An additional parameter, Δ_{haz}, is used to control the length of the longest hazard that is ignored by the delay element. If a hazard is shorter than Δ_{haz}, then that hazard is ignored, just as in the inertial delay model. If a hazard is longer than Δ_{haz}, then the delay element goes into chaos mode. Thus, this model unifies inertial delay and chaos delay: if $\Delta_{haz} = 0$, then it goes into chaos mode in response to any hazard; if $\Delta_{haz} > \Delta_{max}$, then it is identical to the inertial delay model.

The production rules in figure 5.9 use two clocks, t_z and t_c. The clock t_z records the delay until the output z transitions. The clock t_c records the time that must pass before the delay element can exit chaos mode; thus, it only runs in chaos mode. Both of these functions could be combined into one clock t in our previous chaos delay model (figure 5.6).

The first production rule in figure 5.9 is the same as the first rule of the original chaos model, except that t_z is used instead of t. Rule 2 of the original model is split into rule 2 (which acts like the inertial model for short hazards) and rule 3 (which goes into chaos mode for long hazards). Anytime rule 3 fires, t_z is already running (because of a previous firing of rule 1) and its value is not affected. The final six rules in figure 5.9 correspond to the final six rules of the original model. The only changes are that references to t are replaced by references

$$
\begin{array}{rl}
(y=\beta) \wedge\ (z=\beta)\ \wedge (c=0) & \xrightarrow{y}\ y := \neg\beta;\ restart(t_z) \\
(t_z < \Delta_{haz}) \wedge (y=\beta) \wedge (z=\neg\beta) \wedge (c=0) & \xrightarrow{y}\ y := \neg\beta;\ reset(t_z) \\
 & \\
(t_z \geq \Delta_{haz}) \wedge (y=\beta) \wedge (z=\neg\beta) \wedge (c=0) & \xrightarrow{y}\ y := \neg\beta;\ c := 1;\ restart(t_c) \\
(y=\beta) \wedge\ (c=1) & \xrightarrow{y}\ y := \neg\beta;\ restart(t_c) \\
 & \\
(t_z \geq \Delta_{min}) \wedge (y=\beta) \wedge (z=\neg\beta) \wedge (c=0) & \xrightarrow{z}\ z := \beta;\ reset(t_z) \\
(t_z > \Delta_{max}) \wedge (y=\beta) \wedge (z=\neg\beta) \wedge (c=0) & \longrightarrow\ disallow \\
 & \\
(t_z \geq \Delta_{min}) \wedge\ (z=\beta)\ \wedge (c=1) & \xrightarrow{z}\ z := \neg\beta \\
(t_c = \Delta_{max}) \wedge (y=\beta) \wedge\ (z=\beta)\ \wedge (c=1) & \longrightarrow\ c := 0;\ reset(t_c);\ reset(t_z) \\
(t_c > \Delta_{max}) \wedge\ (c=1) & \longrightarrow\ disallow
\end{array}
$$

Figure 5.9: Extended binary chaos delay with hazard length parameter and delayed chaos output.

to t_c or t_c, as appropriate, and rule 7 requires that $t_z \geq \Delta_{min}$ before the output can transition in chaos mode.

The model can be further generalized to include five parameters, instead of just three. The parameter Δ_{min} could have different values for rules that fire in chaos mode than for rules that fire when not in chaos mode; similarly for Δ_{max}. However, we do not consider this generalization further here.

We applied the generalized model in figure 5.9 to the verification problem described earlier. For each gate, we let $\Delta_{haz} = \lfloor 0.75\Delta_{min} \rfloor$. The circuit still did not satisfy its specification even under this more optimistic gate model. The counter-example trace that the verifier produced is

Ri+ φ^4 W8− φ^{10} W7− φ^8 L+ D+ φ^4
W3− φ^2 W1− φ^2 W2− φ^8 Ai+.

Notice that the time between the D+ and the W1− transitions is six time units (which is Δ_{haz} for the gate with those inputs) rather than four, as in the other trace. Also, once in chaos mode, $W2$ does not transition until 8 time units after D did.

Determining that the circuit was not correct, and finding the counter-example trace, required examining slightly fewer states than in the inertial delay case; the verification time was proportionally reduced. This is typical for automatic verification methods based on trace theory; finding an error is usually faster than verifying correctness. Since the circuit is still not

correct even under such an optimistic gate model, it is unlikely that the circuit would work reliably if implemented. We could not be as certain of this conclusion if we had only used the more conservative model of figure 5.6.

5.8 Single Trajectory Delay Models

The binary inertial delay can be extended to use ternary logic values. This idea has been used to develop efficient, conservative simulation algorithms based on inertial delay [90].

Binary bounded delay models can be difficult to analyze because of the non-determinism introduced by having component delays possibly vary. If this non-determinism is represented using the ternary value X, then it is possible to construct a *single trajectory* model [11]. The key property of a single trajectory model is that for a given input stream, only one sequence of output transitions is possible. Computationally, this can be much more efficient then representing non-determinism with a large number of different binary transition sequences. However, single trajectory models can be more conservative and, therefore, lead to more false negative verification results.

Seger [90, 91] used this idea to develop an efficient algorithm for analyzing races in asynchronous circuits. Unlike the models we describe in this section, Seger's *extended inertial delay model* is not a actually single trajectory model. However, only a single trajectory of Seger's model needs to be considered to accurately analyze circuits; this is the key to the efficiency of his analysis algorithm. In our work, the property that only a single trajectory needs to be considered is made explicit in the models themselves.

A single trajectory inertial delay model is described using production rules in figure 5.10. In the production rules for the binary delay models, we labeled the arrows with the name of the signal that transitioned. In the non-binary models of this section, the label must also indicate what value the signal transitions to. Two different clocks, t_B and t_X, are required in the ternary inertial delay model. The clock t_B is used to enforce time bounds on when z must transition to a binary value; t_X enforces time bounds on when z must transition to a non-binary value. We assume $0 < \Delta_{min} < \Delta_{max}$.

In the first rule, the delay element is quiescent with binary values on its input and output. When the input transitions to X, the clock t_X is started to record the delay before z transitions to X; the clock t_B remains stopped. In the second rule, t_B is initially running because z is being driven to a binary value. Once y transitions to X, the clock t_B can be stopped; t_X,

$$
\begin{array}{lll}
(y = \beta) \wedge (z = \beta) & \xrightarrow{y := X} & restart(t_X) \\
(y = \beta) \wedge (z \neq \beta) & \xrightarrow{y := X} & reset(t_B) \\
 & & \\
(y = \beta) \wedge (z = \beta) & \xrightarrow{y := \neg\beta} & restart(t_X);\ restart(t_B) \\
(y = X) \wedge (z = \beta) & \xrightarrow{y := \neg\beta} & restart(t_B) \\
 & & \\
(y \neq \beta) \wedge (z = \beta) & \xrightarrow{y := \beta} & reset(t_X);\ reset(t_B) \\
(y \neq \beta) \wedge (z = X) & \xrightarrow{y := \beta} & restart(t_B) \\
 & & \\
(t_X = \Delta_{min}) \wedge & (z \neq X) & \xrightarrow{z := X} \quad reset(t_X) \\
(t_X > \Delta_{min}) \wedge & (z \neq X) & \longrightarrow \quad disallow \\
 & & \\
(t_B = \Delta_{max}) \wedge (y = \beta) \wedge (z \neq \beta) & & \xrightarrow{z := \beta} \quad reset(t_B) \\
(t_B > \Delta_{max}) \wedge (y = \beta) \wedge (z \neq \beta) & & \longrightarrow \quad disallow
\end{array}
$$

Figure 5.10: Extended inertial delay. The meta-variable β ranges over $\{0, 1\}$.

which can be stopped or running, is unchanged. When y transitions to a binary value not equal to z, then t_B is restarted, as in rules 3, 4 and 6. The fifth rule expresses the key property of the inertial model: when y transitions to a binary value equal to z, both clocks are reset as if no hazard occurred. The remaining rules control the transitions of the output z. Notice that for any sequence of input transitions, there is only one possible sequence of production rule firings, even if the time of the firings is considered. This is the key property of a single trajectory model.

It is also possible to define a single trajectory version of the chaos delay model. However, since the chaos delay model distinguishes between multiple transitions (a hazard) and a single transition that occurs at an unknown time (the normal case), three logic values are not adequate for this purpose. Two additional values D and U (for a total of five) representing downward and upward transitions must be added. The remainder of this section gives a brief description of the model (see figure 5.11).

The operations $\triangleleft$ and $\triangleright$ take a single binary argument and are defined by

$$
\begin{array}{llll}
\triangleleft 0 & = & U & \qquad \triangleright 0 \ = \ D \\
\triangleleft 1 & = & D & \qquad \triangleright 1 \ = \ U.
\end{array}
$$

$$(y = \beta) \wedge (z = \beta) \xrightarrow{y := \triangleleft\beta} \mathit{restart}(t_X)$$

$$(y = \beta) \wedge (z = \beta) \xrightarrow{y := \neg\beta} \mathit{restart}(t_X);\ \mathit{restart}(t_B)$$

$$(y = \beta) \wedge (z = \alpha) \xrightarrow[z := X]{y := \triangleleft\beta} \mathit{reset}(t_X);\ \mathit{reset}(t_B)$$
where $(\alpha \neq \beta) \wedge (\alpha \neq X)$

$$(y = \beta) \wedge (z = \alpha) \xrightarrow[z := X]{y := \neg\beta} \mathit{reset}(t_X);\ \mathit{restart}(t_B)$$
where $(\alpha \neq \beta) \wedge (\alpha \neq X)$

$$(y = \triangleright\beta) \wedge (z \neq X) \xrightarrow{y := \beta} \mathit{restart}(t_B)$$

$$(y \neq X) \wedge (z \neq X) \xrightarrow[z := X]{y := X} \mathit{reset}(t_X);\ \mathit{reset}(t_B)$$

$$(y \neq \beta) \wedge (z = X) \xrightarrow{y := \beta} \mathit{restart}(t_B)$$
$$(y \neq \alpha) \wedge (z = X) \xrightarrow{y := \alpha} \mathit{reset}(t_B)$$
where $(\alpha \neq 0) \wedge (\alpha \neq 1)$

$$(t_X = \Delta_{min}) \wedge (y = \alpha) \wedge (z = \beta) \xrightarrow{z := \triangleleft\beta} \mathit{reset}(t_X)$$
where $(\alpha = \neg\beta) \vee (\alpha = \triangleleft\beta)$

$$(t_X > \Delta_{min}) \wedge (y = \alpha) \wedge (z = \beta) \longrightarrow \mathit{disallow}$$

$$(t_B = \Delta_{max}) \wedge (y = \beta) \wedge (z \neq \beta) \xrightarrow{z := \beta} \mathit{reset}(t_B)$$
$$(t_B > \Delta_{max}) \wedge (y = \beta) \wedge (z \neq \beta) \longrightarrow \mathit{disallow}$$

Figure 5.11: Extended chaos delay. The meta-variables α and β range over $\{0, 1, D, U, X\}$ and $\{0, 1\}$, respectively.

As a memory aide, notice that in the equation $\triangleright 0 = D$, for example, the triangle points *to* the 0, and D is the value of a signal that is transitioning *to* 0. In the equation $\triangleleft 0 = U$, the triangle points away *from* the 0, and U is the value of a signal that is transitioning *from* 0.

The delay element described in figure 5.11 is in chaos mode if and only if its output is X. Thus, there is no need for the state variable c that was used in the binary chaos delay model.

Transitions from β to $\triangleright\beta$, where β is a binary value, are not allowed in the model, since they are not physically meaningful. Similarly, $\triangleright\beta$ can only transition to β and to X, and X can only transition to a binary value. The single trajectory chaos delay element enforces these restrictions on its output, and assumes that its input satisfies these restrictions.

In the first rule, the delay element is quiescent with the binary value β on its input and output. When the input transitions to $\triangleleft\beta$, the clock t_X is started to record the delay before z transitions to $\triangleleft\beta$; the clock t_B remains stopped. If y transitions from β directly to $\neg\beta$, as in rule 2, then both clocks need to be restarted. Rules 3 and 4 involve y transitions that put the delay element in chaos mode. This results in z transitioning to X simultaneously with the y transition, which is represented by having two labels (one for each simultaneous transition) on the arrow of the production rule. Rules 5 through 8 handle the rest of the possible input transitions. The remaining rules control the transitions of the output z.

It can be shown that for reachable states of the delay element, if $y = \beta$, then $z \neq \triangleleft\beta$. Also, if $y = \triangleright\beta$, then $z \neq \beta$ and $z \neq \triangleleft\beta$. Finally, if $y = X$, then $z = X$.

5.9 Discussion

We verified two speed-dependent asynchronous circuits, using a variety of delay models. We demonstrated that the binary inertial delay model can lead to false positive results on one of those circuits. Using the binary chaos delay model, the verifier was able to discover an error in the same circuit.

We described how the binary inertial and binary chaos delay models can be extended to single trajectory models, using 3-valued and 5-valued logics, respectively. It may be possible to combine the binary and the extended models to achieve a better balance between efficiency and accuracy. For example, a subcircuit with reconvergent fanout could be analyzed with binary chaos delay, with the results then abstracted into the single trajectory model. Then the single trajectory model could be used to efficiently simulate or verify the full circuit without having the reconvergent fanout cause an overly conservative result.

Any such model could be immediately used by our automatic verifier; all that is necessary is to compile the models into the appropriate finite automata representations.

Chapter 6

Future Research

In this thesis, we have described general techniques, based on trace algebra and trace structure algebra, for constructing domains of agents models. We introduced the idea of conservative approximations between trace structure algebras, and constructed conservative approximations from continuous time models to discrete time models and from explicit simultaneity semantics to interleaving semantics. We implemented an automatic verifier and demonstrated it on speed-dependent asynchronous circuits with several new delay models.

The work described in this thesis is very much work in progress. The most pressing tasks are to formalize continuous time and discrete time semantics for the production rule notation used in section 5.2, and to show that these semantics are appropriately related by a conservative approximation. The discrete time semantics would be used in the existing automatic verifier. If a system is verified to be correct under the discrete time semantics, then it is guaranteed to also be correct under the continuous time semantics. Currently, it is difficult to verify that the discrete time semantics of a particular agent is a conservative approximation of the desired continuous time semantics.

It is important to understand how much information is lost when using a conservative approximation $\Psi = (\Psi_l, \Psi_u)$ from a continuous time model to a discrete time model. One way to describe the information loss is to characterize the set $\mathcal{T}$ of continuous time trace structures T for which $\Psi_l(T) = \Psi_u(T)$. This is the same as the image of Ψ_{inv} (see section 4.4 for a description of the inverse of a conservative approximation). If T_0 is a continuous time trace structure that is used in a verification problem, the chances of a false negative verification result are reduced if T_0 is a member of $\mathcal{T}$. We have described in previous work [19] how $\mathcal{T}$ can be made to include more realistic models by using abstractions defined only on *initially speed-independent* trace structures. A trace structure $T = (\gamma, P)$ is initially speed-independent

if $suf(x, P) = P$ for any partial trace x that represents a behavior where no actions occur (only time passes); this is a much weaker requirement then speed-independence. All of the agents that can be expressed using the production rule notation of chapter 5 are initially speed-independent since all clocks are stopped in the initial state.

An area for future research is to integrate the idea of initially speed-independent trace structures with our more recent work on conservative approximations of real-time models. We conjecture that all of the continuous time agents expressible with our production rules can be represented exactly by trace structures over $\mathcal{C}^{QTS}$ (the model of quantized time with simultaneity). However, an implementation of the production rule language using discrete time clocks will not always produce this exact representation; a more sophisticated algorithm is required. We will also explore how these results relate to Henzinger, Manna and Pnueli's notion of *digitizable* agents [47].

We would also like to use trace algebra and conservative approximations to study several untimed models of concurrency, such as Mazurkiewicz traces and partial orders. We believe such a study might shed some light on the relationships between these models and interleaving semantics. The relationship between action based models and state based models is another area for future research.

We would like to extend some of our techniques. Trace algebra homomorphisms and conservative approximations could be allowed to change alphabets. This would significantly increase the number of useful abstractions that could be constructed with conservative approximations. It should also be possible to extend trace structures to include two sets of traces (like the success sets and failure sets of Dill's trace structures) and to generalize the notion of *receptiveness* [38] to arbitrary trace structure algebras.

Appendix A

Summary of Notation

$\mathcal{C}_C^{CTO}$	Continuous Time with Ordered rep., isomorphic to $\mathcal{C}_C^{CTU}$ (def. 3.14, p. 69)
$\mathcal{C}_C^{CTU}$	Continuous Time with Unordered rep. (def. 3.2, p. 59)
$\mathcal{C}^I$	extends $\mathcal{C}_C^I$ with partial traces (def. 4.23, p. 88)
$\mathcal{C}_C^I$	(Untimed) Interleaving Semantics (def. 2.9, p. 27)
$\mathcal{C}_C^{QTI}$	Quantized Time with Interleaving (def. 4.8, p. 82)
$\mathcal{C}^{QTI\varphi}$	extends $\mathcal{C}_C^{QTI\varphi}$ with partial traces (def. 4.24, p. 88)
$\mathcal{C}_C^{QTI\varphi}$	isomorphic to $\mathcal{C}_C^{QTI}$, uses φ to denote time (def. 4.10, p. 83)
$\mathcal{C}_C^{QTS}$	Quantized Time with Simultaneity (def. 3.10, p. 65)
$\mathcal{C}_C^{QTSI}$	isomorphic to $\mathcal{C}_C^{QTS}$, power set algebra over $\mathcal{C}_C^{QTI}$ (def. 4.16, p. 84)
$\mathcal{C}_C^{ST}$	Synchronous Time (def. 3.6, p. 60)

Table A.1: Summary of Trace Algebras

$\mathcal{A}_C^{CTU}$	all trace str's over $\mathcal{C}_C^{CTU}$ (def. 3.5, p. 60)
$\mathcal{A}^I$	extends $\mathcal{A}_C^I$ with partial traces (def. 4.57, p. 104)
$\mathcal{A}_C^I$	all trace str's over $\mathcal{C}_C^I$ (def. 2.31, p. 40)
$\mathcal{A}^{IR}$	extends $\mathcal{A}_C^{IR}$ with partial traces (def. 4.58, p. 104)
$\mathcal{A}_C^{IR}$	mixed regular trace str's over $\mathcal{C}_C^I$ (def. 2.32, p. 41)
$\mathcal{A}_C^{QTI}$	all trace str's over $\mathcal{C}_C^{QTI}$ (def. 4.12, p. 83)
$\mathcal{A}_C^{QTI\varphi}$	all trace str's over $\mathcal{C}_C^{QTI\varphi}$ (def. 4.12, p. 83)
$\mathcal{A}_C^{QTS}$	all trace str's over $\mathcal{C}_C^{QTS}$ (def. 3.13, p. 68)
$\mathcal{A}_C^{ST}$	all trace str's over $\mathcal{C}_C^{ST}$ (def. 3.8, p. 61)

Table A.2: Summary of Trace Structure Algebras

Symbol	Decorations	Denotes
A^*	none	set of all finite sequences over A
A^ω	none	set of all infinite sequences over A
A^∞	none	$A^* \cup A^\omega$
$\lfloor t \rfloor$	none	floor of t
$X \cup Y$	none	union of sets X and Y
$\cup X$	none	union of the sets in the set X
2^X	none	set of subsets of an arbitrary set X
$X \subseteq Y$	none	X subset of Y
$T \subseteq T'$	none	trace structure T contained in T' (def. 2.21, p. 34)
$X \times Y$	none	cartesian product of X and Y
$x \cdot y$	none	concatenation of traces in trace algebra (def. 4.20, p. 86)
$\emptyset$	none	empty set
$A \rightarrow B$	none	set of all partial functions with domain A and codomain B
$A \mapsto B$	none	set of all total functions with domain A and codomain B
$r \mid_{A \rightarrow B}$	none	function r restricted to domain A and codomain B
$E \parallel E'$	none	parallel composition of agents in concurrency algebra (def. 2.6, p. 23)
$T \parallel T'$	none	parallel composition of trace structures in trace structure algebra (def. 2.18, p. 33)
$\lvert B \rvert$	none	number of elements in set B

Symbol	Decorations	Denotes
Γ	none	set of all agent signatures (def. 2.1, p. 22)
γ	primes, integer sub's	agent signature (def. 2.1, p. 22), default agent signature of E (note 2.4, p. 23) and T (note 2.16, p. 33)
Δ_{haz}	none	length of longest ignorable hazard (p. 121)
Δ_{max}	none	maximum delay (p. 111)
Δ_{min}	none	minimum delay (p. 111)
ϵ	alphabet sub's	empty trace (T13, p. 87), empty sequence
λ	none	functional abstraction
φ	none	passage of a unit of time in traces of $C_C^{QTI\varphi}$ (def. 4.10, p. 83) and $C^{QTI\varphi}$ (def. 4.24, p. 88)
Ψ	primes	conservative approximation (def. 2.34, p. 42)
Ψ_{inv}	primes	inverse of Ψ (def. 4.60, p. 105)
Ψ_l	primes	lower bound mapping of Ψ
Ψ_u	primes	upper bound mapping of Ψ
τ	primes, integer sub's	sequence of time stamps (def. 3.14, p. 69)
ω	none	infinity

Symbol	Decorations	Denotes
A	primes, integer sub's	alphabet (def. 2.2, p. 22), default alphabet of γ (note 2.3, p. 22)
a	primes, integer sub's	signal (def. 2.1, p. 22)
$\mathcal{A}$	primes, mnem. sup's	trace structure algebra with partial traces (def. 4.48, p. 100)
$\mathcal{A}_C$	primes, mnem. sup's	trace structure algebra without partial traces (def. 2.17, p. 33)
$\mathcal{A}_{PC}$	primes, mnem. sup's	trace structure algebra of prefix-closed trace structures (def. 4.43, p. 96)
A1,...,A4	none	antecedents for thm. 2.35 (p. 43)
B	primes, integer sub's	alphabet (def. 2.2, p. 22)
b	primes, integer sub's	signal (def. 2.1, p. 22)
$\mathcal{B}$	primes, mnem. sup's	set of all traces in a trace algebra with partial traces (def. 4.20, p. 86)
$\mathcal{B}(A)$	primes, mnem. sup's	set of all traces over alphabet A in a trace algebra with partial traces (def. 4.20, p. 86)
$\mathcal{B}_C$	primes, mnem. sup's	set of all complete traces in a trace algebra (def. 2.7, p. 26; def. 4.20, p. 86)
$\mathcal{B}_C(A)$	primes, mnem. sup's	set of all complete traces over alphabet A in a trace algebra (def. 2.7, p. 26; def. 4.20, p. 86)
$\mathcal{B}_P$	primes, mnem. sup's	set of all partial traces in a trace algebra (def. 4.20, p. 86)
$\mathcal{B}_P(A)$	primes, mnem. sup's	set of all partial traces over alphabet A in a trace algebra (def. 4.20, p. 86)

Symbol	Decorations	Denotes
$\mathcal{C}$	primes, mnem. sup's	trace algebra with partial traces (def. 4.20, p. 86)
$\mathcal{C}_C$	primes, mnem. sup's	trace algebra without partial traces (def. 2.7, p. 26; def. 4.25, p. 89)
$\mathcal{C}_P, \mathcal{C}_{PC}$	primes, mnem. sup's	trace algebra (def. 4.25, p. 89)
$\mathcal{C}_{C/P}$	primes, mnem. sup's	trace algebra with traces represented by their set of prefixes (def. 4.45, p. 97)
C1,...,C9	none	axioms of concurrency algebra (def. 2.6, p. 23)
$codom(f)$	none	codomain of an arbitrary function f
$\mathcal{D}$	primes, mnem. sup's	domain of agents for a concurrency algebra (def. 2.6, p. 23)
$dom(f)$	none	codomain of an arbitrary function f
E	primes, integer sub's	agent in a concurrency algebra (def. 2.6, p. 23)
h	none	homomorphism from one trace algebra to another (def. 2.38, p. 45)
I	primes, integer sub's	set of input signals (def. 2.2, p. 22), default input signal set of γ (note 2.3, p. 22)
$id_A(a)$	none	identity function over set A
l	primes, integer sub's	integer
$\mathcal{L}(B)$	none	subset of $2^B_C(B)$ (thm. 2.30, p. 40)
L1,...,L5	none	antecedents for thm. 2.30 (p. 40) and thm. 4.56 (p. 104)
$len(u)$	none	length of sequence u
m	primes, integer sub's	integer
n	primes, integer sub's	integer
$\mathcal{N}$	none	integers
$\mathcal{N}^+$	none	non-negative integers
$\mathcal{N}^+$	none	positive integers
O	primes, integer sub's	set of output signals (def. 2.2, p. 22), default output signal set of γ (note 2.3, p. 22)
P	primes, integer sub's, mnem. sub's l and u	set of possible traces of a trace structure

Symbol	Decorations	Denotes
T1,...,T8	none	axioms of trace algebra without partial traces (def. 2.7, p. 26)
T9,...,T19	none	additional axioms of trace algebra with partial traces (def. 4.20, p. 86)
$pref(X)$	none	prefixing on traces in a trace algebra (def. 4.26, p. 89)
$proj(B)(E)$	none	projection on agents in a concurrency algebra (def. 2.6, p. 23)
$proj(B)(T)$	none	projection on trace structures in a trace structure algebra (def. 2.19, p. 33)
$proj(B)(x)$	none	projection on traces in a trace algebra (def. 4.20, p. 86)
$r(a)$	primes, integer sub's	renaming function (def. 2.5, p. 23)
$\Re$	none	real numbers
$\Re^{\nearrow}$	none	non-negative real numbers
$\Re^{+}$	none	positive real numbers
$rename(r)(E)$	none	renaming on agents in a concurrency algebra (def. 2.6, p. 23)
$rename(r)(T)$	none	renaming on trace structures in a trace structure algebra (def. 2.20, p. 34)
$rename(r)(x)$	none	renaming on traces in a trace algebra (def. 4.20, p. 86)
$reset(t)$	none	operation on clock t (p. 111)
$restart(t)$	none	operation on clock t (p. 111)
$interleave(x)$	none	set of interleavings of a trace x (def. 4.15, p. 84)
$suf(x,T)$	none	suffixing on trace structures in a trace structure algebra (def. 4.48, p. 100)
$suf(x,X)$	none	suffixing on traces in a trace algebra (def. 4.26, p. 89)

Symbol	Decorations	Denotes
T	primes, integer sub's	trace structure of a trace algebra (def. 2.15, p. 33)
t	primes, integer sub's	clock (p. 111) or time stamp
$\mathcal{T}$	primes, mnem. sup's	domain of trace structures of a trace algebra (def. 4.48, p. 100)
u	primes, integer sub's	sequence of actions or sets of actions (def. 3.14, p. 69)
W	none	set of all signals (def. 2.1, p. 22)
w	primes, integer sub's	trace
X	primes, integer sub's	set of traces
x	primes, integer sub's	trace
Y	primes, integer sub's	set of traces
y	primes, integer sub's	trace

Bibliography

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In LICS90 [67], pages 414–425.

[2] R. Alur and D. Dill. Automata for modeling real-time systems. In M. S. Paterson, editor, *Automata, Languages, and Programming: 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, Warwick University, England, 1990. Springer-Verlag.

[3] R. Alur and T. A. Henzinger. A really temporal logic. In *30th Annual Symposium on Foundations of Computer Science*, 1989.

[4] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[5] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of . IEEE*, 79(9):1270–1282, Sept. 1991.

[6] D. L. Black. On the existence of fair delay-insensitive arbiters: Trace theory and its limitations. *Distributed Computing*, 1(4):205–225, 1986.

[7] S. D. Brookes. *A Model for Communicating Sequential Processes.* PhD thesis, Oxford University, 1983.

[8] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In *NSF-SERC Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[9] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. Comput.*, C-35(12):1035–1044, 1986.

[10] R. E. Bryant, editor. *Third Caltech Conference on VLSI.* Computer Science Press, Inc., 1983.

[11] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In DAC91 [36].

[12] J. Brzozowski and C.-J. Seger. A unified framework for race analysis of asynchronous networks. *J. ACM*, 36(1):20–45, Jan. 1989.

[13] J. A. Brzozowski and C.-J. H. Seger. Advances in asynchronous circuit theory; part I: Gate and unbounded inertial delay models. *Bulletin of the European Association for Theoretical Computer Science*, 42:198–249, Oct. 1990.

[14] J. A. Brzozowski and C.-J. H. Seger. Advances in asynchronous circuit theory; part II: Bounded inertial delay models, MOS circuit design techniques. *Bulletin of the European Association for Theoretical Computer Science*, 43:199–263, Feb. 1991.

[15] J. R. Burch. A comparison of strict and non-strict semantics for lists. Master's thesis, Computer Science Department, California Institute of Technology, 1988. Technical Report CS–TR–88–12.

[16] J. R. Burch. Combining CTL, trace theory and timing models. In Sifakis [93].

[17] J. R. Burch. Modeling timing assumptions with trace theory. In ICCD89 [50].

[18] J. R. Burch. Verifying liveness properties by verifying safety properties. In Kurshan and Clarke [56]. Also in Springer-Verlag LNCS 531.

[19] J. R. Burch. Approximating continuous time. Presented at the IEEE Workshop on VLSI, Orlando, Florida, Feb. 1991.

[20] J. R. Burch. Using BDDs to verify multipliers. In DAC91 [36].

[21] J. R. Burch. Delay models for verifying speed-dependent asynchronous circuits. In *Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Princeton University, Mar. 1992.

[22] J. R. Burch. Delay models for verifying speed-dependent asynchronous circuits. In *Proceedings: IEEE International Conference on Computer Design*, Oct. 1992. To Appear.

[23] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In DAC91 [36].

[24] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the International Conference on Very Large Scale Integration*, Edinburgh, Scotland, Aug. 1991.

[25] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. Technical Report CMU-CS-91-195, School of Computer Science, Carnegie Mellon University, Oct. 1991.

[26] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In LICS90 [67].

[28] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[29] J. R. Burch and D. E. Long. Efficient boolean function matching. In *IEEE International Conference on Computer-Aided Design*, Nov. 1992. To Appear.

[30] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, Massachusetts Institute of Technology, 1987. Technical report MIT/LCS/TR-393.

[31] E. M. Clarke, J. R. Burch, O. Grumberg, D. E. Long, and K. L. McMillan. Automatic verification of sequential circuit designs. *Philosophical Transactions of the Royal Society of London, Series A: Physical Sciences and Engineering*, 339(1652):105–120, Apr. 15, 1992.

[32] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, editors, *15th Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, May 1990. Springer-Verlag.

[33] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Kozen [55].

[34] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.

[35] E. M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. *Annual Review of Computer Science*, 2:269–290, 1987.

[36] *28th ACM/IEEE Design Automation Conference*, 1991.

[37] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May/June 1989, Proceedings*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[38] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1988. Also appeared as [42].

[39] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*. MIT Press, 1988.

[40] D. L. Dill. Complete trace structures. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification, and Synthesis: Mathematical Aspects* volume 408 of *Lecture Notes in Computer Science*, Cornell University, July 1989. Springer-Verlag.

[41] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Sifakis [93], pages 197–212.

[42] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[43] J. C. Ebergen. A technique to design delay-insensitive VLSI circuits. Report CS-R8622, Centrum voor Wiskunde en Informatica, June 1986.

[44] Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.–Feb. 1990.

[45] T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Eighteenth Annual ACM Symposium on Principles on Programming Languages*, 1991.

[46] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. Technical Report TR 92–1263, Department of Computer Science, Cornell University, Jan. 1992.

[47] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Automata, Languages, and Programming: 19th International Colloquium*, 1992.

[48] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8), 1978.

[49] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[50] *Proceedings: IEEE International Conference on Computer Design*, Oct. 1989.

[51] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, SE-12(9):890–904, Sept. 1986.

[52] F. Jahanian, A. K. Mok, and D. A. Stuart. Formal specification of real-time systems. Technical Report TR–88–25, Department of Computer Sciences, University of Texas at Austin, June 1988.

[53] M. B. Josephs and J. T. Udding. An algebra for delay-insensitive circuits. In Kurshan and Clarke [56]. Also in Springer-Verlag LNCS 531.

[54] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* PhD thesis, Eindhoven University of Technology, 1989.

[55] D. Kozen, editor. *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.

[56] R. Kurshan and E. M. Clarke, editors. *Computer-Aided Verification, Proceedings of the 1990 Workshop*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1990. Also in Springer-Verlag LNCS 531.

[57] R. P. Kurshan. Analysis of discrete event coordination. In de Bakker et al. [37].

[58] R. P. Kurshan. Automata-theoretic verification of coordinating processes, March 30, 1992.

[59] R. P. Kurshan and K. L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 10(11):1356–1371, Nov. 1991.

[60] L. Lavagno, 1991. Personal Communication.

[61] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In DAC91 [36].

[62] I. Lee and S. B. Davidson. Generalized I/O with timing constraints. Technical Report MS–CIS–87–01, Department of Computer and Information Science, University of Pennsylvania, Jan. 1987.

[63] I. Lee and A. Zwarico. Timed acceptances: A model of time dependent processes. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, Proceedings of a Symposium, Warwick, UK, September 1988*, volume 331 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[64] H. R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR–15–89, Harvard University, Center for Research in Computing Technology, 1989.

[65] H. R. Lewis. A logic of concrete time intervals. In LICS90 [67], pages 380–389.

[66] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles on Programming Languages*, 1985.

[67] *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[68] N. A. Lynch. Multivalued possibilities mappings. In de Bakker et al. [37].

[69] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*. The Association for Computing Machinery, Inc., Aug 1987. Also, MIT/LICS/TR–387, April 1987.

[70] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. In Kozen [55].

[71] A. J. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings: IEEE International Conference on Computer Design*, Oct. 1987.

[72] A. J. Martin and J. R. Burch. Fair mutual exclusion with unfair P and V operations. *Inf. Process. Lett.*, 21:97–100, Aug. 1985.

[73] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, Mar. 1989.

[74] A. Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[75] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[76] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[77] B. Mishra. *Some Graph-Theoretic Issues in VLSI Design*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1985.

[78] I. Moon, G. J. Powers, J. R. Burch, and E. M. Clarke. Automatic verification of sequential control systems using temporal logic. *American Institute of Chemical Engineers Journal*, 38(1):67–75, Jan. 1992.

[79] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: an algebra for timed processes. Technical Report RT–C16, Project SPECTRE, IMAG, Grenoble, France, 1990.

[80] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K. G. Larsen and A. Skou, editors, *Computer-Aided Verification, Proceedings of the 1991 Workshop*, volume 575 of *Lecture Notes in Computer Science*, 1992.

[81] J. S. Ostroff. Automated verification of timed transition models. In Sifakis [93], pages 247–256.

[82] J. S. Ostroff. *Temporal Logic for Real-Time Systems.* Research Studies Press, 1990.

[83] V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, Feb. 1986.

[84] G. M. Reed. A hierarchy of domains for real-time distributed computing. In M. Main, editor, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science.* Springer-Verlag, 1989.

[85] G. M. Reed and A. W. Roscoe. Analysing tm_{fs}: a study of nondeterminism in real-time concurrency. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, volume 491 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[86] G. M. Reed, A. W. Roscoe, and S. A. Schneider. CSP and timewise refinement. In J. M. Morris and R. C. Shaw, editors, *Fourth Refinement Workshop*, Cambridge, England, 1991. Springer-Verlag.

[87] M. Rem, J. L. A. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In Bryant [10].

[88] A. W. Roscoe. *A Mathematical Theory of Communicating Processes.* PhD thesis, Oxford University, 1982.

[89] S. A. Schneider. *Correctness and Communication in Real-Time Systems.* PhD thesis, Oxford University, 1990. Published as technical monograph PRG-88.

[90] C.-J. Seger. A bounded delay race model. In *IEEE International Conference on Computer-Aided Design*, 1989.

[91] C.-J. H. Seger. *Models and Algorithms for Race Analysis in Asynchronous Circuits.* PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1988. Research Report CS-88-22.

[92] C. L. Seitz. System Timing. Chapter 7 in [75], 1980.

[93] J. Sifakis, editor. *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science.* Springer-Verlag, June 1989.

[94] M. Tuttle, M. Merritt, and F. Modugno. Time constrained automata. Unpublished manuscript, Aug. 1988.

[95] Y. Wang. Real-time behaviour of asynchronous agents. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90 : Theories of Concurrency—Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[96] Y. Wang. CCS+TIME = an interleaving model for real time systems. In J. L. Albert, B. Monien, and M. R. Artalejo, editors, *Automata, Languages, and Programming: 18th International Colloquium*, volume 510 of *Lecture Notes in Computer Science*, Madrid, Spain, 1991. Springer-Verlag.

[97] P. Wolper. Temporal logic can be more expressive. In *22nd Annual Symposium on Foundations of Computer Science*, pages 340–348, 1981. Also appeared as [98].

[98] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

Index of Theorems, etc.

Index