A UNIFIED APPROACH TO HARDWARE VERIFICATION THROUGH A HETEROGENEOUS LOGIC OF DESIGN DIAGRAMS

Kathryn Fisler

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

August 1996

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

K. Jon Barwise
(Principal Advisor)

Steven D. Johnson
(Principal Advisor)

Daniel P. Friedman

June 24, 1996

Gerard Allwein

Acknowledgements

This dissertation reflects the influence of three advisors: my research advisors, Jon Barwise and Steve Johnson, and my Bell Laboratories mentor, Bob Kurshan. Thank you Jon for your infectious belief in diagrammatic reasoning and your patience throughout this project. Thank you Steve for keeping me thinking about how different methodologies relate to one another and for directing me towards the diagrammatic reasoning seminar that started it all. Thank you Bob for taking the time to help me focus my ideas through language containment, for supporting my extended stay at Bell Labs, and for helping me learn how to defend my research through our many fruitful arguments.

Special thanks to Gerry Allwein, the Visual Inference Laboratory "den mother", for always having an open door and ready feedback. Thanks also to Dan Friedman for being my fourth reader, to Jonathan Mills for substituting on my oral examination committee, and to Randy Bramley for his advice and encouragement. A special note of appreciation goes to the department's support staff, both administrative and technical, for always addressing problems and concerns efficiently and with a smile.

I have been fortunate to have financial support during this project from the Visual Inference Laboratory at Indiana University and from the AT&T Foundation through the PhD Fellowship Program. I am grateful for the stipend and travel support offered by both organizations.

The faculty in the Williams College Department of Computer Science have continued to be a source of inspiration and sound advice during my years in graduate school. I only hope that I can give to future students what they have always given me, both as teachers and as mentors.

Thanks to the many friends, near and far, who have lent open ears, sent email, and kept me laughing: Amy Butler Greenfield, Dom Grillo, Liz Johnson, Kate Keahey, Tom Loos, Lowell Vaughn, and PJ Weingartner. A special thanks to my family for all the support and encouragement they provided, especially during this last year as I completed this dissertation from home.

Finally, to Shriram, my best critic and my best friend, a simple thank you. This work reflects your presence in innumerable ways.

Abstract

Designers use a variety of notations, some of them diagrammatic, while developing systems. Unfortunately, verification tools mainly support single, textual logics. This gap between design practice and verification methodology makes formal verification difficult for designers to apply. Properly formalized, however, diagrams are just as suitable for formal proof as conventional textual notations. Furthermore, *heterogeneous* logics, those with multiple syntactic representations, support formal interactions between diagrammatic and textual notations.

This dissertation defines HHL, a heterogeneous and diagrammatic logic of six hardware design notations: circuit diagrams, state machines, timing diagrams, CTL, second-order logic, and regular expressions over timing diagrams and linear temporal logic formulas. HHL timing diagrams express symbolic as well as concrete timing constraints, generating a class of languages that cuts across the traditional Chomsky hierarchy but is not fully context sensitive.

HHL's semantic model is language-based, supporting verification by language-containment testing. Its deductive framework extends this approach beyond the regular languages supported by existing tools, provides a syntactic view of language-containment, and naturally integrates algorithmic and interactive verification techniques. The dissertation also proves that containment of a regular language in a non-regular timing diagram language is decidable, establishing that algorithmic finite-state verification is extendible to timing diagram properties. Worked examples demonstrate the potential of this foundational work to bring computer-aided verification closer to design practice.

Contents

List of Figures

1

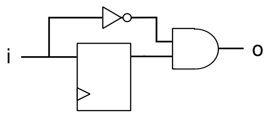
Introduction

Industry constantly strives for cost-effective, reliable, and time-efficient design methodologies. *Formal methods*, a collection of mathematically rigorous design techniques, advertises these benefits, yet industry remains skeptical. A recent IEEE Computer roundtable discussion on this skepticism cited several variations on an underlying problem: incompatibility between industrial practice and formal methods tools [73]. On the application level, tools often lack adequate support for industrial-style problems. On the usability level, designers find formal notations and cumbersome and unnatural. Although both are widely acknowledged problems, the former seems to receive more research attention than the latter.

Usable notation is essential for formal methods. Formal methods is about systematic approaches to design; design is inherently a reasoning process. Representation's role in facilitating reasoning is increasingly recognized [9, 80]. Notation and tools should therefore reflect how designers reason about systems in practice. Few designers reason using mathematical logic, a common basis for tools. Instead, they rely on a variety of notations, some diagrammatic, such as circuit diagrams and timing diagrams. This dissertation investigates the rigorous use of diagrammatic and heterogeneous representations for reasoning about designs.

Diagrams are a natural notation for certain types of hardware reasoning. Typically, hardware reasoning investigates how values flow through systems. Clear representations are therefore needed for connectivity within a system and for patterns of signal changes. Consider two examples: a single-pulser implementation shown as a circuit diagram and a second-order logic formula in Example 1, and a handshaking protocol shown in predicate logic, linear temporal logic, and a timing diagram in Example 2. In each example, the various representations depict the same information. The temporal logic representation uses the standard operators: next time ($\bigcirc$), henceforth($\Box$), eventually ($\Diamond$), and until (U). The timing diagram notations are defined formally in Chapter 2; intuitively, the arrows denote temporal ordering.

Example 1 A single pulser converts each finite duration input pulse into a unit duration output pulse.



$SP(i,o) \equiv \exists x \exists y : delay(i,y) \wedge not(i,x) \wedge and(x,y,o)$

$delay(I,O) \equiv \forall t : O(t+1) = I(t)$

$not(I,O) \equiv \forall t : O(t) = 1 - I(t)$

$and(I_1, I_2, O) \equiv \forall t : O(t) = I_1(t) * I_2(t)$

Example 2 In a four-phase handshaking protocol, whenever a request signal R goes high, it stays high until an acknowledge signal A goes high. A remains high until R recognizes the acknowledgement by going low. R then remains low until A also goes low.

Predicate Logic:

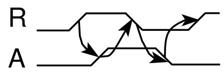
$$
\begin{aligned}
&\text{Handshake}(R,A) \equiv \\
&\quad R \neq A \wedge (\forall t)\ (\neg R(t) \wedge R(t+1)) \rightarrow \\
&\qquad (\exists t_1\ t_2\ t_3)\ \ (t < t_1 \wedge t_1 < t_2 \wedge t_2 < t_3 \wedge \\
&\qquad\quad (\neg(\exists t_4)\ \ ((t < t_4 \wedge t_4 < t_1 \wedge \neg R(t_4)) \vee \\
&\qquad\qquad (t_1 < t_4 \wedge t_4 < t_2 \wedge \neg A(t_4)) \vee \\
&\qquad\qquad (t_2 < t_4 \wedge t_4 < t_3 \wedge R(t_4)))) \wedge
\end{aligned}
$$

$$(\exists t_5)\ \ ((t_5 > t_3) \wedge (R(t_5) \rightarrow \neg(\exists t_6)\ \ (t_3 < t_6 \wedge t_6 < t_5 \wedge A(t_6)))))$$

Temporal Logic:

$$\Box[(\neg R\ \wedge\ \neg A) \rightarrow \neg A \mathsf{U}\ (R \wedge \Diamond A \wedge (R \mathsf{U}\ (A \wedge \bigcirc(A \mathsf{U}\ (\neg R \wedge \bigcirc(\Diamond R \rightarrow (\neg R \mathsf{U} \neg A)))))))]$$

Timing Diagrams:



In each example, the diagrams represent certain information implicitly that the sentential formulas must state explicitly: connectivity in the circuit diagram and event ordering in the timing diagram. As a result, the diagrams are more compact, more readable, and likely less subject to misinterpretation than the formulas. The latter two points are crucial: more readable implies that the designer spends less time reinterpreting the information while reasoning; less subject to misinterpretation implies greater probability that the intended specification is implemented or verified.

Despite these benefits, the research community has traditionally viewed diagrams as unsuitable for formal methods. Avra Cohn cited difficulties arising from diagrammatic specifications of the Viper chip.

> [Viper's] block-level model was given partly formally and partly pictorially (as was natural). Combining these two parts required both ingenuity and some guesswork. The guesses were based on the coincidence of line names, on the names of bound variables in function definitions, and on annotations in the text of the definitions. Before verification can be meaningfully applied in such cases, a fully formal description must be produced. [24]

Cohn's difficulties arose from attempting formal analysis of an unformalized notation. Diagrams, however, are not inherently informal. Diagrammatic notations can be formalized and then used rigorously, in the same manner as sentential notations.

John Rushby, in his excellent overview of formal methods, acknowledged designers' interests in diagrammatic specification languages and their potential formalization, but cited a drawback to their use in tools:

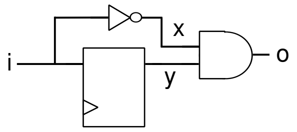
> The problem with [expressing formal specifications in diagrammatic or tabular notations] is that while diagrams and tables may be convenient ways to explain certain types of specification ... different problem domains lend themselves to different types of diagrams or tables ... a formal specification method built around a particular diagrammatic or tabular notation may have rather restricted application, and limited mechanized support for general forms of analysis. [72]

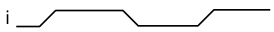
Rushby's argument misses a key point regarding how diagrams are used in practice: they are rarely used in isolation. As the examples above demonstrate, different diagrammatic representations are used to exploit different features of a particular problem domain. Similarly, diagrams are often used in conjunction with sentential information. Formally integrating multiple diagrammatic and sentential representations in one reasoning system would provide both the generality of sentential representations and the specialization and clarity of diagrams.

This dissertation develops a *heterogeneous logic* called HHL that integrates diagrammatic and sentential representations for hardware formal methods, particularly verification. Barwise and Etchemendy proposed heterogeneous logics to capture how people reason with such combinations of representations [8]. Heterogeneous logics do not treat diagrams as interfaces to sentential logics; each representation is given a semantics with respect to a common model through which multi-representational inference rules are defined.

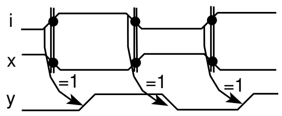
Treating diagrams as rigorous logical objects, rather than as mere interfaces to sentential logics, provides a novel framework for studying how people reason about designs in practice. Designers sometimes reason diagrammatically. This is evident from Avra Cohn's quote as the Viper was specified partially in diagrams and the designers must have reasoned about the design while developing it. Diagrammatic reasoning is powerful and intuitive because it retains a connection to the problem domain throughout the proof, as demonstrated in Example 3.

Example 3 Consider again the single pulser, introduced in Example 1. A single pulser must produce a unit-duration output pulse for each finite duration input pulse. Establishing that the proposed circuit diagram implements this behavior might proceed diagrammatically as follows. First, recall the circuit diagram and assume a pulse on the input.

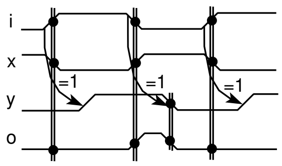




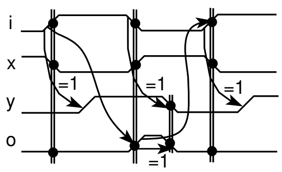
Knowing the behavior of inverters and unit delay elements, the diagram can be extended with waveforms for x and y, where the vertical lines denote event synchronization and the equalities on the arrows denote bounds on the time passing between events.

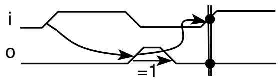


Similarly, a waveform for o can then be inferred.



The diagram contains sufficient information to finish the proof. Making some temporal information explicit then removing some unnecessary information, the resulting diagram verifies the behavior of the circuit diagram.





This proof is formal and sound in HHL; it appears with references to particular inference rules in Chapter 5. The proof is by natural deduction, not simulation. Simulation suffers from scalability problems on complex designs; the possible input patterns become too numerous to cover in a reasonable amount of time. Deductive diagrammatic systems, such as the one used here, provide the feel of simulation within the framework of a formal logic.

In contrast to the proof in Example 3, the corresponding sentential predicate logic proof has no apparent connection to the problem domain and appears to be mainly an exercise in symbol manipulation; the sentential proof is also longer and must assign explicit temporal variables to each event. The input waveform assumption alone, for example, is expressed sententially as

$$(\exists\, t_1\ t_2\ t_3)(t_1 < t_2 \wedge t_2 < t_3 \wedge \neg i(t_1)\ \wedge (\forall t_4)((t_4 > t_1 \wedge t_4 \leq t_2) \rightarrow i(t_4)) \wedge \\ (\forall t_5)((t_5 > t_2 \wedge t_4 \leq t_3) \rightarrow \neg i(t_5)) \wedge i(t_3+1))$$

with similar expressions derived for the other waveforms.

This demonstration motivates using diagrammatic representations in formal methods, but it does not justify a heterogeneous logic approach over one in which the diagrams are translated into and manipulated through sentential logics. On a philosophical level, heterogeneous logic gives a more accurate model of how people reason in practice; logic, the science of reasoning, should be extended to account for such practice. On a practical level, translation can be computationally expensive and often results in larger, less readable expressions. This is particularly true for timing diagrams. Source-error correlation can be difficult; tools must reflect errors back into the diagram or users must extricate errors from their sentential depictions. Most crucially, translations are error-prone and hence should be proven to be semantics-preserving. Diagrams must therefore be assigned a formal semantics *before* translation; in other words, most of the work required to create a diagrammatic logic is also required to create a proper diagrammatic interface.

Developing HHL involves several steps:

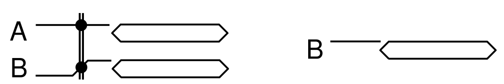
1. Choosing the representations to support.
2. Formalizing their syntax and semantics.
3. Defining rules of inference and a proof theory.
4. Investigating its suitability for formal methods.

Section 1.1 provides an overview of HHL and addresses Step 1. Step 2 is handled in Chapter 2. Chapter 3 presents several results regarding timing diagrams that are useful in addressing Step 3, which is covered in Chapter 4. Step 4 is addressed through examples in Chapter 5.

1.1 Heterogeneous Hardware Logic

HHL integrates six syntactic representations: circuit diagrams, state machines, timing diagrams, monadic second-order predicate logic, CTL (computation tree logic), and CTR (calculus of temporal representations). The first three are diagrammatic representations commonly used in design. Second-order logic and CTL are common sentential logics for hardware verification.

CTR is a heterogeneous calculus of regular expressions over timing diagrams and certain temporal logic formulas. Some temporal statements, such as the handshaking protocol in Example 2, seem more clearly expressed using timing diagrams rather than temporal logic. Others, such as the "strong until" operator of temporal logic, seem clearer in temporal logic. "Strong until", written $A \mathsf{U} B$, says that either B is true now or it becomes true in the future and A is true until B becomes true. A disjunction of two timing diagrams is needed to capture its definition. The stretched diamonds represent "don't care" regions in the timing diagram. The diagram on the left covers the case when B is initially low and the one on the right covers the case when B is initially high.



CTR combines timing diagrams and temporal logic formulas in single temporal statements, thereby allowing individual subformulas to be expressed as naturally as possible. Additional details are given in Chapter 2, Section 2.2.6.

HHL's semantics provides flexible support for verification. Verification logics differ with respect to their expressive power, and hence, their degree of automatability. Expressively powerful logics, such as second-order logic, are undecidable; user interaction is therefore required to perform proofs in these logics. Alternatively, proofs in decidable logics, such as LTL [68] and CTL [23], require little to no user interaction at the expense of expressiveness. The terms *theorem-proving* and *model-checking* refer to interactive and automated verification, respectively. Ideally, verification tools should provide a strong mix of both theorem-proving and model-checking. Model-checkers save substantial amounts of user-time on decidable problems, while theorem-provers provide the flexibility required for undecidable problems and problems that are computationally intractable although theoretically decidable.

HHL naturally integrates theorem-proving and model-checking through its *system language* semantics. A word in a system language is a sequence of value assignments that can appear on the signals of a system; the set of all such words for a particular system forms its language. Each syntactic representation in HHL can be viewed as restricting a system's language; intuitively, the semantic model of an HHL formula is the language of all words that meet its restrictions.

Language containment provides a framework for verification: a property holds of a system iff the language of the system is contained in the language of the property [55]. Containment can similarly be used to test whether one design or specification is a refinement of another. Inference rules in HHL can be viewed as small refinement steps: an inference rule from a set F of formulas to a formula f is sound iff the intersection of the languages modeling elements of F is contained in the language modeling f. HHL proofs therefore establish language containment at the syntactic, rather than the semantic, level.

The language containment test is decidable for regular languages, but not for context-free languages [45]. HHL's timing diagrams, and hence CTR, can express non-regular properties. Interactive verification is therefore required for some proofs in HHL; proofs of regular properties, however, can be handled automatically. Fortunately, many timing diagrams corresponding to regular properties can be recognized syntactically (see Chapter 3). HHL therefore provides a syntactic means of determining which style of verification is required. This results in a clean integration of the two techniques: theorem-proving is applied until the problem is reduced to something recognizably decidable, at which point model-checking can be invoked.

1.2 Contributions of the Dissertation

This dissertation illustrates that diagrammatic and heterogeneous hardware representations can be used formally to reason about systems; furthermore, they support a natural integration of various verification techniques. This work makes several specific contributions to both diagrammatic reasoning and formal methods research:

- A formalization of the syntax and semantics of common design representations, which supports the rigorous use of these representations in formal methods (Chapter 2). HHL also extends work in heterogeneous reasoning systems by supporting multiple diagrammatic representations.

- Decision procedures for testing containment of a regular language in a non-regular timing diagram language (Chapter 3). The procedures extend the applicability of certain automated verification applications past regular languages into context-free and context-sensitive timing diagram languages.

- A correctness-preserving translation from a class of timing diagrams into CTL, which enables these timing diagrams to be used rigorously as interfaces to existing tools while tools based on HHL are being developed (Chapter 3).

- A proof theory of diagrammatic and heterogeneous design representations (Chapter 4). This theory provides both the foundation for building more designer-friendly formal methods tools and a syntactic means of establishing language containment and other language related properties of interest in verification.

1.3 Related Research and Existing Tools

1.3.1 Diagrammatic Reasoning and Diagrammatic Representations

Barwise and Etchemendy have done much of the pioneering research in diagrammatic and heterogeneous logics [9]. Their *Hyperproof* system was the first implemented heterogeneous and diagrammatic reasoning system [7]. Designed for teaching introductory logic and reasoning, *Hyperproof* provides two representations of information: a grid on which blocks of differing sizes and shapes may be placed, and a sentential, first-order predicate logic. Preliminary studies suggest that the heterogeneous approach of *Hyperproof* is beneficial in instructional settings: students reportedly cover more material and retain conventional logic material better in a *Hyperproof* class than in a traditional sentential logic class [80]. Additional heterogeneous logics have been developed by Shin for Venn diagrams [76] and Luengo for geometry diagrams [58]. Glasgow, Narayanan, and Chandrasekaran's volume on diagrammatic reasoning covers other related research in the field, including some of the cognitive factors associated with diagrammatic representations [34].

Barker-Plummer and Bailin have investigated how diagrammatic information can guide heuristic theorem provers. Motivated by the use of diagrams in mathematics, their GROVER/& theorem-prover uses diagrammatic information to guide which heuristics to apply and to suggest lemmas on which the proof might rely. Several results that are difficult for purely sentential heuristic theorem-provers, such as the Diamond Lemma and the Schröder-Bernstein theorem, have been proven using GROVER/& [5, 6]. Currently, GROVER/& does not work directly with diagrams; the system takes sententially represented, structural information that would be derived from a diagram. Their work, however, establishes that there are logical benefits to working with the types of information typically represented by diagrams.

Manna and Pnueli promote diagrams to guide human, as opposed to heuristic, proof development [16]. Their *verification diagrams* form a Hoare-logic style diagrammatic logic for conveying structural details of complex proofs about transition systems. The diagrams are used in Manna *et al.*'s STeP prover [60] to visualize verification conditions of the form $\{\varphi\}\tau\{\psi\}$, which state that whenever τ is taken from a state satisfying φ, ψ holds in the following state. These conditions abstract over computation paths in a transition system, thus displaying the essence of the proof. The diagrams are constructed by the user, with an underlying tool deriving the verification conditions that correspond to the diagram. Verification diagrams are used in a substantially different way than the diagrams in HHL. Verification diagrams help the user organize information about a proof — they are a tool with which a user reasons. In HHL, diagrams are the objects *about* which a user reasons. HHL attempts to address how hardware diagrams are used to reason about design. Verification diagrams arise from proof practice, not design practice.

The Real-Time Graphical Logic Toolset [71] provides a diagrammatic logic with a proof checker for a graphical interval temporal logic. Formulas in the logic express conditions over intervals delimited by states in which cited expressions hold. Proofs establish relationships between graphical interval logic formulas; the proofs are evaluated through a verifiable translation into a sentential interval logic and a decision procedure for the sentential logic. The toolset does not provide a means for testing graphical interval logic formulas against system designs; as a result, it is not a full-fledged verification system. However, it is an alternative, formal treatment of diagrammatic representations for reasoning about relationships between temporal properties.

Various diagrammatic specification languages have been proposed for system design. Harel's StateCharts express hierarchical state machines by allowing nesting of finite automata within states [42, 43]. They are widely used for describing reactive systems. However, StateCharts do not comprise a diagrammatic logic; while they have an operational semantics that allows them to be executed, no inference rules have been defined on them. Chu first proposed *Signal-Transition Graphs* as a graphical representation for specifying asynchronous circuits [22]; their use was extended to verification by Moon [64]. No inference rules have been defined on these graphs. Their application to asynchronous systems also sets them apart from HHL, which treats synchronous systems. Sheeran's Ruby is a relational calculus for circuit design that uses pictorial icons to specify circuit relations, though the icons are not used to reason about circuits [50, 51].

Several verification tools have supported diagrams through translation into sentential logic. The LAMBDA theorem-prover provides a circuit schematic interface for specifying systems [31]. Tool support for timing diagrams has increased in recent years. Schlör has used timing diagrams to state properties for VHDL design verification [74], and Grass has used them to specify synthesis constraints [30]. Both of these efforts are part of larger project aimed at producing a designer-friendly, integrated verification and synthesis environment.

Khordoc, Cerny, and their collaborators have used timing, and timing-like, diagrams in numerous ways for hardware reasoning. They have annotated timing diagrams with sentential behavioral assertions for interface specification [52]; these diagrams generate event graphs that are used for the actual analysis. Syntactic realizability conditions have been developed for these diagrams [20], thus reducing their role as pure graphical interface tools. Their *action diagrams* use a hierarchy of diagrammatic notations for behavioral modeling: timing diagram notations are combined using graphical notations for concurrency, concatenation, looping, and choice [53]. Action diagrams are given an operational semantics through translation to timed automata [10]

Message sequence charts are a popular diagrammatic formalism for specifying communication traces between protocols. The ITU message sequence chart standard uses BNF grammars to specify the correlation between a message sequence chart and a sentential system description language [49]. Tools for manipulating message sequence charts for system design are under development by Alur, Holzmann, Peled, and Kernighan [1]. At present, no inference rules have been developed for message sequence charts; they are more a tool for simulation than for verification.

1.3.2 Automated Verification

Model-checkers are extremely useful, despite their limited expressibility. Temporal logic model checking, proposed by Clarke and Emerson, tests whether a temporal logic formula is true over a finite Kripke structure [23]. McMillan proposed representing the state space of the structures symbolically, to reduce the size of the state space and thereby increase the size of the verifiable models [18, 61]. Efficient symbolic model-checking algorithms arise from the decision procedures for *binary decision diagrams* (BDDs), a compact representation of the structure of boolean formulas [17]. Language containment is another form of model-checking using automata to capture systems and properties [55]. Properties can be translated from temporal logic into automata using techniques described by Vardi and Wolper [11, 85]. Automata-theoretic treatments of verification work with automata over infinite words or trees, known as ω-automata [83].

HHL's semantics are inspired by language containment, based on experience using the COSPAN verification system [82]. COSPAN demonstrates that language models support elegant and varied techniques for handling computationally intractable verification problems. These techniques support both top-down and bottom-up approaches to design and verification, resulting in a powerful and flexible framework for finite-state systems and regular properties; COSPAN, however, supports only regular properties. Several system properties, such as a producer-consumer problem stating that the number of consumptions does not exceed the number of productions, cannot be expressed using regular languages [21]. Mathematically, COSPAN and HHL also differ in that COSPAN only establishes language containment semantically, while HHL provides a framework for studying language containment syntactically.

1.3.3 Interactive Verification

The term *theorem-prover* can refer to two different types of systems: *proof-checkers*, which check that proof steps specified by a user are valid, and *heuristic theorem provers*, which attempt to locate proofs using various search techniques. Most systems display aspects of both types. HOL [36], PVS [79], Larch [39], and Nuprl [57] are primarily proof-checkers that are popular in formal methods. Boyer and Moore's heuristic theorem-prover NQTHM [13] has been used in several verification efforts [14, 46]. Effective theorem-prover based verification relies on a variety of abstraction techniques, as discussed by Melham [62].

Theorem-provers are generally applied to designs after they have been developed. Interactive verification can also be *derivational* in nature, where a correct circuit is obtained by transforming a specification into an implementation; this approach is often referred to as *synthesis*. Kumar *et al.*'s recent survey of synthesis techniques provides a good general introduction and overviews several formal synthesis systems [54].

1.3.4 Integrated Verification

Several systems have attempted to integrate automated and interactive verification. PVS includes a mu-calculus decision procedure in its higher-order logic deductive framework [66]. STeP (the Stanford Temporal Prover), provides a combination of automated verification and invariant generation techniques, guided through the use of Manna and Pnueli's verification diagrams [60]. Schlör and others have designed a VHDL prover that uses model-checking for behavioral specifications, combining the results with interactive techniques in a compositional verification system [74]. Kurshan and Lamport used an integration of COSPAN and TLA to verify a multiplier circuit [56]. COSPAN itself can be considered an integrated system, due to the number of user-guided techniques it supports. Derivational techniques were integrated with tautology checking by Bose, Johnson, and Pullela [12].

Integrations have also been studied of various interactive techniques, or various automated techniques. Miner and Johnson used a combination of theorem-proving and derivation techniques in the verification of a clock-synchronization circuit [63]. HSIS, a verification tool developed by the Berkeley CAD group, combines temporal logic model checking and language containment [4]; their newer system, VIS, extends this integration to include synthesis techniques [15].

1.3.5 Decision Procedures and Non-Regular Languages

Several researchers have investigated the decidability of problems on non-regular languages. Ginsburg shows the decidability of language emptiness, containment, and equivalence for bounded languages, those of the form $w_1{}^*w_2{}^* \dots w_n{}^*$, such as a^nb^n [33]. Ibarra *et al.* extended these results to the general class of languages accepted by two-way, one counter machines [48].

Harel and Raz investigated non-regular extensions of PDL (Propositional Dynamic Logic) programs, identifying three classes of non-regular languages such that PDL extended with any language from one of the classes remains decidable [44]. Their characterizations are based partially on the structural features of the machines accepting the languages.

Alur, Henzinger, and Vardi have studied timed automata with symbolic, as opposed to strictly concrete, real-time constraints [3]. Their work is the most closely related to the decidability results in Chapter 3. They extended a timed automata model with sets of clocks that all progress at the same rate. Transitions in their model may be conditional on the clocks falling within certain (possibly symbolic) ranges; transitions are also able to reset clocks. They established that language emptiness is decidable for parametric timed automata when only one clock is used, is undecidable when three clocks are used, and is still an open problem when two clocks are used. The temporal variables in HHL's timing diagrams are most closely related to their symbolic parameters; their clocks have closer correspondence with the amount of overlap between constraints in HHL timing diagrams. The results are therefore not directly comparable, although both are based on Ibarra's results.

2

Syntax and Semantics

Formalizing the syntax of HHL is fairly straightforward. The logic uses the standard syntax for each sentential representation and the conventional notations for circuit diagrams and state machines. Timing diagrams represent conventional types of information, but lack conventional notations; HHL provides notations for each conventional type of information.

Choosing a suitable semantic model is more difficult. Heterogeneous logics support inference between information given in different representations. Relationships must therefore exist between the semantic models associated with each representation. Three approaches to defining these relationships are:

1. Translate each representation into an underlying logic and define all inference rules on the underlying logic.
2. Define the semantics of each representation relative to a common semantic model and define inference rules on the basis of this model.
3. Give each representation its own semantics, provide formal links between the semantics, and define inference through these links.

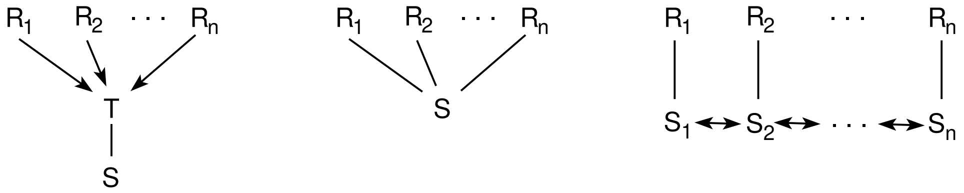


Figure 2.1: Three approaches to establishing semantic relationships in heterogeneous logics. Each R_i denotes a representation, S and each S_i denote a semantics, and T denotes a target representation in a translation. Single-headed arrows denote translation, double-headed arrows denote semantic linkage, and plain lines denote the assignment of semantics to syntax.

Figure 2.1 depicts these options. The first is a form of language translation. Existing heterogeneous logics use the second. The third parallels the approach taken in logical frameworks [19].

Heterogeneous logic research advocates choosing semantic models and notations that are *homomorphic*, meaning structurally similar to one another and to any physical objects they describe [8].[1] This is fairly simple in a heterogeneous logic with a single diagrammatic representation. For HHL, however, it is difficult to find a model homomorphic to both circuit diagrams and timing diagrams. Circuit diagrams describe hardware at the state level. A model homomorphic to them would depict states and the conditions required to transition from one state to another. Timing diagrams describe patterns of signal behavior, but abstract away from the states in which the patterns occur. A homomorphic model for timing diagrams would depict events and their possible sequences, but not states. A model homomorphic to both types of diagrams would therefore need to simultaneously describe and suppress state-level information.

As an alternative, HHL employs two semantic models: a state-based device model and an event-based language model that abstracts over state-based information.[2] The device model is homomorphic to circuit diagrams and is also used to model state machines and second-order logic. The language model reflects changes on signals over time; it is not homomorphic to timing diagrams because it does not directly represent some of the timing diagram notations. It is, however, close enough to be a suitable semantic model for timing diagrams and CTR. CTL's semantics are usually

[1] This applies mainly to diagrammatic representations, as sentential representations are generalized across domains.
[2] The term *device* is used instead of *circuit* to avoid confusion between circuit diagrams and the semantic model.

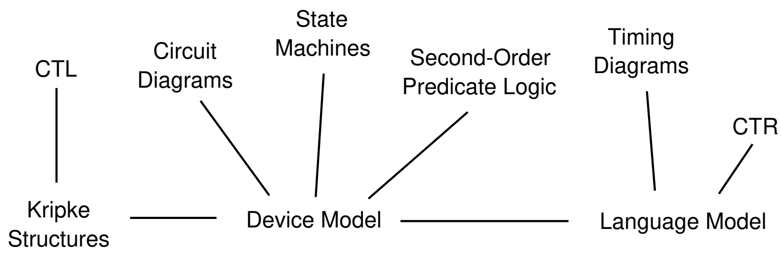


Figure 2.2: Overview of the semantic structure of HHL.

defined relative to finite Kripke structures. A linkage relationship from devices to Kripke structures ties CTL into the logic. An abstraction relationship between the device model and the language model provides a sufficient framework for defining heterogeneous inference rules. Figure 2.2 summarizes the semantic structure of HHL.

Section 2.1 defines the semantic models and their interrelationships. Section 2.2 contains the syntax and semantics definitions for each representation in HHL, relative to the models in Section 2.1. Finally, Section 2.3 provides the definition of HHL based on the definitions in Section 2.2.

2.1 Semantic Models

2.1.1 Physical Devices

Physical devices can be viewed either statically, as an arrangement of components, or dynamically, as a computational machine. Accordingly, the device model is defined in two stages: *static devices* capture layout, and *dynamic devices* build on static devices to capture computational information. *Ports* and *wires* are primitive in these definitions. A port is a physical point of connection between a gate and its surrounding circuitry. Wires connect ports. Values are associated with ports, and any ports connected by a wire carry the same value at all times. The valid values on a port are 0 and 1. All devices contain two special ports known as the *power port* and the *ground port*, carrying constant values 1 and 0, respectively.

A series of preliminary definitions are given first, followed by the definition of a static device.

Definition 1

1. An *assignment* is a total function from a set of ports to $\{0, 1\}$. An assignment on a set of ports P will be called a P-assignment.

2. A *basic component* b is a tuple $\langle I, O, F, d\rangle$ where I and O are disjoint sets of ports called the input ports and output ports of b, respectively; F is a function which assigns each $p \in O$ a function F_p from I-assignments into $\{0, 1\}$; and d is a function from O to the non-negative integers called the output delay function.

3. A *gate* is a basic component $\langle I, O, F, d\rangle$ in which $d(p) = 0$ for all ports p in O. Any other basic component is called a *sequential element.*

4. A *wiring* is a function c from wires to sets of ports of cardinality at least two. Ports which are elements of $c(w)$ for the same wire w are said to be *wired together* or *connected* by w.

Basic components are restricted to having delay at most one. Encapsulated devices with longer delays are definable using the approach given in Section 2.2.7. Three basic components provide the foundation for devices: binary and gates, inverters, and unit delay elements. Their formal definitions reflect their respective standard behaviors.

Definition 2

1. An *inverter*, denoted $\mathsf{Inverter}(ip, op)$, is a gate in which $I = \{ip\}$ and $O = \{op\}$ for some ports ip and op and $F_{op}(i) = 1 - i(ip)$ for all I-assignments i.

2. A *unit delay element*, denoted $\mathsf{UnitDelay}(ip, op)$, is a sequential element in which $I = \{ip\}$ and $O = \{op\}$ for some ports ip and op and F_{op} is the identity function.

3. An *and gate*, denoted $\mathsf{AndGate}(ip_1, ip_2, op)$, is a gate in which $I = \{ip_1, ip_2\}$ and $O = \{op\}$ for some ports ip_1, ip_2, and op and $F_{op}(i) = i(ip_1) * i(ip_2)$ for I-assignment i.

Definition 3 A *static device* is a 5-tuple $D = \langle I, O, B, W, c \rangle$ satisfying the following conditions:

1. I and O are disjoint sets of ports providing the interface to D called the *interface input ports* and the *interface output ports*, respectively. $I \cup O$ is called the *interface ports.*

2. B is a set of basic components. The sets of ports for the elements in B must all be pairwise disjoint and disjoint from I and O. Any port for an element in B that lies in its component I is an *internal input port.* Any port for an element in B that lies in its component O is an *internal output port.* These ports are collectively called *internal ports.* The *ports* of D are the union of the interface ports and the internal ports.

3. W is a set of wires and c is a wiring with domain W and co-domain the powerset of the ports of D; this includes the interface ports of D as well as the ports of all gates and unit delay elements of D. Every non-input interface port of D appears in exactly one $c(w)$ and every input interface port of D appears in at most one $c(w)$. For all w, $c(w)$ contains exactly one internal output port or interface input port, and $c(w)$ contains at least one internal input port or output interface port of D.

Although any static device can be constructed physically, some have undesirable properties. For example, a device with a feedback loop in combinational logic may be computationally unstable. The next set of definitions restricts the class of static devices to exclude those with undesirable properties. Any device not excluded by the restrictions is called *well-connected.*

Definition 4 Given a static device $D = \langle I, O, B, W, c \rangle$, there is a *connecting step* from port p_i of D to port p_j of D, denoted $p_i \rightsquigarrow p_j$, under the following conditions:

1. If p_i is an internal input port of basic component b in B, then $p_i \rightsquigarrow p_j$ iff p_j is an internal output port of b.

2. If p_i is an internal output port or an input interface port, then $p_i \leadsto p_j$ iff p_j is an internal input port or an output interface port and p_i and p_j are wired together by some wire w in W.

If $p_i \leadsto p_j$, the basic component or wire that connects p_i and p_j is called the *connecting element* of p_i and p_j.

Notice that according to this definition, there can be no connecting steps from output interface ports to any other ports.

Definition 5 Given a static device D, there is a *connecting path* in D from port p_0 of D to port p_n of D (denoted $p_0 \leadsto^* p_n$) iff there exists a finite chain of connecting steps

$$p_0 \leadsto p_1 \leadsto p_2 \leadsto \ldots \leadsto p_n$$

There is a *feedback cycle* in D if there is a connecting path $p \leadsto^* p$ for some port p in D.

Definition 6 Let $D = \langle I, O, B, W, c \rangle$ be a static device and let G be a graph containing one node for each element of B. Let G contain an edge between the nodes corresponding to basic components b_1 and b_2 iff there exists a wire connecting a port of b_1 to a port of b_2. D is *well-connected* iff:

1. G is connected,
2. For each internal port in D there exists a connecting path from p to an element of O, and
3. Every feedback cycle in D contains some step $p_i \leadsto p_j$ such that the connecting element of p_i and p_j is a sequential element.

Optimizations on devices are generally performed by local transformations on subdevices. Although the transformations are not defined until Chapter 4, the following definition establishes how subdevices are isolated using monomorphisms between circuit diagrams.

Definition 7

1. Given basic components $b_1 = \langle I_1, O_1, F_1, d_1 \rangle$ and $b_2 = \langle I_2, O_2, F_2, d_2 \rangle$, a function $\theta : I_1 \cup O_1 \rightarrow I_2 \cup O_2$ is an *isomorphism between b_1 and b_2* iff
   (a) $\theta(I_1) = I_2$.
   (b) $\theta(O_1) = O_2$.
   (c) For each $p \in O_1$, $F_1(p)$ is equal to $F_2(\theta(p))$.
   (d) $d_1 = d_2$.
2. Given static devices $D_1 = \langle I_1, O_1, B_1, W_1, c_1 \rangle$ and $D_2 = \langle I_2, O_2, B_2, W_2, c_2 \rangle$, a function θ from the ports of D_1 to the ports of D_2 is a *homomorphism between D_1 and D_2* iff
   (a) $\theta(I_1) = I_2$.
   (b) $\theta(O_1) = O_2$.
   (c) For each $b_1 \in B_1$, there exists a $b_2 \in B_2$ such that the restriction of θ to the ports of b_1 is an isomorphism between b_1 and b_2.
   (d) For all ports p_a and p_b in D_1, p_a and p_b are wired together iff $\theta(p_a)$ and $\theta(p_b)$ are wired together.
3. Static devices D_1 and D_2 are *homomorphic* if there exists an homomorphism between them. Homomorphism θ from D_1 to D_2 is a *monomorphism* if θ is injective and an *epimorphism* if θ is surjective. If θ is bijective, D_1 and D_2 are *isomorphic*.

Definition 8 Let D be a static device.

1. Static device D' is a *subdevice* of D iff D' is well-connected and there exists a monomorphism from D' to D.
2. Let P and P' be disjoint subsets of the ports of D. If there exists a subdevice $D' = \langle I', O', B', W', c' \rangle$ of D such that $I' = P$ and $O' = P'$, then D' is said to be the *subdevice of D encapsulated by P and P'*.

When working with subdevices, it is often useful to talk about the portion of the circuit other than the subdevice. If $D' = \langle I', O', B', W', c' \rangle$ is a subdevice of $D = \langle I, O, B, W, c \rangle$, notation $D - D'$ represents the tuple $\langle I - I', O - O', B - B', W - W', c - c' \rangle$. $D - D'$ is not necessarily a well-connected device.

The above definitions capture the physical structure of devices. Capturing their computational behavior requires additional terminology. Computational behavior can be defined at multiple levels of granularity; this work considers the extremal levels: behavior as a function over all ports, and behavior as a function over only interface ports. Dynamic devices, defined below, directly support the former, but can be abstracted to support the latter.

Definition 9 A *dynamic device* is an ordered pair $\langle D, i \rangle$ where $D = \langle I, O, B, W, c \rangle$ is a static device and i is an assignment to the ports of D that is consistent with the values assigned by each gate in B; in other words, for all gates $g = \langle I_g, O_g, F, 0 \rangle$ in B and for all output ports p_o in O_g, $i(p_o)$ equals the value of $F(p_o)$ applied to the restriction of i to I_g.

The domain of the assignment in a dynamic device may contain ports that do not appear in the associated static device. This is necessary to support logic minimization in devices. Two devices may compute the same outputs using different numbers of input signals. Allowing assignments to give values to unused ports simplifies the definition of output equivalence between such devices.

A dynamic device $\langle D, i \rangle$ is *well-connected* iff D is well-connected. The structure of a well-connected device dictates the relationships between the values on its ports. Given a dynamic device and an assignment to its input interface ports, there is therefore a unique assignment indicating how the input values propagate through the device. This lemma describes how to compute this assignment and establishes its uniqueness.

Lemma 1 Given a well-connected dynamic device $\langle D, i \rangle$ for $D = \langle I, O, B, W, c \rangle$ and an assignment i' to the input ports of D, there is a unique assignment i'' satisfying the following conditions:

1. If p is an interface input port of D, then $i''(p) = i'(p)$.

2. If p is an output port to a basic component b in B with delay 1, then $i''(p)$ is F_p applied to the restriction of i to the input ports of b.

3. If p is an output port of some gate g in B, then $i''(p)$ is F_p applied to the restriction of i'' to the input ports of g.

4. If p is any other port, $i''(p) = i''(q)$ where q is the unique internal output port or input interface port wired to p.

Proof As constructed, i' is an assignment. Assume there are assignments i_1 and i_2 satisfying the listed conditions, but such that $i_1 \neq i_2$. Let p be a port in D such that $i_1(p) \neq i_2(p)$ and there is no port p' in D such that $i_1(p') \neq i_2(p')$ and p' lies on a path from an input interface port to p. By definition, p is not an interface input port since otherwise $i_1(p) = i'(p) = i_2(p)$. Similarly, p is not the output port of a unit delay element since otherwise $i_1(p) = i(q) = i_2(p)$, where q is the input port to that delay element.

If p is the output port of a gate g, then since gates are associated with deterministic functions, $i_1(q) \neq i_2(q)$ for some port q that serves as an input port for g. The existence of a path from an input interface port to q would violate the choice of p, so all paths to q must go through feedback loops. Since D is well-connected, by Definition 6 all paths to q must pass through some unit delay element. By definition, i_1 and i_2 must agree on all values on the output ports of unit delay elements. Since q must be a deterministic function of these delay outputs, i_1 and i_2 must agree on the value for q, and hence, the value for p.

If p is some other port, then $i_1(q) \neq i_2(q)$ where q is the unique output port or input interface port connected to p. This also contradicts the choice of p since q would then lie on a path from an input interface port to p. The constructed assignment is therefore unique.

□

Definition 10 Let $D_Y = \langle D, i\rangle$ be a dynamic device and let i' be an assignment to the interface input ports of D. The unique assignment i'' given by Lemma 1 is called the *derived assignment for* D_Y *given* i', and is denoted $D_Y[i']$. Dynamic device $\langle D, D_Y[i']\rangle$ is said to *follow from* D_Y, given i'.

The above definitions are sufficient for observing a device's behavior when one set of values is applied to the input ports. Iterating this process provides observational equivalence between devices over time.

Definition 11 An *assignment sequence* for a static device D is a sequence of assignments $i_1, i_2, \ldots$ to the interface input ports of D.

Definition 12 Let $D_Y = \langle D, i\rangle$ be a dynamic device. A *run* of D_Y is a sequence $r = \langle D_Y, D_{Y_1}, \ldots\rangle$ of dynamic devices such that there is an assignment sequence $i_1, i_2, \ldots$ of length one shorter than that of r and for each $j \geq 1$, D_{Y_j} is the dynamic device that follows from $D_{Y_{j-1}}$ given i_j. If the assignment sequence is of finite length k, the last dynamic device in the run is denoted by $\langle D, \langle i_1, \ldots, i_k\rangle\rangle$.

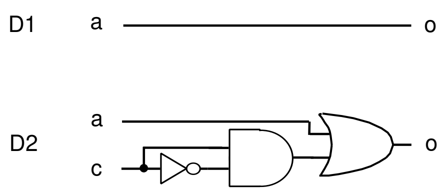
A run can be characterized by the sequences of states and outputs it reflects; both types of sequences are easily extracted from a run.

Definition 13 The *output* of a dynamic device $\langle D, i\rangle$ is the restriction of i to the interface output ports O of D. The output of a run $r = \langle D_{Y_0}, D_{Y_1}, \ldots\rangle$ of a dynamic device is a sequence $\langle O_0, O_1, \ldots\rangle$ of length equal to that of r, where each O_i is the output of D_{Y_i}.

Definition 14 The *state* of a dynamic device $\langle D, i\rangle$ is the restriction of i to the delay element output ports of D. The set of *possible states* of device D is the set of all possible assignments on the output ports of the delay elements in D; the possible states of a dynamic device are the possible states of the corresponding static device.

Devices with different physical layouts can compute the same function. Interface-level behavioral comparisons can identify such devices. This level of comparison is used extensively in hardware reasoning. The term *behavioral equivalence* henceforth refers to interface level comparison. Two types of behavioral equivalence are considered. In the first, two devices are behaviorally equivalent if they always produce the same outputs on the same inputs. In the second, two devices are behaviorally equivalent if one can always be driven into a state such that the two have the same outputs from that point onward. Two devices may be behaviorally equivalent yet have different numbers of input interface ports as shown in Example 4.

Example 4 Each of D_1 and D_2 computes the function $o = a$, but D_2 uses the additional input c.



In order to account for such differences, the definitions of behavioral equivalence are not symmetric when one device has more input ports than the other. This is required to guarantee that the assignments used in the definitions have sufficiently large domains to apply to both devices. The first two definitions account for behavioral equivalence: first a notion of behavioral indistinguishability is defined on dynamic devices, then this is used to define behavioral equivalence on static devices. The last three definitions define alignable equivalence on static devices, again using intermediate definitions on dynamic devices.

Definition 15

1. Dynamic devices $D_{Y_1} = \langle D_1, i_1 \rangle$ and $D_{Y_2} = \langle D_2, i_2 \rangle$ are *behaviorally indistinguishable* if D_1 and D_2 have the same output interface ports, the input interface ports of D_2 are a subset of the input interface ports of D_1, and for every assignment sequence I for D_1, the output of the run of D_{Y_1} under I is the same as the output of the run of D_{Y_2} under I.

2. Static devices D_1 and D_2 are *initializably equivalent* iff there exist static devices D_1' and D_2'

isomorphic to D_1 and D_2, respectively, such that for any assignment i_1 for D'_1 there exists assignment i_2 for D'_2 such that $\langle D'_1, i_1\rangle$ and $\langle D'_2, i_2\rangle$ are behaviorally indistinguishable and for any assignment i_2 for D'_2 there exists assignment i_1 for D'_1 such that $\langle D'_1, i_1\rangle$ and $\langle D'_2, i_2\rangle$ are behaviorally indistinguishable. If there always exist choices of i_2 (i_1) such that i_1 and i_2 agree on the values of all common input ports, D_1 and D_2 are *behaviorally equivalent*.

3. Static device D_1 can *simulate* dynamic device $\langle D_2, i_2\rangle$ if there exists a finite assignment sequence $i_{1_1}, \ldots, i_{1_k}$ such that $\langle D_1, \langle i_{1_1}, \ldots, i_{1_k}\rangle\rangle$ is behaviorally indistinguishable from $\langle D_2, i_2\rangle$.

4. Static device D_1 *unconditionally simulates* D_2 if for all assignments i_2 for D_2, D_1 can simulate $\langle D_2, i_2\rangle$.

5. Static devices D_1 and D_2 are *alignably equivalent* iff D_1 is isomorphic to some D'_1 which unconditionally simulates D_2 and D_2 is isomorphic to some D'_2 which unconditionally simulates D_1.

Proposition 1 *If static device D_1 is behaviorally equivalent to static device D_2, then D_1 is alignably equivalent to D_2.*

Proof Let D'_1 and D'_2 be the static devices isomorphic to D_1 and D_2, respectively, such that for any assignment i_1 for D'_1 there exists assignment i_2 for D'_2 such that $\langle D'_1, i_1\rangle$ and $\langle D'_2, i_2\rangle$ are behaviorally indistinguishable and for any assignment i_2 for D'_2 there exists assignment i_1 for D'_1 such that $\langle D'_1, i_1\rangle$ and $\langle D'_2, i_2\rangle$ are behaviorally indistinguishable. By Definition 15, D'_2 unconditionally simulates D'_1. Similarly, D'_1 unconditionally simulates D'_2. D_1 and D_2 are therefore alignably equivalent.

$\square$

Proposition 2 *If static devices D_1 and D_2 are isomorphic, then they are behaviorally equivalent.*

Proof Let D_1 and D_2 be isomorphic static devices. D_1 is isomorphic to itself, so there exists a static device isomorphic to both D_1 and D_2. As any static device is behaviorally indistinguishable from itself, D_1 and D_2 are behaviorally equivalent.

□

2.1.2 System Language

Any model expressing a relationship between the signals of a system gives rise to a language describing that relationship. The alphabet for a system contains all possible combinations of values on its signals; a word over a system is a string or sequence over that alphabet. The set of all words consistent with the definition of the system is its language. Formally, the primitive object of system language is an *assignment*, a function from a set of names to the set of values over which those names are defined. Although this term is overloaded from Definition 1, its usage will be clear from context. As in the case for devices, the term M-assignment refers to an assignment over a set of names M.

Definition 16 Given a set M of names, an *M-word* is a sequence of M-assignments. If the sequence is finite, it is called an M-$*$-word, else it is called an M-ω-word. By default, an M-word is assumed to be an M-ω-word. If w is an M-word, w_i refers to the i^{th} assignment in the sequence w and w^i refers to the suffix of w beginning with the i^{th} assignment in the sequence.

Particular words are represented using a tabular format. Signal names appear on the left side of the table and indices appear across the bottom. Symbols 0, 1, and ? denote values in each position, where ? denotes a value of either 0 or 1. The symbol $\cdots$ indicates that the last depicted value holds thereafter.

Example 5 The table depicts the word in which a starts low and pulses twice before remaining low permanently and b starts high and falls twice, alternatively with a, before remaining high permanently.

a	0	1	0	1	0	$\cdots$
b	1	0	1	0	1	$\cdots$
	0	1	2	3	4	$\cdots$

Definition 17 An *M-language* is a set of M-words. An *M-$*$-language* contains only M-$*$-words. An *M-ω-language* contains only M-ω-words. By default, an M-language is assumed to be an M-ω-language.

Languages may also be represented by expressions, as is standard in automata theory. For example, the expression $a^n b^n$ represents the language of all finite words consisting of some number of a's followed by the same number of b's. In order to support infinite languages, the operator ω may be used in a manner similar to the Kleene-star operator $*$; $(a^n b^n)^\omega$ is the set of all infinite concatenations of strings consisting of some number of a's followed by the same number of b's.

Later definitions compare an M-language L with an M'-language L', where $M \subseteq M'$. In such cases, the statement $L \subseteq L'$ seems awkward. Languages over different sets of names are compared only for those names common to both; values on other names are ignored. Values to be ignored can be formally eliminated as follows, though this elimination is not done explicitly in the remainder of this work.

Definition 18 Let L be an M-language and let $M' \subseteq M$. The *M'-restriction* of L, denoted $L|M'$ is the language formed by restricting each assignment in L to the names included in M'. M-language L is an *extension* of M'-language L' iff $L|M' = L'$.

The Language of Devices

Unless otherwise noted, the language of a device is defined relative to its input and output ports; the internal ports are not included. This is reasonable because languages are generally used to model devices at the behavioral level. As a result, languages provide a clear means of defining behavioral equivalence classes of devices.

Definition 19 Let D be a device and let R be a run of D. Let P be a set of ports of D. The sequence of assignments in R restricted to P is called the *P-word derived from R*, denoted $\mathcal{W}(R)_P$. The *P-language of D*, denoted $\mathcal{L}(D)_P$, is the maximal P-language such that for each w in $\mathcal{L}(D)_P$, there exists a run R of D such that w is $\mathcal{W}(R)_P$. The P subscript is omitted if P is the set of interface ports of D.

Lemma 2 *Let D_1 and D_2 be devices over the same sets of interface ports. D_1 and D_2 are behaviorally equivalent iff $\mathcal{L}(D_1) = \mathcal{L}(D_2)$.*

Proof Assume D_1 and D_2 are behaviorally equivalent devices over the same sets of interface ports. Since an isomorphism simply renames ports, for any assignment i_1 for D_1 there exists assignment i_2 for D_2 such that $\langle D_1, i_1\rangle$ and $\langle D_2, i_2\rangle$ are behaviorally indistinguishable and for any assignment i_2 for D_2 there exists assignment i_1 for D_1 such that $\langle D_1, i_1\rangle$ and $\langle D_2, i_2\rangle$ are behaviorally indistinguishable; in each case, i_1 and i_2 agree on the values of all input ports.

Let $x \in \mathcal{L}(D_1)$. By Definition 19, x corresponds to a run of D_1 with assignment sequence the restriction of x^1 to the input interface ports of D_1. Let I denote the restriction of x^1 to the input interface ports of D_1 and let i_1 be the restriction of x_0 to the input interface ports of D_1.

Consider dynamic device $\langle D_1, i_1\rangle$. By definition, there exists i_2 such that $\langle D_2, i_2\rangle$ is behaviorally indistinguishable from $\langle D_1, i_1\rangle$ and i_1 and i_2 agree on all values for input interface ports. By definition of behaviorally indistinguishable, the run of $\langle D_1, i_1\rangle$ on a has the same output as the run of $\langle D_2, i_2\rangle$ on I. Therefore, x is the P-word derived from the run on $\langle D_2, i_2\rangle$, so $x \in \mathcal{L}(D_2)$.

An analogous argument shows that any word in $\mathcal{L}(D_2)$ is also a word in $\mathcal{L}(D_1)$.

Now assume $\mathcal{L}(D_1) = \mathcal{L}(D_2)$. Let i_1 be any assignment for D_1. Let I be an assignment sequence for D_1 and D_2. Let x be the word derived from the run of $\langle D_1, i_1\rangle$ on I. By assumption, x is in $\mathcal{L}(D_2)$, so there must exist assignment i_2 for D_2 such that x is the word derived from the run of $\langle D_2, i_2\rangle$ on I. Since the first letter of x agrees with i_1 by construction and since i_2 agrees with the first letter of x by definition, i_1 and i_2 must assign the same values to all common input ports. D_1

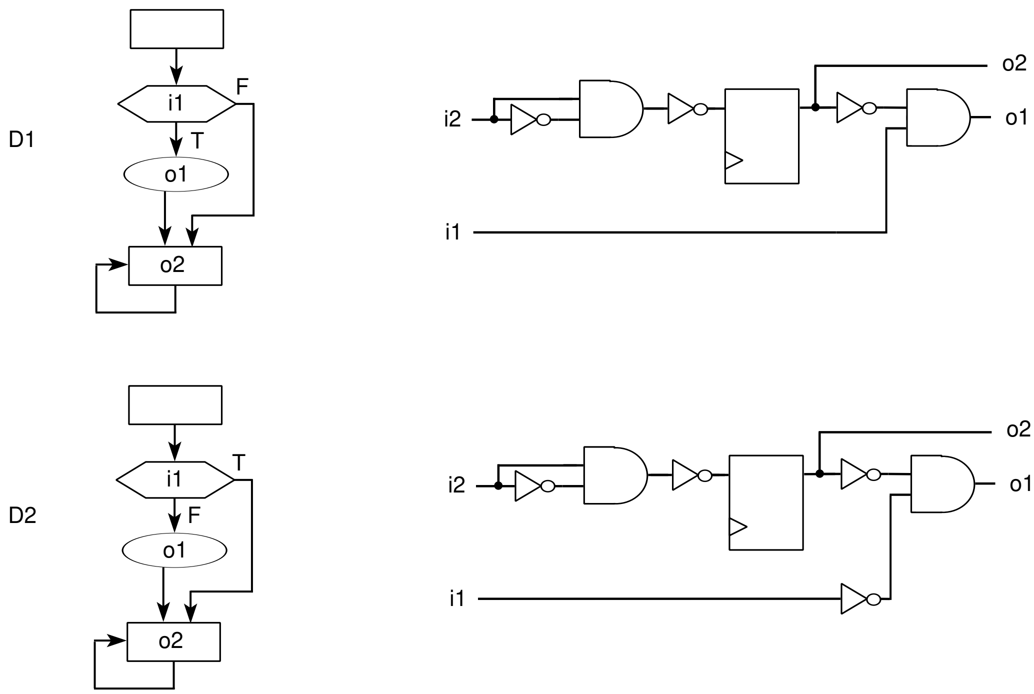


Figure 2.3: Diagrams depicting devices D_1 and D_2 that are initializably equivalent but that generate different languages (see Example 6).

and D_2 must therefore be behaviorally equivalent by Definition 15.

□

Lemma 2 does not necessarily hold if D_1 and D_2 are initializably equivalent without being behaviorally equivalent, as demonstrated in Example 6.

Example 6 Consider devices D_1 and D_2, depicted in Figure 2.3, with languages

$$\mathcal{L}(D_1) = \langle i_1, o_1 \rangle \langle o_2 \rangle^* + \langle \neg i_1, \neg o_1 \rangle \langle o_2 \rangle^* + \langle o_2 \rangle^*$$

$$\mathcal{L}(D_2) = \langle i_1, \neg o_1 \rangle \langle o_2 \rangle^* + \langle \neg i_1, o_1 \rangle \langle o_2 \rangle^* + \langle o_2 \rangle^*$$

and formally defined as follows:

- $D_1 = \langle \{i_1, i_2\}, \{o_1, o_2\}, B_1, \{w_1, \ldots, w_8\}, c_1 \rangle$, where

$$\begin{aligned} B_1 &= \{\mathsf{UnitDelay}(dp_1, dp_2), \mathsf{Inverter}(np_1, np_2), \mathsf{Inverter}(np_3, np_4), \\ &\quad \mathsf{Inverter}(np_5, np_6), \mathsf{AndGate}(ap_1, ap_2, ap_3)\}, \mathsf{AndGate}(ap_4, ap_5, ap_6)\} \\ c_1 &= \{(w_1, \{i_2, np_1, ap_1\}), (w_2, \{np_2, ap_2\}), (w_3, \{ap_3, np_3\}), (w_4, \{np_4, dp_1\}), \\ &\quad (w_5, \{dp_2, np_5, o_2\}), (w_6, \{np_6, ap_4\}), (w_7, \{i_1, ap_5\}), (w_8, \{ap_6, o_1\})\} \end{aligned}$$

- $D_2 = \langle \{i_1, i_2\}, \{o_1, o_2\}, B_2, \{w_1, \ldots, w_9\}, c_2 \rangle$, where

$$\begin{aligned} B_1 &= \{\mathsf{UnitDelay}(dp_1, dp_2), \mathsf{Inverter}(np_1, np_2), \mathsf{Inverter}(np_3, np_4), \mathsf{Inverter}(np_7, np_8) \\ &\quad \mathsf{Inverter}(np_5, np_6), \mathsf{AndGate}(ap_1, ap_2, ap_3)\}, \mathsf{AndGate}(ap_4, ap_5, ap_6)\} \\ c_1 &= \{(w_1, \{i_2, np_1, ap_1\}), (w_2, \{np_2, ap_2\}), (w_3, \{ap_3, np_3\}), (w_4, \{np_4, dp_1\}), \\ &\quad (w_5, \{dp_2, np_5, o_2\}), (w_6, \{np_6, ap_4\}), (w_7, \{i_1, np_7\}), (w_8, \{np_8.ap_5\}), \\ &\quad (w_9, \{ap_6, o_1\})\} \end{aligned}$$

The languages show that D_1 and D_2 are not behaviorally equivalent. They are, however, initializably equivalent. Given a dynamic device D_{Y_1} defined on D_1 with $dp_2 = 0$ and $i_1 = 1$, a dynamic device for D_2 can be defined using $i_1 = 0$ that is behaviorally indistinguishable from D_{Y_1}. Similar assignments can be made for all valid assignments on either D_1 or D_2. This points to the distinction between initializable equivalence and behavioral equivalence: in the former, the languages may differ in the inputs of the first assignment per word (but are the same if the first assignments are eliminated from each word). In the latter, identical languages are generated by both devices.

Subdevices can be replaced by behaviorally equivalent devices without affecting the behavior of the overall device. This property is useful when optimizing devices with respect to their numbers of gates. Stronger notions of when one subdevice can replace another have been studied in [78]; the behavioral equivalence notion is sufficient to prove certain inference rules on circuit diagrams in Chapter 4.

Lemma 3 *Let D be a device and let D_1 be a subdevice of D. Let D' be a device with subdevice D_2 such that $D - D_1$ is isomorphic to $D' - D_2$. If D_1 and D_2 are behaviorally equivalent, then D and D' are behaviorally equivalent.*

Proof Assume D_1 and D_2 are behaviorally equivalent. Then by Definition 15, there exist D_1' and D_2' isomorphic to D_1 and D_2, respectively, such that D_1' and D_2' are behaviorally equivalent and have the same sets of output ports, and the input interface ports of D_2' are a subset of the input interface ports of D_1'.

Let D'' be isomorphic to D' but defined over the ports of $(D - D_1) \cup D_2'$; D'' is definable since an isomorphism exists between $D - D_1$ and $D' - D_2$. Let i be any assignment for D and let I be any assignment sequence for D; I must also be an assignment sequence for D'' by the construction of D''. Let x be the word derived from the run of $\langle D, i\rangle$ over I. Let i'' be an assignment for D'' such that for all ports in $D'' - D_2'$, i'' agrees with i and for all ports in D_2', i'' is assigned such that $\langle D_1', i\rangle$ and $\langle D_2', i''\rangle$ are behaviorally indistinguishable. Since D_1' and D_2' are behaviorally equivalent, i'' can be defined such that the values on the input ports to D_2' agree with those on the input ports to D_1' in i.

The run of $\langle D'', i''\rangle$ must also derive word x. Since D and D'' are isomorphic minus D_1' and D_2', if $\langle D'', i''\rangle$ did not derive x, the difference would have to result from different outputs of D_1' and D_2'. Since these are behaviorally equivalent, they must have the same affect on ports that surround them. x must therefore be derived from the run of $\langle D'', i''\rangle$, so the output of the run of $\langle D, i\rangle$ and the output of the run of $\langle D'', i''\rangle$ must be equivalent. D and D'' are behaviorally equivalent from Definition 15, so D and D' must be behaviorally equivalent.

□

2.2 The Representations

HHL supports heterogeneous reasoning about a single design; it is therefore useful to have a common set of identifiers used for naming signals across all the representations. This set is called $\mathsf{SigNames}$ throughout the logic. In addition, some representations relate to moments of discrete time. Common expressions for denoting discrete time are therefore required. Intuitively, discrete time expressions consist of natural number variables and constants, and sums over natural numbers and constants, with their standard interpretations. Formally, they are interpreted by function TEval. Calls to TEval are of the form $\mathsf{TEVal}(e)[\phi]$, where e is the discrete time expression to be evaluated and ϕ is an assignment mapping temporal variables in e to natural number.

Definition 20 Let $t, t_1, t_2, \ldots$ be variables over the natural numbers called the *temporal variables*. Let $\overline{0}, \overline{1}, \ldots$ be temporal constants, one for each natural number; $\overline{t_c}, \overline{t_{c_1}}, \ldots$ will denote temporal constants with unspecified value. The *discrete time expressions* contain the temporal variables and temporal constants and are subject to the following construction rule: if e_1 and e_2 are discrete time expressions, then $e_1 + e_2$ is a discrete time expression.

Definition 21 Let ϕ be a function from temporal variables to natural numbers.

- $\mathsf{TEval}(\overline{k})[\phi] = k$ if k is a temporal constant.
- $\mathsf{TEval}(t)[\phi] = \phi(t)$ if t is a temporal variable.
- $\mathsf{TEval}(e_1 + e_2)[\phi] = \mathsf{TEval}(e_1)[\phi] + \mathsf{TEval}(e_2)[\phi]$ if e_1 and e_2 are discrete time expressions.

2.2.1 Circuit Diagrams

Circuit diagrams are the most standardized of the diagrammatic representations. Standard icons exist for gates and more complex components, such as multiplexers and ALUs. HHL defines circuit diagrams as a collection of components built up from three basic icons: and gate icons, inverter icons, and unit delay element icons. Circuit diagrams are extended to include more abstractly defined components in Section 2.2.7.

The term "circuit diagram" is reserved for those diagrams depicting well-connected devices. A drawing of gates and wires is called a "circuit sketch". A well-formedness definition identifies those circuit sketches that may be called circuit diagrams.

Syntax

HHL captures circuit diagrams at a high level; the position of a gate in a layout is irrelevant, as are the lengths of the wires. HLL only captures basic connectivity information. The primitive objects in a circuit diagram are therefore the points at which gates and wires are connected to one another; these are called *connection points*. Each gate and wire in the circuit diagram is depicted by a particular style of icon and is captured by a tuple of the connection points that it spans. Connection points on the right side of an icon are called *right elements*; those on the left side of an icon are correspondingly called *left elements*. Each circuit sketch is assumed to contain two special connection points called the *power point* and the *ground point*. These are denoted by the labels 0 and 1.

Definition 22 A *circuit sketch* is a tuple $\langle P, I, O, W, K, \mathsf{Labels}\rangle$ satisfying the following conditions:

1. P is a set of objects called the *connection points*.
2. W is a subset of $P \times P$ such that $(p_a, p_b) \in W$ iff they are the endpoints of a wire line. A wire line may consist of several horizontal and vertical segments attached endpoint to endpoint. Only the extremal endpoints are captured in W. A series of segments may apply to multiple wire lines iff a small black circle is placed at the branching point. (p_a, p_b) is not captured in W if they are both right endpoints or both left endpoints of segments in the line. Thus the rightmost wire line shown below generates only two wire lines. A wire (p_a, p_b) in a circuit

sketch is called *branch-free* if there exists no connection point $p_c \neq p_b$ such that (p_a, p_c) is a wire in W.

Wire Line

3. K is a (possibly empty) set of circuit sketches, each of whose I and O components are a subset of P.

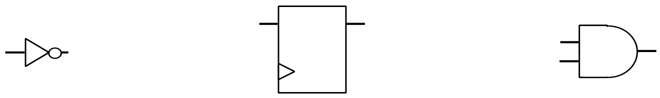
4. I and O are disjoint subsets of P such that O contains all connection points that do not appear as the left element of any element of K and I is a subset of those connection points that do not appear as the right element of any element of K. I and O are called the *input points* and *output points*, respectively.

5. Labels is a function from $I \cup O \cup W$ to $\mathsf{SigNames}$. Labels assigns a name to an interface port if the name appears next to the point. Labels assigns a name to a wire if it appears directly above or directly below the wire.

As this definition shows, circuit sketches are built up incrementally from other circuit sketches; this requires a set of basic circuit sketch building blocks. A circuit sketch $\langle P, I, O, W, K, \mathsf{Labels}\rangle$ is *basic* iff its set C of circuit sketches is empty.

Circuit sketches are generally drawn using standard icons for certain components, including the basic components defined in Definition 2. Definition 22 does not relate circuit sketches as mathematical objects to the icons that represent them diagrammatically. A mapping is required from the icons to the mathematical objects they denote. This mapping is called Icons; it is initialized with the icons corresponding to the predefined basic components.

Definition 23 Let the following three icons be named as shown



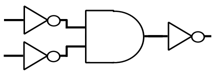
InverterIcon UnitDelayIcon AndIcon

The mapping Icons is initialized accordingly

1. $\mathsf{Icons}(\mathsf{InverterIcon}) = \langle \{p_1, p_2\}, \{p_1\}, \{p_2\}, \emptyset, \emptyset, \emptyset \rangle$
2. $\mathsf{Icons}(\mathsf{AndIcon}) = \langle \{p_1, p_2, p_3\}, \{p_1, p_2\}, \{p_3\}, \emptyset, \emptyset, \emptyset \rangle$
3. $\mathsf{Icons}(\mathsf{UnitDelayIcon}) = \langle \{p_1, p_2\}, \{p_1\}, \{p_2\}, \emptyset, \emptyset, \emptyset \rangle$

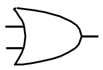
When specifying a non-basic circuit sketch $\langle P, I, O, W, K, \mathsf{Labels} \rangle$, icon names may appear in C. The icon name serves as a shorthand for its associated mathematical definition. It is treated as a function, that binds port names to the associated sketches interface ports. For example, one common non-basic component, an or gate, can be defined and assigned an icon as shown in Example 7.

Example 7 Consider the following non-basic circuit sketch and its mathematical definition



$$\begin{aligned}&\langle \{p_1, p_2, \ldots, p_{12}\}, \{p_1, p_2\}, \{p_{12}\},\\ &\ \{(p_1, p_3), (p_2, p_4), (p_5, p_7), (p_6, p_8), (p_9, p_{10}), (p_{11}, p_{12})\},\\ &\ \{\mathsf{InverterIcon}(p_3, p_5), \mathsf{InverterIcon}(p_4, p_6), \mathsf{InverterIcon}(p_{10}, p_{11}),\\ &\ \ \mathsf{AndIcon}(p_7, p_8, p_9)\}, \emptyset\}\rangle\end{aligned}$$

This sketch can be assigned an icon named OrIcon, shown on the left, in the Icons mapping as follows:



$$\mathsf{Icons}(\mathsf{OrIcon}) = \langle\{p_1, p_2, \ldots, p_{12}\}, \{p_1, p_2\}, \{p_{12}\},$$
$$\{(p_1, p_3), (p_2, p_4), (p_5, p_7), (p_6, p_8), (p_9, p_{10}), (p_{11}, p_{12})\},$$
$$\{\mathsf{InverterIcon}(p_3, p_5), \mathsf{InverterIcon}(p_4, p_6), \mathsf{InverterIcon}(p_{10}, p_{11}),$$
$$\mathsf{AndIcon}(p_7, p_8, p_9)\}, \emptyset\}\rangle$$

A circuit sketch $\langle P, I, O, W, K, \mathsf{Labels}\rangle$ is called *primitive* iff every element of K is a basic circuit sketch, as in Example 7. Any non-primitive circuit sketch can be transformed into a primitive circuit sketch by recursively collapsing K into the sets of gates and wires defined for the sketch via the following algorithm.

Algorithm 1

1. Let $C = \langle P, I, O, W, K, \mathsf{Labels}\rangle$ be a non-primitive circuit sketch. Denote the fields in circuit sketch k in K by $\langle k_P, k_I, k_O, k_W, k_K, k_{\mathsf{Labels}}\rangle$. Transform C into circuit sketch $\langle P', I', O', W',$ $K', \mathsf{Labels}'\rangle$ where
   (a) P' is $P \cup \bigcup_{k\in K} k_P$.
   (b) $I = I'$
   (c) $O = O'$
   (d) W' is $W \cup \bigcup_{k\in K} k_W$
   (e) Labels' is $\mathsf{Labels} \cup \bigcup_{k\in K} k_{\mathsf{Labels}}$
   (f) K' is the union of all basic circuit sketches in K with the elements of k_K for each non-basic circuit sketch $k \in K$.
2. Iterate Step 1 until the resulting circuit sketch is primitive.

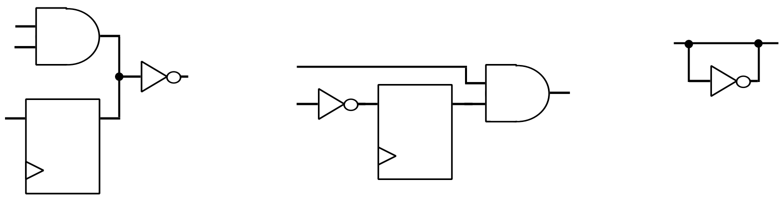
Circuit diagram semantics are defined relative to primitive circuit sketches. In addition, in giving semantics, it is easier to have the set of basic circuit sketches partitioned into sets of each type of basic circuit sketch.

Definition 24 Let $C = \langle P, I, O, W, K, \mathsf{Labels}\rangle$ be a primitive circuit sketch. The *primitive representation* of C is the tuple $\langle P, I, O, W, N, D, A, \mathsf{Labels}\rangle$ where N is the set of InverterIcons, D is the set of DelayIcons and A is the set of AndIcons of C.

Some readability conditions are placed on circuit sketches: icons should not overlap one another and wire lines should not overlap icons other than at connection points. In addition, sufficient space must exist so that labels can be unambiguously associated with wires and interface ports.

Several more formal restrictions must be placed on circuit sketches in order to related them to well-connected devices. For example, circuit sketches allow wires to connect points to themselves and allow multiple gate outputs to use the same connection point, as shown as Example 8.

Example 8 Only the second of these circuit sketches is well-formed. The first and third each connect two outputs to the same input.



Circuit sketches may be restricted to circuit diagrams either by imposing these restrictions on the syntax, as is usually done with a well-formedness definition, or by defining what it means for a circuit diagram to describe a physical device, then defining which sketches describe well-connected devices. The latter approach is used here; the former approach is taken on other HHL representations. First, some auxiliary definitions are required.

Definition 25 Given a circuit sketch $\langle P, I, O, W, N, D, A, \mathsf{Labels}\rangle$, there is a *simple connection* between connection points p_a and p_b in P, denoted $p_a \rightsquigarrow p_b$, if p_a is a left element and p_b the right element of some member of $W \cup N \cup D \cup A$.[3] A *connecting path* from connection point p_a in P to connection point p_b in P is a finite chain of simple connections

$$p_a = p_0 \leadsto p_1 \leadsto p_2 \leadsto \ldots \leadsto p_n = p_b$$

Definition 26 Given circuit sketches $C_1 = \langle P_1, I_1, O_1, W_1, N_1, D_1, A_1, \mathsf{Labels}_1 \rangle$ and $C_2 = \langle P_2, I_2, O_2, W_2, N_2, D_2, A_2, \mathsf{Labels}_2 \rangle$, a bijection θ from P_1 to P_2 is an *isomorphism between C_1 and C_2* iff

1. $\theta(I_1) = I_2$.
2. $\theta(O_1) = O_2$.
3. $\langle p_a, p_b \rangle \in W_1$ iff $\langle \theta(p_a), \theta(p_b) \rangle \in W_2$.
4. $\langle p_a, p_b \rangle \in N_1$ iff $\langle \theta(p_a), \theta(p_b) \rangle \in N_2$.
5. $\langle p_a, p_b \rangle \in D_1$ iff $\langle \theta(p_a), \theta(p_b) \rangle \in D_2$.
6. $\langle p_a, p_b, p_c \rangle \in A_1$ iff $\langle \theta(p_a), \theta(p_b), \theta(p_c) \rangle \in A_2$.

Circuit sketches C_1 and C_2 are *isomorphic* iff there exists an isomorphism between P_1 and P_2.

Definition 27 Circuit sketch $C_1 = \langle P_1, I_1, O_1, W_1, N_1, D_1, A_1, \mathsf{Labels}_1 \rangle$ is a *subsketch* of circuit sketch $C_2 = \langle P_2, I_2, O_2, W_2, N_2, D_2, A_2, \mathsf{Labels}_2 \rangle$, denoted $C_1 \subseteq C_2$, iff $P_1 \subseteq P_2$, $W_1 \subseteq W_2$, $N_1 \subseteq N_2$, $D_1 \subseteq D_2$, $A_1 \subseteq A_2$, and $\mathsf{Labels}_1 \subseteq \mathsf{Labels}_2$. Whenever $C_1 \subseteq C_2$, C_2 is an *extension* of C_1.

Semantics

Devices provide a semantic model for circuit sketches. Sketches can capture an entire device, or just a portion of one. The former is called *implementation* and the latter *description*. Treating

[3] $\leadsto$ is being used for both devices and circuit sketches as delineated by the context.

these separately allows diagrams to be built up into a full description of a real device. Relationships between circuit sketches and physical devices are based on mappings between the connection points of the sketches and the ports of the devices.

Definition 28 Let $C = \langle P, I_C, O_C, W_C, N, D_C, A, \mathsf{Labels}\rangle$ be a primitive circuit sketch and $D = \langle I_D, O_D, B, W_D, c\rangle$ be a static device.

1. A *depiction map* from C to D is an injective function σ from the connection points P of C into the ports of D such that the power point maps to the power port, the ground point maps to the ground port, and for all p in P,
   (a) If $\langle p_a, p_b\rangle$ is a wire line of C then $\sigma(p_a)$ and $\sigma(p_b)$ are wired together by some wire $w \in W_D$; $\sigma(p_a)$ must be an input interface port or internal output port and $\sigma(p_b)$ must be an output interface port or internal input port.
   (b) If $\langle p_a, p_b\rangle$ is a negation icon of C then B contains $\mathsf{Inverter}(\sigma(p_a), \sigma(p_b))$.
   (c) If $\langle p_a, p_b\rangle$ is a unit delay icon of C then B contains $\mathsf{UnitDelay}(\sigma(p_a), \sigma(p_b))$.
   (d) If $\langle p_a, p_b, p_c\rangle$ is an and-gate icon of C then B contains $\mathsf{AndGate}(\sigma(p_a), \sigma(p_b), \sigma(p_c))$.
2. C *describes* D, written $D \models C$, if there is a depiction map from C to D.
3. Non-primitive circuit sketch C' describes D, written $D \models C'$ if there is a depiction map from the primitive representation of C' to D.
4. D is a *structural implementation of* C, denoted $D \models_s C$, if there is a surjective depiction map from C to D, $p \in I_C$ implies $\sigma(p) \in I_D$, $p \in O_C$ implies $\sigma(p) \in O_D$, and the converse of requirements 1a through 1d hold. D is a structural implementation of a non-primitive circuit sketch C' iff it is a structural implementation of the primitive representation of C'.
5. D is a *behavioral implementation of* C, denoted $D \models_b C$, if D is behaviorally equivalent to some device D' which is a structural implementation of C. D is a behavioral implementation of a non-primitive circuit sketch C' iff it is a behavioral implementation of the primitive representation of C'.

Proposition 3 *If $D \models_s C$, then $D \models C$ and $D \models_b C$.*

Proof If $D \models_s C$, then $D \models C$ by definition. $D \models_b C$ since every device is behaviorally equivalent to itself.

□

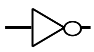
There is no relationship between $D \models C$ and $D \models_b C$; the two notions are incomparable as shown in Example 9.

Example 9

1. Let D be a device consisting of an AndGate whose output is wired to the input of an Inverter. Formally, D is modeled as follows:

   - $I = \{ip_1, ip_2\}$
   - $O = \{op\}$
   - $B = \{\mathsf{AndGate}(ap_1, ap_2, ap_3), \mathsf{Inverter}(np_1, np_2)\}$
   - $W = \{w_1, w_2, w_3, w_4\}$
   - $c = \{(w_1, \{ip_1, ap_1\}), (w_2, \{ip_2, ap_2\}), (w_3, \{ap_3, np_1\}), (w_4, \{np_2, op\})\}$

   Let C be the following circuit sketch:

   C is modeled as $\langle \{sp_1, sp_2, p_{in}, p_{out}\}, \{p_{in}\}, \{p_{out}\}, \{\{p_{in}, sp_1\}, \{sp_2, p_{out}\}\}, \{sp_1, sp_2\}, \emptyset, \emptyset \rangle$

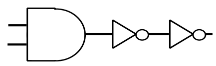
   $D \models C$ under depiction map $\sigma = ((p_{in}, ap_3), (p_{out}, op), (sp_1, np_1), (sp_2, np_2))$. However, $D \not\models_b C$ since the number of input interface ports in D differs from the number of input points of C.

2. Let D be a device consisting only of an AndGate. Formally, D is modeled as follows:

   - $I = \{ip_1, ip_2, ip_3\}$

- $O = \{op\}$
- $B = \{\mathsf{AndGate}(ap_1, ap_2, ap_3)\}$
- $W = \{w_1, w_2, w_3\}$
- $c = \{(w_1, \{ip_1, ap_1\}), (w_2, \{ip_2, ap_2\}), (w_3, \{ap_3, op\})\}$

Let C be the following circuit sketch:



C is modeled as follows:

- $P = \{sp_1, sp_2, sp_3, sap_1, sap_2, sap_3, snp_1, snp_2, snp_3, snp_4\}$
- $I = \{sp_1, sp_2\}$
- $O = \{sp_3\}$
- $W = \{(sp_1, sap_1), (sp_2, sap2), (sap_3, snp_1), (snp_2, snp_3), (snp_4, sp_3)\}$
- $N = \{\{snp_1, snp_2\}, \{snp_3, snp_4\}\}$
- $D = \emptyset$
- $A = \{sap_1, sap_2, sap_3\}$

From the definition, it can be shown that $D \models_b C$. However, $D \not\models C$ since C contains an inverter icon while D contains no Inverter.

Proposition 4 *If $D \models C$ and C' is a subsketch of C, then $D \models C'$.*

Proof If $D \models C$, then by definition there exists a depiction map σ from C to D. Since C' is a subsketch of C, the connection points of C' are a subset of those for C. It follows that the depiction map formed by restricting σ to the connection points in C' is a depiction map from C' to D.

□

Notice that Proposition 4 does not hold with $\models$ replaced by $\models_s$ or $\models_b$.

The next two theorems classify which circuit sketches describe well-connected devices and which are implemented by well-connected devices. The conditions in these theorems are used to give a well-formedness condition on circuit sketches immediately following the proofs of the theorems.

Theorem 1 *Primitive circuit sketch $C = \langle P, I, O, W, N, D, A, \mathsf{Labels}\rangle$ describes some well-connected device if and only if it has the following properties:*

1. *The elements of $N \cup D \cup A$ are all pairwise disjoint with respect to the sets of connection points they contain; these connection points are also disjoint from I and O.*
2. *There is no connection point p in C that is the right element of more than one tuple in W.*
3. *There is no connection point p_a in C such that $\langle p_a, p_b\rangle$ and $\langle p_c, p_a\rangle$ are both wire lines in C.*
4. *There is no $\langle p_a, p_b\rangle$ in W such that p_a and p_b are both right elements for tuples in $N \cup D \cup A$.*
5. *Any cyclic path from connection point p_a to connection point p_b contains an element $\langle p_c, p_d\rangle$ of D.*

Proof Assume C describes some well-connected device $D' = \langle I', O', B', W', c'\rangle$. Let σ be the depiction map from the connection points of C to the ports of D'. The following argument establishes the necessity of the five conditions:

1. If there exists a connection point p_a that appears in more than one element of $N \cup D \cup A$, then port $\sigma(p_a)$ must be connected to more than one basic component in B. This is prohibited by the definition of a static device which requires the sets of ports for all basic components to be disjoint. Similarly, p_a can not appear in both an element of $N \cup D \cup A$ and either of I or O without violating the requirement that the interface ports in a static device be disjoint from those included in the basic components.

2. Assume distinct tuples $\langle p_a, p_b\rangle$ and $\langle p_c, p_b\rangle$ are both elements of W. Then $\sigma(p_a)$ and $\sigma(p_b)$ are wired together in D, as are $\sigma(p_b)$ and $\sigma(p_c)$. A wire w therefore exists in D such that $\{\sigma(p_a), \sigma(p_b), \sigma(p_c)\} \subseteq c(w)$. By the definition of σ, each of p_a and p_c map to input interface ports or internal output ports. This contradicts the restriction on the partition generated by $c(w)$, which may include only one input interface port or internal output port.

3. Assume $\langle p_a, p_b\rangle$ and $\langle p_c, p_a\rangle$ are both wires lines in C. Then by the definition of σ, $\sigma(p_a)$ must be both an input interface port or internal output port and an output interface port or internal input port. All of these cases violate the disjointness requirements on the ports of D'.

4. Assume $\langle p_a, p_b\rangle$ is an element of W and both p_a and p_b are right elements for tuples in $N \cup D \cup A$. Since $\langle p_a, p_b\rangle$ is in W, $\sigma(p_a)$ must be an input interface port or internal output port and $\sigma(p_b)$ must be an output interface port or internal input port. The definition of σ requires that all right elements of tuples in $N \cup D \cup A$ map to output ports of basic components in D'. This raises a contradiction for the port determined by $\sigma(p_b)$.

5. Assume C contains a cyclic path from connection point p_a to connection point p_b that does not contain any delay icon $\langle p_c, p_d\rangle$. By the definition of connecting paths (Defn. 25) and σ, there must be a connecting cycle in D' on the path from $\sigma(p_a)$ to $\sigma(p_b)$. If there is no delay icon on the cyclic path from p_a to p_b, then there is no delay element in the connecting cycle from $\sigma(p_a)$ to $\sigma(p_b)$ because every $\langle p_1, p_2\rangle$ in the cyclic path must map to an object in D' corresponding to its icon; by the definition of describes (Defn. 28), no other sequential element can lie along the path from $\sigma(p_a)$ to $\sigma(p_b)$. D' would therefore contain a connecting cycle with no sequential element, which violates the assumption that D' is well-connected.

For the converse, assume C satisfies the five conditions. A well-connected static device D' such that C describes D' is constructed using the following algorithm:

1. Initialize a function σ to be empty.

2. For each connection point p_a in C, add a new port p to D' and add the pair $\langle p_a, p\rangle$ to σ.

3. For each connection point p_a in C_I, add $\sigma(p_a)$ to the set of input interface ports in D'; for each connection point p_a in C_O, add $\sigma(p_a)$ to the output interface ports of D'.

4. For each tuple $\langle p_a, p_b \rangle$ in the negation icons of C, add $\mathsf{Inverter}(\sigma(p_a), \sigma(p_b))$ to the set of basic components of D'.

5. For each tuple $\langle p_a, p_b, p_c \rangle$ in the and-gate icons of C, add $\mathsf{AndGate}(\sigma(p_a), \sigma(p_b), \sigma(p_c))$ to the set of basic components of D'.

6. For each tuple $\langle p_a, p_b \rangle$ in the unit delay icons of C, add $\mathsf{UnitDelay}(\sigma(p_a), \sigma(p_b))$ to the set of basic components of D'.

7. Define a function g from connection points to subsets of connection points such that $g(p_a)$ is the set of all connection points wired to p_a in C. For each element in the domain of g that maps to a non-empty subset, create a wire in D'. Define the function c in D' on these wires as follows: let w be the wire in D' associated with element p_a in the domain of g. Then $c(w) = \sigma(\{p_a\} \cup g(p_a))$.

8. If D' is not connected, let D_1 and D_2 be two disjoint components of D'. Let p_1 be any port in D_1 and p_2 be any port in D_2. Let p_n, p_m, p_l be three ports not already contained in any component of D'. Add $\langle \{\sigma(p_n), \sigma(p_m)\}, \{\sigma(p_l)\}, F, 0 \rangle$ to the basic components of D', where F returns the maximum of its arguments. If $p_1 \in c(w)$ for some wire $w \in D'$, add p_n to $c(w)$; otherwise create a new wire w in D' and set $c(w) = \{p_1, p_n\}$. Similarly connect p_2 and p_m. Repeat this step until D' is connected.

9. For each internal port p of D' for which there is no connecting path from an input interface port of D' to p, create a new input interface port p_n in D'. If $p \in c(w)$ for some wire w, add p_n to $c(w)$; otherwise create a new wire w in D' and set $c(w) = \{p, p_n\}$.

10. For each internal port p of D' for which there is no connecting path from p to an output interface port of D', create a new output interface port p_n in D'. If $p \in c(w)$ for some wire w, add p_n to $c(w)$; otherwise create a new wire w in D' and set $c(w) = \{p, p_n\}$.

The depiction map σ constructed by this algorithm is injective. Device D' constructed by this algorithm must be shown to meet the requirements of a well-connected static device. Recall that the algorithm created a unique port in D' for each connection point in C.

1. The connection points in the sets I and O of C are disjoint by definition, so the sets of input interface ports and output interface ports in D' must be disjoint by construction.

2. The algorithm created a basic component in D' for each element of $N \cup D \cup A$ of C. The connection points in these sets are disjoint by assumption, so basic components introduced in Steps 4 through 6 are guaranteed to have disjoint ports. The only other step in the algorithm that adds basic components to D' is Step 8. This step uses new ports for each basic component it creates, so these ports must also be disjoint from the other ports appearing in basic components in D'. The sets of ports for all basic components in D' are therefore pairwise disjoint; they are disjoint from the interface ports of D' by construction.

3. By definition, the wiring function c in D' has the set of wires as its domain and subsets of ports as its co-domain. Assume c does not partition the set of ports; then either some port appears in more than one cell of c or some port does not appear in any cell of c. Each case is considered in turn.

   Assume there exists a port p such that $p \in c(w_1)$ and $p \in c(w_2)$ for distinct wires w_1 and w_2 in D'. Consider the algorithm up to Step 7. One wire was generated for each left element of a wire line in C; connection points p_a and p_b must therefore exist such that $p \in \sigma(\{p_a\} \cup g(p_a))$ and $p \in \sigma(\{p_b\} \cup g(p_b))$. If $p = \sigma(p_a)$, then $p \neq \sigma(p_b)$ since σ is injective by construction. Therefore, $p \in \sigma(g(p_b))$, so $\langle p_b, p_a \rangle$ must be a wire line in C. Since a wire was created corresponding to p_a, there must exist a p_c such that $\langle p_a, p_c \rangle$ is also a wire line in C. This contradicts the assumption that no connection point appears as a left element in one wire line and a right element in another. A similar argument proves that p can not be $\sigma(p_b)$. This leaves the case that p corresponds to some other connection point p_c and $\langle p_a, p_c \rangle$ and $\langle p_b, p_c \rangle$ are both wire lines in C, which violates the assumption that no connection point serves as the right element for more than one wire line. All ports introduced in the algorithm up through Step 7 are therefore partitioned by c

The algorithm cannot place a port p introduced after Step 7 into more than one cell of c. Whenever a new port is created after Step 7, it is placed in the cell of the port to which it is wired, if such a cell exists. Otherwise a new cell is created, indicating that the pre-existing port was not already included in the partition generated by c. The partitions are thus maintained by construction; this completes the proof that no port p can appear in more than one cell of the partition generated by c.

The other possibility is that there exists a port p in D' that is not in $c(w)$ for any wire w in D'. In this case, there can be no wire w' attached to port p; if there were, p would have been placed in $c(w')$ by the algorithm. Since the ports in the basic components and the interface ports in D' are all disjoint, there can be no path to p from the input interface ports if p is an internal input port or no path from p to an output interface port if p is an internal output port. That this case is impossible is established below.

4. Step 8 establishes that D' is connected by construction.

5. Steps 9 and 10 establish by construction that every internal port p in D' has a connecting path from an element of the input interface ports to p and a connecting path from p to some element of the output interface ports of D'.

6. The only steps in the algorithm that could have generated a connecting cycle in D' are Steps 4 through 7. Since the depiction map is injective, a cycle could only have been created in D' if a cycle existed in C. The assumptions state that any cyclic path in C contains some delay icon $\langle p_a, p_b \rangle$; this icon would have mapped to a delay element in D' by construction.

$\square$

Theorem 2 *Primitive circuit sketch* $C = \langle P, I, O, W, N, D, A, \mathsf{Labels} \rangle$ *is implemented by some well-connected device if and only if it has the properties of Theorem 1 plus the following properties:*

1. *Treating the connection points as nodes and the elements of $W \cup N \cup D \cup A$ as denoting edges renders a connected graph; in other words, the entire sketch must represent a single connected component.*

2. *For all connection points p, there must be a directed path to p from a connection point in I.*

3. *For all connection points p, there must be a directed path from p to a connection point in O.*

4. *For all connection points p not in I, p is the right element for some tuple in $W \cup N \cup D \cup A$.*

5. *For all connection points p not in O, p is the left, top left, or bottom left element for some tuple in $W \cup N \cup D \cup A$.*

Proof

1. This step proves that the properties of the theorems are satisfied. Assume C is implemented by some well-connected device D'. Then by definition C describes D', so the conditions of the first part of the Theorem 1 are all satisfied. Let σ be the bijective depiction map from the connection points of C to D. Each of the above properties is considered in turn.

   (a) Assume that C is not connected. Let C_1 be a connected component of C and let P_1 be the set of connecting points in C_1. Define F_1 to be the set of all ports of D' with connecting steps to or from ports in $\sigma(P_1)$, but not contained in $\sigma(P_1)$. Let p be any port in F_1 and let C_2 be the connected component of C that contains $\sigma^{-1}(p)$. Let e be the connecting element of p and port $p_1 \in \sigma(p_1)$. Neither $\langle \sigma^{-1}(p), \sigma^{-1}(p_1) \rangle$ nor $\langle \sigma^{-1}(p_1), \sigma^{-1}(p) \rangle$ can be an element of $W \cup N \cup D \cup A$ in C, for otherwise C_1 and C_2 would be connected. This contradicts the assumption that the converse of each condition in the definition of a depiction map holds; since there is some object in D' that connects p and p_1, there must be a tuple in $W \cup N \cup D \cup A$ containing both $\sigma^{-1}(p)$ and $\sigma^{-1}(p_1)$, and C must indeed be connected.

   (b) Assume there exists a connection point p such that there does not exist a connection point p_i in I with a path from p_i to p. Since D' is well-connected, there must exist a connecting path from some input interface port of D' to $\sigma(p)$. Let $p_0 \rightsquigarrow p_1 \rightsquigarrow \ldots p_n = \sigma(p)$ represent this path. Consider each connecting step $p_i \rightsquigarrow p_{i+1}$. By the converse of the definitions for a depiction map (Defn. 28), the connecting element for each connecting step must correspond to some icon in C. A path must therefore exist from some connection point p_i to p; namely, $\sigma^{-1}(p_0)$ is such a p_i. The proof that there is directed path from each connection point p to a connection point in O is analogous.

(c) Assume there is a connection point p that is not the right element for some tuple in $W \cup N \cup D \cup A$. Since p is not the right element of a tuple in $N \cup D \cup A$, by the converse of the definitions in a depiction map, $\sigma(p)$ is not an internal output port. Since p is not the right element of a wire in C, then $\sigma(p)$ cannot be an internal input port in D'; $\sigma(p)$ must be an interface port in D'. If $\sigma(p)$ is an output interface port of D', there must be some wire in D' that connects $\sigma(p)$ to the rest of the device since D' is assumed to be connected. This wire maps to a wire line in C with p as a right element, which is a contradiction. That leaves only the case that $\sigma(p)$ is an input interface port of D', but this also contradicts the assumption that p is not in I. Therefore, p must be the right element of some element of $W \cup N \cup D \cup A$. A similar argument shows that all connection points not in O must be a left element of some tuple in $W \cup N \cup D \cup A$.

2. This step proves that the properties are sufficient to guarantee that C is implemented by some well-connected device. Assume C has all properties stated in Theorems 1 and 2. Use the algorithm provided in Theorem 1 to construct a well-connected static device D' and the corresponding depiction map σ. D' is well-connected from the proof of Theorem 1; it must be shown that σ is surjective and that the converse of each statement in the definition of σ holds. Each of these is considered in turn.

(a) Assume that σ is not surjective. Then there exists some port p in D' that is not mapped to by a connection point in C. The only steps of the algorithm that add ports to D' without having a corresponding connection point are 8, 9, and 10.

Step 8 adds ports as necessary to make D' connected. However, since C is connected by assumption, D' must also be connected by construction. Step 8 is therefore never taken.

Step 9 adds ports as necessary to connect internal ports to input interface ports. However, since there is a path to every connection point in C from some connection point in I by assumption, by construction there is a connecting path to every internal port in D' from some input interface port of D'. Therefore, Step 9 is never taken; a similar argument shows that Step 10 is also not taken. Hence, the only ports in D' are those that were added specifically to correspond to a connection point in C, so σ is surjective.

(b) If the converse of each statement in the definition of a depiction map does not hold, then one of the following cases must apply:

i. $\sigma(p) \in D'_I$ but $p \notin s_I$. Then by the assumptions, p must be the right element of some tuple in $W \cup N \cup D \cup A$. By definition of σ, $\sigma(p)$ is either an internal input port or an output interface port; either case contradicts $\sigma(p) \in D'_I$ by the conditions on disjointness of sets of ports. A similar argument refutes the case where $\sigma(p) \in D'_O$ but $p \notin s_O$.

ii. $\sigma(p_a)$ and $\sigma(p_b)$ are wired together in D', $\sigma(p_a)$ is an input interface port or internal output port, $\sigma(p_b)$ is an output interface port or internal input port, but $\langle p_a, p_b \rangle$ is not a wire line in C. By definition, two connection points are wired together if they are elements of $c(w)$ for some wire w. Since $\sigma(p_a)$ is the only input interface or internal output port in $c(w)$ (by Definition 1, $c(w)$ can contain only one such port), $\sigma(p_b)$ was added to $c(w)$ because $\langle p_a, p_b \rangle$ is a wire line in C.

iii. $\sigma(p_a)$ and $\sigma(p_b)$ are connected to the input and output ports of an inverter, respectively, but $\langle p_a, p_b \rangle$ is not a negation icon in C. This is a contradiction based on construction; the algorithm only adds an inverter between $\sigma(p_a)$ and $\sigma(p_b)$ if $\langle p_a, p_b \rangle$ is an $\mathsf{Inverter}$ in C. A similar argument holds for $\mathsf{UnitDelay}$ elements. The algorithm does add $\mathsf{AndGates}$ in Step 8, but it was established above that this step is not executed under the current assumptions; hence, a similar argument holds for $\mathsf{AndGates}$.

Since all cases result in contradiction, the converse of each statement in the definition of depiction map σ is upheld under the given set of assumptions. D' therefore is a structural implementation of C.

□

Definition 29 A *well-formed* primitive circuit sketch is a sketch satisfying the conditions of Theorem 1. A *complete* well-formed primitive circuit sketch is a sketch that satisfies the conditions of both Theorem 1 and Theorem 2. A non-primitive circuit sketch is well-formed iff its primitive representation is well-formed. Henceforth, the term *circuit diagram* refers to a complete well-formed circuit sketch.

Proposition 5 *For any device D there exists a circuit diagram C that is unique up to isomorphism on circuit diagrams such that $D \models_s C$.*

Proof This is a direct result of the correlation between AndGates, Inverters, UnitDelay elements, and their respective circuit diagram icons.

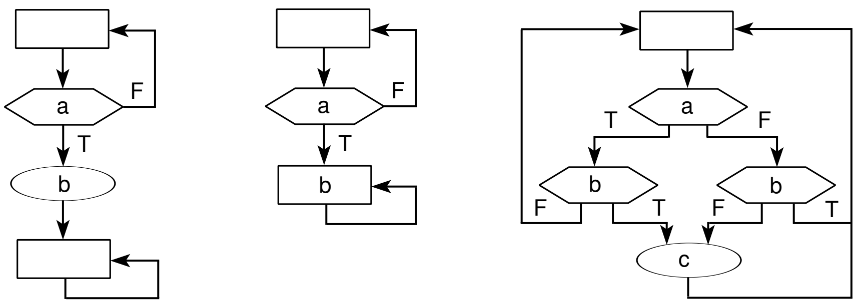
□

2.2.2 ASM Charts

ASM charts are a variant of state machine notation that combines the traditional Mealy and Moore machines [65]. They have an appearance reminiscent of flow-charts: rectangles represent states, diamonds represent conditional branches, and ovals represent conditional (Mealy) outputs. Placing a variable within a state rectangle designates a Moore output. Example 10 provides simple ASM charts; more extensive examples and details can be found in [70]. As in the section on circuit diagrams, the term *ASM chart* is reserved for well-formed diagrams; *ASM sketch* is used for the general case.

Example 10 ASM chart 1 waits for the value of signal a to be true and then asserts b. Control then enters the looping state at the bottom of the chart and remains there; signal b is not asserted in the looping state. ASM chart 2 is similar to chart 1 except b is now asserted continuously once a

is asserted. ASM chart 3 depicts the algorithm that continually assigns c the boolean value of $a = b$.



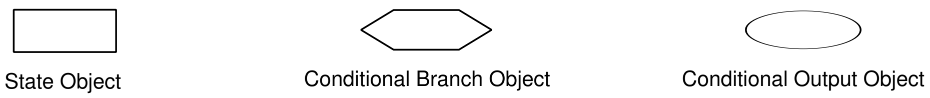
(1) (2) (3)

When using ASM charts, it is common to assign names to the individual states. A particular set of identifiers, $\mathsf{StateNames}$, is reserved for this purpose. $\mathsf{StateNames}$ and $\mathsf{SigNames}$ are disjoint; context indicates to which set a particular identifier belongs.

Syntax

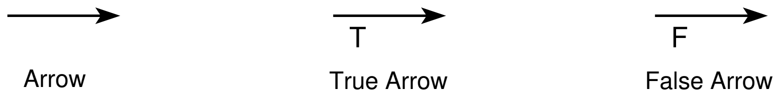
Definition 30 An *ASM sketch* is a tuple $\langle T, B, O, A, A_T, A_F, \mathsf{Signals}, \mathsf{Labels}, \mathsf{CurrStates}\rangle$ satisfying the following conditions:

1. T is a set of square-shaped objects called the *state-objects*, B is a set of diamond-shaped objects called the *conditional branch objects*, and O is a set of oval-shaped objects called the *conditional output objects*.

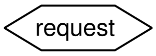


2. A is a subset of $\{T \cup O\} \times \{T \cup B \cup O\}$ called the *arrows*. $(s, t) \in A$ iff there exists an arrow whose tail touches s and whose head touches t. A_T and A_F are subsets of $B \times \{T \cup B \cup O\}$

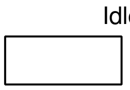
called the *true arrows* and *false arrows*, respectively. $(s,t) \in A_T$ iff there exists a true arrow whose tail touches s and whose head touches t. $(s,t) \in A_F$ iff there exists a false arrow whose tail touches s and whose head touches t. The components of these arrow types are called the *source object* and the *target object*, respectively. The set $A \cup A_T \cup A_F$ are called the *arrow objects*.



3. Signals is a subset of $T \cup B \cup O \times$ SigNames such that $(obj, n) \in$ Signals iff n appears inside object *obj*.



4. Labels is a subset of StateNames $\times\ T$ such that $(l,t) \in$ Labels if l appears at the upper right hand corner of t.



5. CurrStates is the (possibly empty) set of state objects such that each $t \in$ CurrStates has a current state object pointing to its left edge. There are two types of current state objects: those annotated with circles containing discrete time expressions and those with no annotations. A current state object of the latter type are called *time-free*. An ASM sketch is *time-annotated* if it contains a non-time-free current state object.

States Annotated By Current State Objects

Unlike circuit sketches, ASM sketches do not have a notational means of distinguishing input signals from output signals. However, the roles of signals can be inferred from their usage in the chart: those signals that are tested conditionally must be generated externally to the physical device corresponding to an ASM sketch, while those signals generated (or output) by the ASM sketch must be internal to that device. This observation prompts the following definition:

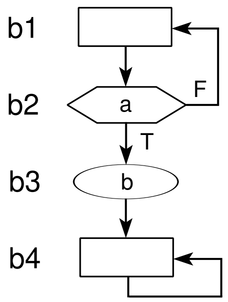
Definition 31 The *external signals* of an ASM sketch M are those elements $n \in N$ for which $(obj, n) \in \mathsf{Signals}$ and *obj* is a conditional branch object. The *internal signals* of M are those elements $n \in N$ for which $(obj, n) \in \mathsf{Signals}$ and *obj* is a state object or a conditional output object.

ASM sketches are used mainly to specify control-flow in a device. As a result, the paths through the sketch are of considerable interest. Certain types of paths are useful, namely those from one state to another, and those from states to conditional output objects.

Definition 32 A *flow-path* in an ASM sketch $M = \langle T, B, O, A, A_T, A_F, \mathsf{Signals}, \mathsf{Labels}, \mathsf{CurrStates}\rangle$ is a finite sequence of objects $j_0, \ldots, j_k$ such that for all $0 \leq i < k$, $(j_i, j_{i+1}) \in A \cup A_T \cup A_F$.

1. A flow-path is *state-contained* if j_0 is a state object and for no $0 < i < k$ is j_i a state-object.
2. A state-contained flow-path $j_0, \ldots j_k$ such that j_k is a state object is called a *next-state transition path.*
3. A state-contained flow-path $j_0, \ldots j_k$ such that j_k is a conditional output object is called an *output condition path.*
4. A flow-path $j_0, \ldots, j_k$ such that $j_0 = j_k$ and j_0 is a state object is called a *transition cycle.*
5. Objects j_1 and j_2 are *in the same state* if there exists a state-contained flow path containing both j_1 and j_2 where neither j_1 or j_2 is the final object in the path.

Example 11 Suppose the objects in the ASM sketch are referred to by the labels on the left.



Flow-path b_1 b_2 b_3 is a state-contained output condition path. Flow-path b_1 b_2 b_1 is a state-contained next-state transition path; it is also a transition cycle. Flow-path b_2 b_3 is not state-contained, not a next-state transition path, and not an output condition path. Objects b_2 and b_2 are, however, in the same state.

There are a number of constraints on well-formed ASM sketches: paths through them should be infinite in length, cycles should pass through states, there should be only one current state at a time, and inputs and outputs should be distinct. The well-formedness constraints also eliminate sketches with redundancies: the same signal need not be tested or output more than once along the same path. Example 12 shows some non-well-formed ASM charts.

Definition 33 An ASM sketch $M = \langle T, B, O, A, A_T, A_F, \mathsf{Signals}, \mathsf{Labels}, \mathsf{CurrStates}\rangle$ is *well-formed* iff all of the following constraints are satisfied:

1. M contains at least one state object.
2. The graph formed by treating the objects as nodes and the arrows, true arrows and false arrows as edges consists of a single, connected component.
3. For each state object t, there is exactly one element of A with source object t.
4. For each conditional branch object b, there is exactly one element of A_T with source object b and exactly one element of A_F with source object b.
5. For each conditional output object o, there is exactly one element of A with source object o.
6. In every flow-path $j_0, \ldots, j_k$ for which $j_0 = j_k$, there exists $0 \leq i < k$ such that j_i is a state object.
7. There does not exist a state-contained flow-path $j_0, \ldots, j_k$ with $0 \leq i \leq k$ and $0 \leq m \leq k$, $i \neq m$, and a signal name n such that

(a) j_i and j_m are both conditional output objects or both conditional branch objects, and

(b) (j_i, n) and (j_m, n) are both in Signals.

8. There does not exist a state-contained flow-path $j_0, \ldots, j_k$ with $0 \leq i < k$ and signal name n such that j_i is a conditional output object, $(j_0, n) \in$ Signals, and $(j_i, n) \in$ Signals.

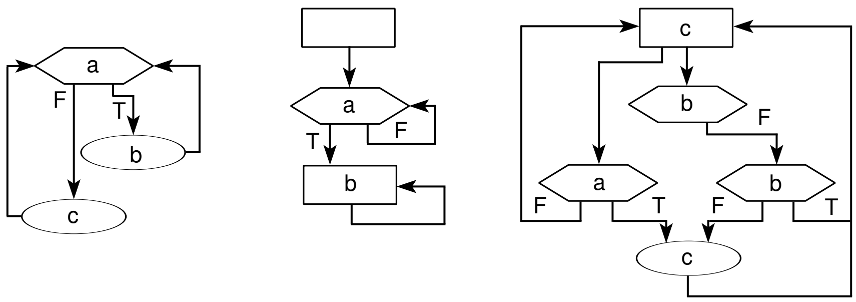
9. There does not exist a conditional branch object or a conditional output object that is not the target object for some element of $A \cup A_T \cup A_F$.

10. The external and internal signals of M are disjoint.

11. CurrStates contains at most one element.

Well-formed ASM sketches are called *ASM charts*.

Example 12 These ASM sketches are not well-formed. Chart 1 does not have a state. Chart 2 contains a cycle with no state. Chart 3 has four problems: a state with two edges leaving it, a conditional test with only one edge leaving it, a signal that is asserted twice along the same path, and a signal that is tested twice along the same path.



(1) (2) (3)

Semantics

ASM charts are given semantics relative to physical devices. ASM charts are more abstract than devices. Their modeling relationships are therefore more complicated than those for circuit diagrams.

Two sets of information are abstracted from an ASM chart in order to simplify the presentation: the conditions on signals required for each state transition, and the conditions on signals required for each output assertion.

Definition 34 Let $M = \langle T, B, O, A, A_T, A_F, \mathsf{Signals}, \mathsf{Labels}, \mathsf{CurrStates}\rangle$ be an ASM chart and let p be a next-state transition path $j_0, \ldots, j_k$ in M. The *next-state transition* corresponding to p is the tuple $\langle j_0, j_k, c_t, c_f\rangle$ where

1. c_t contains all signal names n for which there exists $0 < i < k$ such that j_i is a conditional branch object, $(j_i, n) \in \mathsf{Signals}$, and $(j_i, j_{i+1}) \in A_T$.

2. c_f contains all signal names n for which there exists $0 < i < k$ such that j_i is a conditional branch object, $(j_i, n) \in \mathsf{Signals}$, and $(j_i, j_{i+1}) \in A_F$.

The set of all next-state transitions corresponding to some next-state transition path in M form the *next-state transitions* of M.

Definition 35 Let $M = \langle T, B, O, A, A_T, A_F, \mathsf{Signals}, \mathsf{Labels}, \mathsf{CurrStates}\rangle$ be an ASM chart and let p be a conditional output path $j_0, \ldots, j_k$ in M. An *output condition* corresponding to p is a tuple $\langle j_0, m, c_t, c_f\rangle$ where

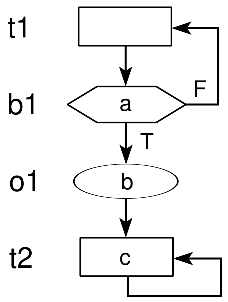
1. m is a name such that $(j_k, m) \in \mathsf{Signals}$.

2. c_t contains all signal names n for which there exists $0 < i < k$ such that j_i is a conditional branch object, $(j_i, n) \in \mathsf{Signals}$, and $(j_i, j_{i+1}) \in A_T$.

3. c_f contains all signal names n for which there exists $0 < i < k$ such that j_i is a conditional branch object, $(j_i, n) \in \mathsf{Signals}$, and $(j_i, j_{i+1}) \in A_F$.

The set of all output conditions corresponding to some conditional output path in M form the *output conditions* of M.

Example 13 Suppose the objects in the ASM chart are referred to by the labels on the left.



The chart has next-state transitions $\langle t_1, t_1, \emptyset, \{a\}\rangle$, $\langle t_1, t_2, \{a\}, \emptyset\rangle$, and $\langle t_2, t_2, \emptyset, \emptyset\rangle$. It has output conditions $\langle t_1, b, \{a\}, \emptyset\rangle$ and $\langle t_2, c, \emptyset, \emptyset\rangle$.

In order to relate ASM charts to devices, the signals in the charts need to be assigned values that can be related to those on wires in devices.

Definition 36 Let M be an ASM chart. A *signal-value assignment* for M is a function from the internal and external signals of M to the set $\{0, 1\}$. An *external signal-value assignment* is the restriction of a signal-value assignment to the external signals of M.

Assignments to signals can now be related to transitions and output conditions.

Definition 37 Given ASM chart M, state t in M, and external signal-value assignment i for M:

1. Next-state transition $\langle t_s, t_t, c_t, c_f\rangle$ is *satisfied by t and i* if $t_s = t$, $i(n) = 1$ for all names $n \in c_t$, and $i(n) = 0$ for all names $n \in c_f$. Any next-state transition satisfied by t and i is called a *next state of M under t and i.*

2. Output condition $\langle t_s, n, c_t, c_f\rangle$ is *satisfied by t and i* if $t_s = t$, $i(n) = 1$ for all names in c_t, and $i(n) = 0$ for all names in c_f.

Definition 38 An output condition is said to *correspond* to a next-state transition iff the values required to satisfy the next-state transition also satisfy the output condition.

Definition 39 Given a transition path, let S be the set of all output conditions corresponding to that path. Let E_t be a conjunction of all internal signals assigned in a condition in S. Let E_f be a conjunction of the negations of all internal signals not assigned in a condition in S. Boolean expression E over internal signals in an ASM chart *holds* along the transition path iff $E_t \wedge E_f \rightarrow E$.

The relationship between ASM charts and physical devices relies on two notions.

Definition 40 An ASM chart is *deterministic* if no two next state transitions are satisfied by the same state t and signal-value assignment i. An ASM chart is *transitionally complete* if for all states t and all signal-value assignments i there exists a next-state transition that is satisfied by t and i.

Theorem 3 *Any ASM chart is both deterministic and transitionally complete.*

Proof Let M be an ASM chart. Since redundant signal testing is eliminated in the conditions on ASM charts, the same state and signal-value assignments can only satisfy two distinct next state transitions if there exists

- A state object or conditional branch object serving as the source of two arrows to different targets, or
- A conditional branch object serving as the source for two distinct true arrows or two distinct false arrows.

As the restrictions on ASM charts rule out both cases, M must be deterministic.

Since a signal-value assignment provides values for all signals, by definition, M can only fail to be transitionally complete if there exists a conditional branch object that is the source object for a true arrow but not for a false arrow, or vice versa. This is ruled out by the restrictions on ASM charts, so M must be transitionally complete.

□

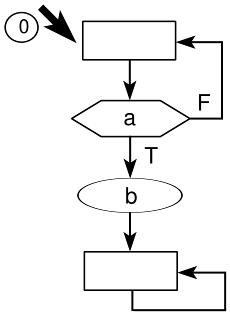


Figure 2.4: ASM Chart annotated with a start-state.

The semantics of ASM charts must account for the correlation between states of a device and state objects in the ASM chart. ASM chart states are implemented in one of two ways using unit delay elements: either each state object is assigned to a unique delay element, or the state objects are encoded with binary names, thereby allowing the number of delay elements to be logarithmic in the number of state objects. In the latter case, the number of possible states of the device is likely larger than the number of state objects appearing in the ASM chart (unless the number of states in the chart is a power of two). The semantics must handle these extra states in some reasonable fashion. In practice, a default "start state" is often chosen, with an implicit transition made from any undepicted state to this default start state.

The current state symbol with the discrete time expression for natural number 0 can identify a start state as shown in Figure 2.4. An ASM chart need not be annotated with a start state in order to be considered well-formed. One that is so annotated, however, is called a *grounded ASM chart*. The semantics of grounded ASM charts checks for transitions in the underlying device from any undepicted state to the indicated start state, as defined below.

Definition 41 Let $M = \langle T, B, O, A, A_T, A_F, \mathsf{Signals}, \mathsf{Labels}, \mathsf{CurrStates}\rangle$ and $M' = \langle T', B', O', A', A'_T, A'_F, \mathsf{Signals}', \mathsf{Labels}', \mathsf{CurrStates}'\rangle$ be ASM charts such that M is grounded and M' is not grounded. Let t_s denote the element of $\mathsf{CurrStates}$ in M. M is called the *functional reduction* of M' iff the following conditions are satisfied:

1. M and M' are component-wise identical, with the exception of the state objects and the arrows.
2. For all $t \in T' - T$, $(t, t_s) \in A'$
3. $A \cup \bigcup_{t \in T' - T}(t, t_s) = A'$

The semantic relationship between physical devices and ASM charts can now be defined.

Definition 42 Let M be an ASM chart and D be a static device.

1. A *state map* from M to D is an injective function from the state objects of M to the possible states of D (Defn. 14).
2. Let τ be an injective function from the signal names of M to the ports of D. τ is called a *signal map* iff it maps each external signal in M to an input interface port of D and each internal signal in a to an output interface port in D. A signal map is *complete* iff every output interface port of D is in its co-domain.
3. Given a signal map τ and signal-value assignment i, assignment i' for D is *compatible* with τ and i iff $i'(p) = i(\tau^{-1}(p))$ for all ports p in the co-domain of τ.
4. A state map τ_t and a signal map τ_n are said to be *feasible for* M *and* D if for all external signal-value assignments i for M and all states t in M there exists an assignment i' for D which is compatible with τ_n and i and reflects state $\tau_t(t)$ such that:
   (a) If t' is the next state of M under t and i, then $\tau_t(t')$ is the next state of D under $\tau_t(t)$ and i'.
   (b) For each signal n asserted in an output condition satisfied by t and i, $i'(\tau_n(n)) = 1$. If there does not exist an output condition for n satisfied by t and i, $i'(\tau_n(n)) = 0$.

   If τ_t and τ_n are feasible for M and D and the converse of requirement 4a holds for all i and t, τ_t and τ_n *capture* M and D.

5. M *describes* D, denoted $D \models M$, if there exists a state map and a signal map that are feasible for M and D.

6. D is a *full structural implementation* of M, denoted $D \models_s M$, iff there exists a surjective state map and a complete signal map that capture M and D.

7. D is a *functional structural implementation* of M, denoted $D \models_{fs} M$, iff there exists an ASM graph M' such that D is a full structural implementation of M' and M is the functional reduction of M'.

8. D is a *behavioral implementation* of M, denoted $D \models_b M$, if D is behaviorally equivalent to some device D' which is a functional structural implementation of M.

These semantic definitions give rise to a notion of behavioral equivalence over ASM charts. The soundness proofs for ASM chart inference rules in Chapter 4 rely on this notion and a sufficiency result for establishing behavioral equivalence.

Definition 43 Let M_1 and M_2 be grounded ASM charts such that the number of external signals in M_1 is as least as large as the number of external signals in M_2. M_1 and M_2 are *behaviorally equivalent* iff for all devices D with at least as many input interface ports as each of M_1 and M_2 has external signals, $D \models_b M_1$ iff $D \models_b M_2$.

Lemma 4 *Let M be an ASM chart. There exists a device D that is a full structural implementation of M ($D \models_s M$) iff the number of state objects in M is a power of two.*

Proof Let M be an ASM chart.

1. Assume there exists a device D such that $D \models_s M$ (Defn. 42). By definition, if $D \models_s M$, there exists a bijective state map from the state objects of M to the possible states of D. The set of possible states of D is defined to be the set of possible assignments to the output ports

of D. Since assignments have $\{0,1\}$ as a range, the number of possible states of D must be a power of two. For a bijection to exist between the possible states of D and the state objects of M, the number of state objects in M must be a power of two.

2. Assume the number of state objects in M is a power of two. This result follows from the state-encoded implementation technique presented in [70].

□

The following lemma provides a sufficiency result for behavioral equivalence of two ASM charts.

Lemma 5 *Let M_1 and M_2 be grounded ASM charts such that the number of states in M_1 is at least as large as the number of states in M_2, and the external signals of one of M_1 and M_2 is a subset of the external signals of the other. M_1 and M_2 are behaviorally equivalent if there exists a function τ from the states of M_1 to the states of M_2 such that*

1. *τ maps the start state of M_1 to the start state of M_2,*
2. *Whenever a next transition $\langle t_s, t_t, c_t, c_f \rangle$ is satisfied under assignment A in M_1, there is a transition of the form $\langle \tau(t_s), \tau(t_t), c'_t, c'_f \rangle$ that is satisfied under A in M_2, and*
3. *An output condition $\langle t_s, n, c_t, c_f \rangle$ is satisfied under assignment A in M_1 iff there is an output condition of the form $\langle \tau(t_s), n, c'_t, c'_f \rangle$ that is satisfied under A in M_2.*

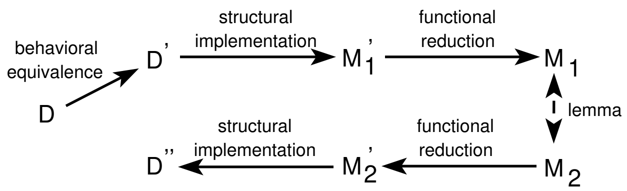
Proof Let M_1 and M_2 be ASM charts such that the number of states in M_1 is at least as large as the number of states in M_2, and the external signals of one of M_1 and M_2 is a subset of the external signals of the other.

Assume there exists a function τ with the properties stated in the lemma. Assume $D \models_b M_1$. By Definition 42, D is behaviorally equivalent to some device D' which is a functional structural implementation of M_1. D' may be chosen so as to have the same number of input interface ports as

D. By definition, there exists an ASM chart M_1' such that D' is a full structural implementation of M_1' and M_1 is the functional reduction of M_1'.

Let M_2' be the ASM chart constructed from M_2 by adding a non-zero number of new state objects such that the number of state objects in M_2' is a power of two. For each new state object b, add an arrow (b, b_g) to M_2', where b_g is the state object annotated with the start state symbol; b_g must exist by definition since M_2 is grounded. Set CurrStates of M_2' to be empty. M_2 is the functional reduction of M_2' by Definition 42.

By construction, since the number of state objects in M_2' is a power of two, there exists a device D'' such that $D'' \models_s M$. The following figure displays the relationship between the various ASM charts and machines introduced in the proof so far:



Establishing that D' and D'' are behaviorally equivalent is sufficient to complete the proof, given the transitivity of behavioral equivalence. Since the states of M_1 and M_2 are a subset of the states of M_1' and M_2', τ can be extended to a function mapping the states of M_1' to the states of M_2'. For any state of M_1' that is not in M_1, τ will assign a state in M_2' that is not in M_2; such states are guaranteed to exist by construction.

D' and D'' must have the same number of output interface ports since the properties of the lemma force M_1 and M_2 to have the same internal signals and the definition of $\models_s$ (Definition 42) requires a bijective mapping between internal signals and output interface ports. These mappings also provide a mapping from the output interface ports of D'' to the output interface ports of D' such that one port is mapped to another if they correspond to the same internal signal.

A partial mapping from the input interface ports of D'' to those of D' is defined in a similar fashion. The mapping may be partial because M_1 and M_2 are not required to have the same sets of external signals. Specifically, if p'' in D'' is mapped to by external signal a in the signal map for M_2 and a is also an external signal in M_1, p'' maps to the port p' in D' that is mapped to by a in the signal map for M_1.

Using these two mappings, construct device D''_I to be isomorphic to D'' and defined over a subset of the interface ports of D' such that if p'' maps to p''_i in the isomorphism, then p''_i is the port of D' mapped to by p'' in the above interface port mappings. Any input interface port of D''_I not covered by the interface port mappings does not need to be renamed.

D' and D''_I therefore have the same output interface ports and either the input interface ports of D' are a subset of those of D''_I or the input interface ports of D''_I are a subset of those of D'. Assume the input interface ports of D''_I are a subset of those of D'. The proof for the alternative case is analogous.

Let $\langle D', i' \rangle$ be a dynamic device. Let $\langle D''_I, i'' \rangle$ be a dynamic device such that i'' and i' agree on the values of all common input interface ports. Let i'' also be restricted such that the state reflected by i'' is mapped to in the state map by state t'_2 in M'_2, where t'_2 equals $\tau(t'_1)$, where t'_1 is the state reflected by i'. Showing $\langle D', i' \rangle$ and $\langle D''_I, i'' \rangle$ to be behaviorally indistinguishable is sufficient.

Let I be an assignment sequence for $\langle D', i' \rangle$. The output of the run of $\langle D', i' \rangle$ under I must be shown equivalent to the output of the run of $\langle D''_I, i'' \rangle$ under I. The output of $\langle D', i' \rangle$ must be the same as the output of $\langle D''_I, i'' \rangle$. Let n be any internal signal of M_1; by construction, n must be an internal signal of M'_1. By the definition of $\models_s$ (Defn. 42), the value in i' on the port corresponding to n in D' must be 1 iff there is an output condition for n satisfied in state t'. By the assumptions of the lemma, there is an output condition for n satisfied in state t' iff there is an output condition for n satisfied in state t''. By the definition of $\models_s$ (Defn. 42), the value in i'' on the port corresponding to n in D''_I must be 1 iff there is an output condition for n satisfied in state t''. Thus the output of $\langle D', i' \rangle$ must be the same as the output of $\langle D''_I, i'' \rangle$.

This result must be extended to the remaining dynamic devices in each run under I. This is done by showing that, at each stage of the run, the states of D' and D''_I are consistent with τ and the state mappings on M'_1 and M'_2. However, this follows trivially by the definition of τ, which only exists if the required relationship holds between the states. D' and D'' must therefore be behaviorally equivalent, which completes the proof.

The case for $D \models_b M_2$ is analogous. □

Proofs involving ASM charts often examine only particular sequences of state transitions. As these sequences correspond naturally to runs of devices, an ASM chart can describe a particular run of a device. This relies on the notion of a run of an ASM chart:

Definition 44 A *configuration* of ASM chart M is a tuple $\langle M, t, i \rangle$ where t is a state object in M and i is an external signal assignment for M. A *run* of M is a sequence $M_1, \ldots, M_k$ of configurations for M such that the next state of M under t_j and i_j is t_{j+1} for all $1 \leq j < k$.

Definition 45 Let M be an ASM chart without any $\mathsf{CurrStates}$, D be a device, $R_D = D_{Y_1}, \ldots, D_{Y_k}$ be a run of D and τ_t and τ_n state and signal maps for M and D. Let R_M be the sequence of configurations $M_1, \ldots, M_k$ for M such that for all $1 \leq j \leq k$, t_j is τ^{-1} of the state of D_{Y_j} and i_j is τ_n^{-1} applied to the assignment in D_{Y_j}. M *run-describes* R_D iff R_M is a run of M.

Given ASM chart M and device D, it is straightforward to show that if D is a full structural implementation of M, then M run describes all possible runs of D. The structural implementation relationship may therefore be viewed as a form of quantification over runs. Annotating an ASM chart with the current state marker isolates particular runs. The semantics of time-annotated ASM charts are therefore defined with respect to runs of devices.

Definition 46 Let M be a time-annotated ASM chart with $\mathsf{CurrStates}$ cs. Let $R_D = D_{Y_1}, \ldots, D_{Y_k}$ be a run of a device D and let τ_t and τ_n be state and signal maps for D and M.

1. If M is annotated with discrete time expression e containing no free variables, M describes R_D, denoted $R_D \models M$, iff $D \models_s M$ and the state of D_{Y_e} is $\tau_t(cs)$.

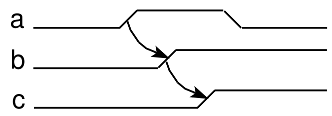
2. If M is annotated with discrete time expression e containing free variables $v_1, \ldots, v_n$, let $c_1, \ldots, c_n$ denote natural number constants and let $e[c_1, \ldots, c_n]$ denote e with each v_i replaced by c_i. M describes R_D relative to $[c_1, \ldots, c_n]$, denoted $R_D \models M[c_1, \ldots, c_n]$, iff $D \models_s M$ and the state of $D_{Y_{e[c_1,\ldots,c_n]}}$ is $\tau_t(cs)$.

3. If M is not annotated with a discrete-time expression, M describes R_D, denoted $R_D \models M$, iff $D \models_s M$ and the state of D_{Y_1} is cs.

2.2.3 Timing Diagrams

Timing diagrams depict transitions on signals and constraints on the temporal relationships between transitions. Although the types of information depicted are relatively uniform, neither their syntax nor their semantics are standardized in practice. The most challenging aspect of timing diagram formalization is the definition of the semantics, as timing diagrams are subject to several interpretations. A demonstration appears in Example 14.

Example 14 Consider the following timing diagram, in which arrows intuitively denote temporal ordering. Questions arising when assigning semantics to this diagram include:



1. Is the timing diagram an invariant, meaning must it hold all the time or only some of the time?

2. If the timing diagram is not an invariant, does it hold only once, or should it hold more than once? Must it hold only for a particular duration of time or can it hold at various points of

time throughout the execution of the system?

3. If it is an invariant, when can the next iteration begin? Is a allowed to rise again before c rises, or is a restricted to only rise again after c has risen?

4. Does this diagram say "a rises, then b rises, then c rises", or "if a rises, then b rises, then c rises", or "if a rises, then if b rises, then c rises"?

5. Can a single rising edge on b apply to two rising edges on a, or is a one-to-one relationship on events required?

6. Is the falling edge on a required to occur? If so must it occur after the rising edge on b, or can it occur before the rising edge on b?

7. Are a, b, and c required to start as low values, or can the initial conditions waver?

8. Are b and c required to return to low values before a repetition can occur? In other words, must the left and right edges of the diagram match values for the diagram to specify repetition?

Some of the questions in Example 14 are easy to answer: timing diagrams do generally depict invariants and all depicted transitions should be required to occur. Question 3 is challenging because more than one interpretation is used in practice. Databooks use timing diagrams to specify timing restrictions on read/write cycles of memory components [25]. In such components, one cycle completes before the next one begins: the timing diagram is therefore viewed as an infinite concatenation. In contrast, timing diagrams are also being used as graphical depictions of temporal logic to state properties for verification [74]. The temporal logic interpretation views timing diagrams as being *time invariant*, meaning that the depicted relationship should hold in every state of a computation, regardless of the status of previous iterations [68].

Choosing between the time invariant and concatenation interpretations is analogous to choosing between natural support for verification or natural support for design. Rather than choose, this section defines two semantic relationships: one time invariant and one iterative. These are discussed after the presentation of timing diagram syntax.

Syntax

Timing diagrams are built around invisible vertical lines referred to as *time points.* Time points are used to specify coordinates for the lines that comprise waveforms. Waveforms contain two types of primitive syntactic objects: *level objects* and *transition objects.* Intuitively, level objects specify voltage levels, while transition objects specify changes in voltage levels.

Definition 47 A *waveform* W is a tuple $\langle P, H, L, U, R, F, N \rangle$ where

1. P is a nonempty, finite, horizontally arranged, and totally ordered collection of time points, $p_0, \ldots, p_k$. p_0 and p_k are called the *minimal* and *maximal* points of W, respectively.

2. H, L, and U are disjoint subsets of $P \times P$ called the *high-level objects*, *low-level objects*, and *unknown-level objects*, respectively; they are collectively referred to as the *level objects* of W. High and low level objects are depicted using horizontal lines. Unknown level objects are depicted using horizontally stretched diamonds.

High-Level Object   Low-Level Object   Unknown-Level Object

3. R and F are disjoint subsets of $P - \{p_0, p_k\}$ called the *rising-transition objects* and the *falling-transition objects* of W, respectively. They are collectively called the *transition objects* of W. Rising-transition objects are denoted by positively sloped lines and falling-transition objects are denoted by negatively sloped lines.

Rising-Transition Object   Falling-Transition Object

4. N is an element of the set $\mathsf{SigNames}$.

The well-formedness constraints on waveforms guarantee that level objects do not overlap, that the waveform is continuous, and that the depicted levels and transitions make sense: for example, a rising transition should not connect two high-level objects.

Definition 48 A waveform $W = \langle P, H, L, U, R, F, N \rangle$ is *well-formed* iff

1. For every level object (p_a, p_b) in W, $p_a < p_b$.

2. There do not exist level objects (p_a, p_b) and (p_c, p_d) in W such that $p_a \leq p_c < p_b$.

3. Unless $p_b = p_k$, for any level object (p_a, p_b) in W, there exists a time point p_c in P such that there exists a level object (p_b, p_c) in W.

4. Unless $p_b = p_0$, for any level object (p_b, p_c) in W, there exists a time point p_a in P such that there exists a level object (p_a, p_b) in W.

5. There exist time points p_a and p_b such that (p_0, p_b) and (p_a, p_k) are level objects in W.

6. $p \in R$ iff $(p_a, p) \in L$ and $(p, p_b) \in H \cup U$ or $(p_a, p) \in U$ and $(p, p_b) \in H$.

7. $p \in F$ iff $(p_a, p) \in H$ and $(p, p_b) \in L \cup U$ or $(p_a, p) \in U$ and $(p, p_B) \in L$.

8. There do not exist level objects (p_a, p_b) and (p_b, p_c) in W that are both objects of the same type.

In addition to these constraints, name labels are assumed to appear to the left of the waveform. The name labeling a waveform will sometimes be used interchangeably with the mathematical object capturing the waveform.

Timing diagrams consist of well-formed waveforms and notations to specify succession and concurrency between objects on waveforms. Succession relationships may be constrained by lower and upper bounds on the number of discrete time units within which the relationship should occur.

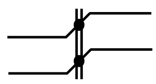
Definition 49 A *time-bound expression* is a pair of the form $[l, u]$, where l is a positive natural number constant, a positive natural number variable, or a sum of these and u is a positive natural number constant, a positive natural number variable, a sum of these, or the symbol ∞. A time-bound expression is called *simple* if l and u are only constants, variables, or ∞, with no sums. A time-bound expression containing a sum is called *complex*.

Definition 50 A *timing diagram* is a tuple $\langle P, W, E_S, E_C, B, A\rangle$ where

1. P is a nonempty, finite, totally ordered collection of time points.

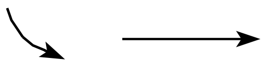
2. W is a finite set of waveforms, each defined over P.

3. E_C is a subset of $P \times \mathcal{P}(W)$ denoted by double parallel lines and called the *concurrent edge objects*. The waveforms included in the second component are denoted by small black circles on the intersection of the waveforms and the parallel lines.



Concurrent Edge Object

4. E_S is a subset of $J \times J$, where J is the union of E_C and the transition objects contained in each $w \in W$, denoted by arrows and called the *successor edge objects*. The first and second components of each successor edge object are called the *source* and *target* objects, respectively.

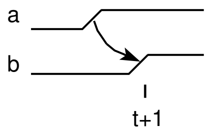


Successor Edge Objects

5. B is a partial function from E_S to time-bound expressions called the *time bound objects*. The icons pictured below represent time-bound expressions $[l, u]$, $[l, l]$, $[1, u-1]$, and $[l+1, \infty]$ attached to elements of E_S, respectively.

Time Bound Objects

6. A is a partial function from transition objects, level objects, and concurrent edge objects to discrete time expressions called the *time anchors.* Time anchors are denoted by small vertical lines directly under the object on the waveform to be related, labeled by the discrete time expression. If A is the empty function, the timing diagram is called *anchor-free.*



Successor edge objects may be referred to by the time points, rather than the actual objects, they connect. When a pair of time points is treated as a successor edge object, it is assumed that there exists a successor edge object whose components lie at the given time points. In addition, it is helpful to refer directly to the objects related by a concurrent edge object.

Definition 51 Let (p, W') be a concurrent edge object in timing diagram T. Define the set of *concurrent objects* of (p, W') as follows:

1. If $w \in W'$ and w is defined over p, then the transition object in w denoted by p is in the set.

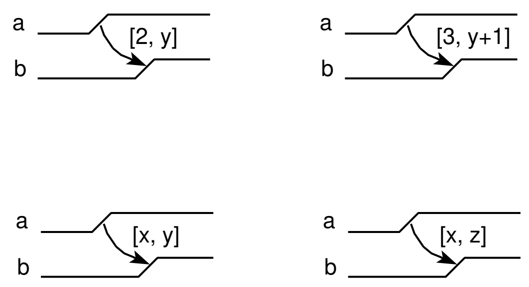
2. If $w \in W'$ and w is not defined over p, then the level object denoted by (p_a, p_b) is in the set, where p_a is the largest time point of T over which w is defined that is smaller than p and p_b is the smallest time point of T over which w is defined that is larger than p.

Timing diagrams sometimes differ from one another only with respect to the time-bound objects assigned to edges. Such diagrams are called *event isomorphic.* Within sets of pairwise-event isomorphic timing diagrams, the time bound expressions assigned to a particular edge by each diagram are only comparable if they involve the same variables. These notions are formalized in the following definitions.

Definition 52 Let $T_1 = \langle P_1, W_1, E_{S_1}, E_{C_1}, B_1, A_1 \rangle$ and $T_2 = \langle P_2, W_2, E_{S_2}, E_{C_2}, B_2, A_2 \rangle$ be timing diagrams. T_1 and T_2 are *event isomorphic* iff $P_1 = P_2$, $W_1 = W_2$, $E_{S_1} = E_{S_2}$, $E_{C_1} = E_{C_2}$, and $A_1 = A_2$. T_1 and T_2 are *isomorphic* if they are event isomorphic and $B_1 = B_2$.

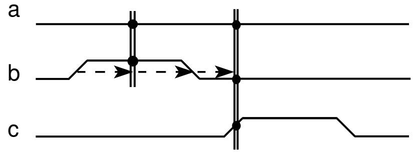
Definition 53 Let $T_1, \ldots, T_n$ be a set of pairwise event isomorphic timing diagrams. Let B_i denote the time bound objects of T_i. $T_1, \ldots, T_n$ are *time bound expression compatible for a successor edge object e* if the lower bound in every $B_i(e)$ contains the same variables and the upper bound in every $B_i(e)$ contains the same variables. $T_1, \ldots, T_n$ are *time bound expression compatible* if they are time bound expression compatible for all of their successor edge objects.

Example 15 The diagrams in the top pair are event isomorphic and time-bound expression compatible. The diagrams in the bottom pair are event isomorphic but are not time-bound expression compatible since the bottom left diagram has y as an upper bound and the bottom right diagram has z as an upper bound.

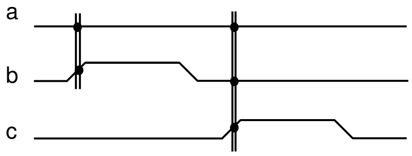


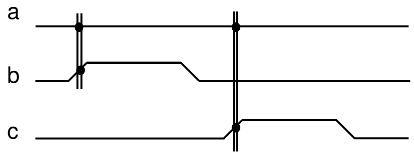
Semantically, successor edge objects force the target object to occur after the source object. A timing diagram contains several implicit successor relationships. Within a single waveform, there is an implicit successor relationship between consecutive transition objects. Implicit successor edges also exist between concurrent edge objects and the transition objects surrounding the objects on the waveforms to which they apply.

Example 16 The successor edge objects drawn as dashed arrows are implicit in the diagram.



The horizontal ordering of concurrent edge objects in a timing diagram does not necessarily imply temporal ordering between them.





The diagram on the left has an implicit successor relationship between the two concurrent edge objects. The diagram on the right does not. The difference between the two diagrams is whether the second concurrent edge object is attached to waveform b. Without the small black circle on b, the two concurrent edge objects do not relate different objects on the same waveform, and thus will not force a successor relationship semantically. Consecutive time points therefore cannot be assumed to generate implicit successor relationships.

The following definitions capture when all of a timing diagram's implicit temporal relationships are depicted by explicit successor edge objects.

Definition 54 Let $T = \langle P, W, E_S, E_C, B, A \rangle$ be a timing diagram. T is *edge-explicit* iff

1. For each level object (p_a, p_b) in waveform w such that $p_a \neq p_0$ and p_b is not the maximal point of w, $(b_1, b_2) \in E_S$, where b_1 and b_2 are the transition objects in w at p_a and p_b, respectively.

2. For every concurrent edge object (p, W) and for every $w \in W$, if the concurrent object at w is a level-object, there are successor edge objects $((p, W), p_a)$ and $(p_b, (p, W))$ where (p_a, p_b) is a

level object in w and $p_a < p < p_b$.

Definition 55 Let T and T' be timing diagrams over the same set of time points and assume T' is edge-explicit. T' is the *edge-explicit extension* of T iff for every successor edge object (p_a, p_b) in $T' - T$, either

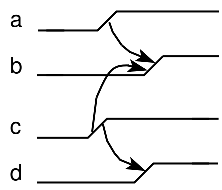
1. There exists a waveform $w \in T$ for which (p_a, p_b) is a level object, or
2. There is a concurrent edge object (p_a, W) and p_b is a transition object on some waveform $w \in W$ such that $p_c < p_a < p_b$, where (p_c, p_b) is a level object in w, or
3. There is a concurrent edge object (p_b, W) and p_a is a transition object on some waveform $w \in W$ such that $p_a < p_b < p_c$, where (p_a, p_c) is a level object in w.

Once a timing diagram is edge-explicit, it is syntactically clear which objects must occur before which other objects semantically. In particular, the first and last objects to occur are easily identifiable, as are the objects that must directly precede a given object.

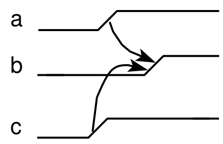
Definition 56 The *tail objects* of an edge-explicit timing diagram T are those target objects of successor edge objects that are not the source object for another successor edge object. The *head objects* of T are those source objects of successor edge objects that are not the target object for another successor edge object.

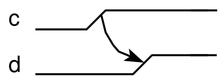
Definition 57 Let b be an object in an edge-explicit timing diagram T. The *enabling objects* of b are all transition objects and concurrent edge objects b' in T such that (b', b) is in the transitive closure of E_S.

Example 17 The rising transition object on b has the rising transition objects on a and c as enabling objects and the rising transition object on d has the rising transition object on c as its enabling object. The rising transition objects on b and d are both tail objects.



Intuitively, the timing diagram in Example 17 could be split into the following two *single-tail* diagrams, one corresponding to each tail object and its respective enabling objects.



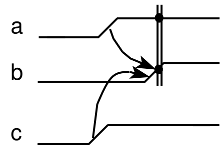


The semantic relationship between a timing diagram and its corresponding single-tail diagrams is discussed in Chapter 3, Section 3.2. For now, the single-tail diagrams must be defined explicitly.

Definition 58 Let $T = \langle P, W, E_S, E_C, B, A\rangle$ be a timing diagram with multiple tail objects. Timing diagram $T' = \langle P', W', E'_S, E'_C, B', A'\rangle$ is a *single-tail diagram arising from* T iff $P' = P$, $W' \subseteq W$, $E'_C \subseteq E_C$, $E'_S \subseteq E_S$ such that T' contains exactly one tail object t, the enabling objects of t are the same in each of T and T', and T' does not contain any transition object or concurrent edge object other than t and its enabling objects.

Some timing diagrams, such as the one in Example 18, contain objects that cannot be related through edge-explicit extensions.

Example 18 Edge-explicit extensions cannot relate the rising transition objects on a and c.



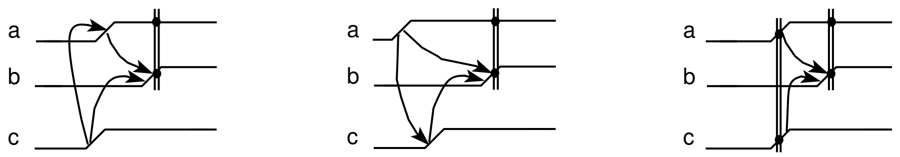
Timing diagrams containing such objects are called *temporally ambiguous*. As shown in later sections, it is often useful to work only with temporally unambiguous timing diagrams. Such diagrams can be formally defined as follows:

Definition 59 Let T be a timing diagram and let $T' = \langle P', W', E'_S, E'_C, B', A' \rangle$ be the edge-explicit extension of T. T is *temporally unambiguous* iff for every pair of time points p_a and p_b corresponding to transition objects in T', either (p_a, p_b) or (p_b, p_a) is in the transitive closure of E'_S. A timing diagram that is not temporally unambiguous will be called *temporally ambiguous*.

Note that a temporally unambiguous timing diagram contains only a single head object. Given a temporally unambiguous timing diagram, its *prefix* relative to time point p_i is all time points up to and including p_i, plus any successor edge objects, concurrent edge objects, time bound expressions, and time anchors defined on those time points. The time points in a temporally unambiguous timing diagram are assumed ordered numerically with respect to the order of their events; renaming can be applied to any temporally unambiguous timing diagram whose time points do not meet this criterion.

Any temporally ambiguous timing diagram corresponds to a collection of temporally unambiguous timing diagrams, simply by creating one temporally unambiguous timing diagram for each possible temporal ordering of unrelated events.

Example 19 These three diagrams show the possible relationships between the rising transitions on a and c from the diagram in Example 18.



The following algorithm formalizes how to obtain the set of temporally unambiguous diagrams from an arbitrary timing diagram.

Algorithm 2

1. Let T be a timing diagram and initialize set S to contain only T.

2. Let $T' = \langle P', W', E'_S, E'_C, B', A' \rangle$ be any temporally ambiguous timing diagram in S. Let b_1 and b_2 be two transition objects or concurrent edge objects not related in $Cl(T')$. Remove T' from S and replace it with T'_1, T'_2 and T'_3, where

   (a) $T'_1 = T'$ plus successor edge object (b_1, b_2).

   (b) $T'_2 = T'$ plus successor edge object (b_2, b_1).

   (c) If b_1 and b_2 are both transition objects, then $T'_3 = T'$ with the exception that the time point of b_2 be changed to that of b_1 and concurrent edge object $(p, \{w_1, w_2\})$ be added where p is the time point of b_1 and w_1 and w_2 are the waveforms corresponding to b_1 and b_2, respectively. Otherwise, without loss of generality assume b_1 is a concurrent edge object. If b_2 is a transition object, change the time point of b_2 to that of b_1 and add the waveform corresponding to b_2 to the waveforms included in b_1. If b_2 is a concurrent edge object, change the time point of each transition object in b_2 to that of b_1, union the waveforms included in b_2 into those included in b_1, and remove b_2.

   (d) Repeat until S contains only temporally unambiguous diagrams.

The well-formedness conditions on timing diagrams cover intuitive conditions, such as requiring arrows to point in the proper direction and concurrent edge objects to relate at least two events. They also require that some transition object or concurrent edge object be defined at each time point and that the time-bound expressions be satisfiable.

Definition 60 A timing diagram $T = \langle P, W, E_S, E_C, B, A \rangle$ is *well-formed* iff the following conditions are satisfied:

1. For all $(b_1, b_2) \in E_S$, the time point corresponding to b_1 is lower in the time point ordering than the time point corresponding to b_2.

2. No concurrent edge object is defined at either the minimal or maximal time points.

3. The set of concurrent objects for each element of E_C contains at least two distinct objects.

4. No element of $\mathsf{SigNames}$ labels more than one waveform.

5. There exists a function ϕ from temporal variables to natural numbers such that for all pairs of time anchors (b_1, e_1) and (b_2, e_2) such that (b_1, b_2) is in the transitive closure of E_S, if there exists time bound $[l, u]$ associated with (b_1, b_2), it is not the case that $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_1)[\phi] < l$ or $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_1)[\phi] > u$ (TEval appears in Defn. 21).

6. For all time points p_i, either there exists concurrent edge object (p_i, W) or some waveform has a transition object at p_i.

Henceforth, the term *timing diagram* refers to a well-formed timing diagram.

It is decidable whether a function ϕ exists that meets the requirements of Step 5. Algorithm 5 in Chapter 3 shows how to construct a set of linear inequalities corresponding to the bounds restrictions in a temporally unambiguous timing diagram, the satisfiability of which is decidable by a procedure due to Shostak [77]. The time-bounds expressions in a temporally ambiguous timing diagram are satisfiable if the time-bound expressions in any of its corresponding temporally unambiguous diagrams are satisfiable.

Semantics

Timing diagram semantics are defined relative to the language model. The signals assigned in the language presumably include all elements of $\mathsf{SigNames}$ used as waveform labels within the timing diagram in question. In giving the semantics related to time-bound objects, the syntactic natural number constants correspond to their standard natural numbers, and symbols ∞, $=$, $<$, $>$, and $+$ correspond to infinity, equals, less than, greater than, and addition, as expected.

Timing diagrams describe words when the transition patterns in the diagram repeatedly reflect signal changes in the words. Timing diagrams are viewed as assume-guarantee relationships on sets of objects; that is, whenever a word describes some portion of the diagram (the assume portion), it must describe the entire diagram (the guarantee condition).

HHL contains several timing diagram semantics, defined by three parameters: the set of objects that comprise the assume portion, when the assume portion is checked against the word, and how strictly the level objects in the diagram must be matched in the word. Fluctuations in each of these parameters arise in practice. For example:

- A timing diagram that is derived from a circuit diagram would assume the waveforms for the inputs to the circuit, guarantee the waveforms for the remaining signals in the circuit, have this relationship hold at all times, and have level objects represent exactly what signal patterns hold in the circuit.
- A timing diagram providing the specification for a memory chip interface might have no assume portion, instead treating the entire diagram as behavior that must be guaranteed repeatedly. Such diagrams represent cycles of behavior, rather than behaviors intended to hold at each individual time step in a computation. Level objects are also interpreted strictly in this case; that is, a signal depicted as low is required to be low, rather than being allowed to waver.
- A timing diagram used to state a liveness condition in verification would have varying sets of assume objects, require the depicted relationship to hold at all times, and not care how the signals fluctuated as long as the depicted objects occurred as specified.

The definitions in this section leave these parameters uninstantiated. Two description relationships are defined between timing diagrams and words: one in which the timing diagram patterns are required to hold at all times (the invariant semantics) and one in which the patterns are required to hold in a concatenation fashion (the iterative semantics). Each definition is given with respect to parameters for the partitions and the degree of strictness in the level-object interpretations. One other variation on timing diagram semantics exists, depending upon whether the diagram is anchor-free.

Timing diagrams with time anchors are viewed as referring to particular moments of discrete time; repetition is therefore meaningless in such diagrams. Special semantic relationships are defined at the end of the section for handling timing diagrams containing time anchors.

The key to all of the semantic definitions is a function assigning indices in a word to objects in a timing diagram.

Definition 61 A function from transition objects and concurrent edge objects to the natural numbers is called an *object-index assignment.* The term B*-object-index assignment* represents an object-index assignment with domain B. For any transition object included in a concurrent edge object, an object-index assignment must assign the same index to both the transition object and the concurrent edge object.

The assume-guarantee interpretation of timing diagrams requires that an object-index assignments on the assume portion be extendible to an object-index assignment on the entire diagram. Object-index assignment I can be *extended* to object-index assignment I' iff the domain of I is a subset of the domain of I', and I and I' agree on the values of all objects in the domain of I.

An object-index assignment is only useful if the index it assigns to an object is consistent with the signal values required by the object; for example, an index assigned to a rising transition object should contain a low signal value, with a high signal value in the next index. The following definition captures when an index is consistent with the signal value requirements of an object.

Definition 62 Let $T = \langle P, W, E_S, E_C, B, A \rangle$ be a timing diagram and let b be a level object, transition object or concurrent edge object in T. An index i into a word D *satisfies* b iff

1. If b is a high-level object in a waveform w in W, then $D_i(w) = 1$.

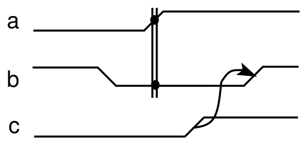
2. If b is a low-level object in a waveform w in W, then $D_i(w) = 0$.

3. If b is an unknown-level object in a waveform w in W, then $D_i(w) = 0$ or $D_i(w) = 1$.

4. If b is a transition object in a waveform w in W at time point p, i satisfies (p_a, p) and $i+1$ satisfies (p, p_b) where (p_a, p) and (p, p_b) are level objects in w.

5. If b is a concurrent edge object (p, W), then for each $w \in W$ i satisfies the concurrent object of b on w.

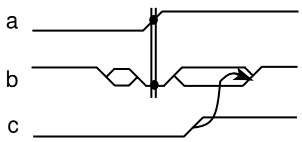
Example 20 Index 1 satisfies the falling transition object on b. Index 3 satisfies the concurrent edge object, but index 7 does not because b has value 1 in index 7.



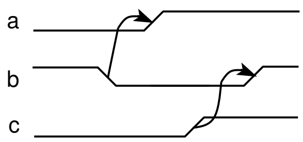
a	0	0	0	0	1	1	1	0	1
b	0	1	0	0	0	0	0	1	1
c	0	0	0	0	0	0	1	1	1
	0	1	2	3	4	5	6	7	8

Extending index-level satisfaction to word-level satisfaction depends on the interpretation of level objects. Level objects, as discussed earlier, should sometimes be interpreted strictly and other times interpreted more loosely, depending upon the context. Concurrent edge objects also affect how level objects are interpreted. HHL interprets the timing diagram in Example 20 as saying "after b falls, it remains low until a rises", instead of simply "b is low when a rises". Giving concurrent edge objects an "until" interpretation in addition to their concurrency interpretation provides a means of specifying locally strict interpretations in non-strictly interpreted timing diagrams.

There are two ways to override the until interpretation of concurrent edge objects. Suppose the desired interpretation of the timing diagram in Example 20 is "b must be low when a rises, but unrestricted between the fall on b and the rise on a". The timing diagram could be augmented with "don't care" objects,



or the heterogeneous nature of HHL could be exploited, and the following pair of timing diagram and CTL formula used:



$\mathsf{AG}((\neg a \wedge \mathsf{AX}(a)) \rightarrow \neg b)$

The definitions for when an object-index assignment satisfies a timing diagram are now given, one for each extreme interpretation of level-object satisfaction. Each definition requires that the individual indices assigned to objects satisfies those objects, that the time bounds reflected in the diagram are satisfied, that any differences between the discrete time expressions used in time anchors are satisfied, and that any until conditions generated by concurrent edge objects are satisfied. The two definitions differ only in their treatment of level objects not involved in concurrent edge objects.

Definition 63 Let $T = \langle P, W, E_S, E_C, B, A \rangle$ be a timing diagram and let I be an object-index assignment for T. *I satisfies the constraints of T* relative to a function ϕ from temporal variables in T to natural numbers iff

1. For each transition object and concurrent edge object $b \in T$, $I(b)$ satisfies b.

2. For each variable involved in a time bound expression in T, the variable can be replaced by a natural number constant such that for each $(b_1, b_2) \in E_S$, if B associates a time-bound object containing expression $[l, u]$ with (b_1, b_2), then $l \leq I(b_2) - I(b_1) \leq u$. If b does not associate a time-bound object with (b_1, b_2), then $I(b_1) < I(b_2)$.

3. For each pair of time anchors (b_1, e_1) and (b_2, e_2), it is not the case that $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_2)[\phi] \neq I(b_2) - I(b_1)$. If B associates a time-bound object containing expression $[l, u]$ with (b_1, b_2), then it is not the case that $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_2)[\phi] < l$ or $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_2)[\phi] > u$.

4. For each $b = (p, W) \in E_C$ and each $w \in W$, if the object of w at p is a level object (p_a, p_c) and

$p_a \neq p_0$, then for all indices i, $I(p_a) < i < I(b)$, i satisfies (p_a, p_c). If $p_a = p_0$ and there exists $(p', W') \in E_C$ such that $p' < p$ and $w \in W'$, then for all indices $p' \leq i \leq p$, i satisfies (p_a, p_c).

5. For each $b = (p, W) \in E_C$ and each $w \in W$, if the object of w at p is a level object (p_a, p_c) and p_c is not the maximal point of T, then for all indices i, $I(b) < i < I(p_c)$, i satisfies (p_a, p_c). If p_c is the maximal point of T and there exists $(p', W') \in E_C$ such that $p < p'$ and $w \in W'$, then for all indices $p \leq i \leq p'$, i satisfies (p_a, p_c).

Definition 64 Let $T = \langle P, W, E_S, E_C, B, A\rangle$ be a timing diagram and let I be an object-index assignment for T. I *strictly satisfies the constraints of* T relative to a function ϕ from temporal variables in T to natural numbers iff

1. For each transition object and concurrent edge object $b \in T$, $I(b)$ satisfies b.

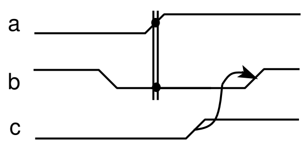
2. For each $(b_1, b_2) \in E_S$, if b_1 and b_2 are consecutive transition objects on the same waveform w, then for all $I(b_1) < i < I(b_2)$, i satisfies the level object between b_1 and b_2.

3. For each variable involved in a time bound expression in T, the variable can be replaced by a natural number constant such that for each $(b_1, b_2) \in E_S$, if B associates a time-bound object containing expression $[l, u]$ with (b_1, b_2), then $l \leq I(b_2) - I(b_1) \leq u$. If B does not associate a time-bound object with (b_1, b_2), then $I(b_1) < I(b_2)$.

4. For each pair of time anchors (b_1, e_1) and (b_2, e_2), it is not the case that $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_2)[\phi] \neq I(b_2) - I(b_1)$. If B associates a time-bound object containing expression $[l, u]$ with (b_1, b_2), then it is not the case that $\mathsf{TEval}(e_2) - \mathsf{TEval}(e_2)[\phi] < l$ or $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_2)[\phi] > u$.

In the remainder of this section, ϕ is left unspecified unless it captures particular values.

Example 21 Let I be an object-index assignment for the timing diagram that assigns index 0 to the rising transition object on c, index 1 to the falling transition object on b, index 3 to the concurrent edge object, and index 6 to the rising transition object on b. Under I, the word on the left strictly satisfies the timing diagram and the word on the right non-strictly satisfies the timing diagram.



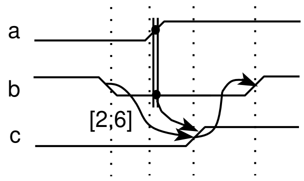
a	0	0	0	0	1	1	1	1	1
b	0	1	0	0	0	0	0	1	1
c	0	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

a	0	0	0	0	1	0	1	0	1
b	0	1	0	0	0	0	0	1	1
c	0	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

The remaining definitions in this chapter are presented using the non-strict satisfaction relationship; the definitions also hold using the strict satisfaction relationship.

A *time slice* of a timing diagram is any pair of consecutive time points. The strict and non-strict satisfaction definitions effectively place certain constraints on the indices that fall within a time slice. In the strict semantics, each index in a time slice must satisfy all the level objects depicted in that slice. In the non-strict semantics, concurrent edge objects force some level objects to be satisfied in a slice. The *signal pattern constraint* for a time slice is the set of all level objects that must be satisfied by the indices falling within the slice. A time-bound expression *spans* time slice (p_i, p_{i+1}) if the expression is associated with successor edge object $(p_j, p_{j'})$ where i is as least as large as j and j' is at least as large as $i + 1$.

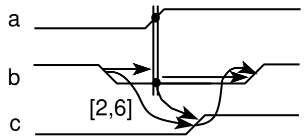
Example 22 The dotted lines demarcate the time slices in this timing diagram. The time-bound expression spans two time slices. For the time slice from the falling transition object on b to the concurrent-edge object, the signal pattern constraints contain b under non-strict satisfaction and contain a, b, and c under strict satisfaction.



The transition objects and concurrent edge objects in a timing diagram must be partitioned into an assume portion and a guarantee portion, as described earlier. A partitioning must respect the temporal ordering between the two portions; those that do are called valid.

Definition 65 Let $T = \langle P, W, E_S, E_C, B, A\rangle$ be an edge-explicit timing diagram and let P_1 and P_2 partition the transition objects and concurrent edge objects of T. P_1 and P_2 form a *valid assume-guarantee partitioning* iff there do not exist objects $b_1 \in P_1$ and $b_2 \in P_2$ such that (b_2, b_1) is in the transitive closure of E_S.

Example 23 An assume-guarantee partitioning of this timing diagram with the rising transition object on b in P_1 is valid, as is one with the rising transition objects on all three waveforms plus the concurrent-edge object. An assume-guarantee partitioning with the rising transition object on c as the only object in P_1 is not valid since that object is enabled by both the rising transition object on b and the concurrent edge object.

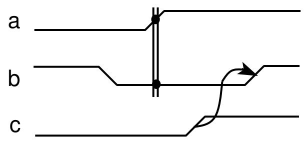


Any valid assume-guarantee partitioning implicitly partitions the level objects: any level object involved in a concurrent edge object is associated with the partition of the concurrent edge object. Any other level object is associated with the same partition as the transition object on its left endpoint. Objects with left endpoint p_0 are associated with the same partition as the object on its right endpoint. Level objects spanning the minimal to maximal points are assigned to partition P_2.

Assume-guarantee partitionings also implicitly partition the time-bound objects. The time-bound objects of P_1 are those in the restriction of the time-bound objects of T to those edge of E_S that lie in P_1. All other time-bound objects are associated with P_2. For temporally unambiguous timing diagrams, the division between P_1 and P_2 can be defined by a single time point, called the *split point*; all time points up to and including the split point belong to P_1 and all time points after the split point belong to P_2.

Both the iterative and the invariant semantic definitions are based on locating object-index assignments for timing diagrams in words and testing whether those assignments satisfy the restrictions of the timing diagram. The *next object-index match*, formally defined below, is an object-index assignment constructed by first walking a word from a particular starting index looking for indices that contain the head objects of a timing diagram. The search then continues for the first occurrences of any subsequent objects whose enabling objects have been located.

When an index satisfying an object is found, it must be determined whether the object-index assignment can be extended with the new index and still satisfy the timing diagram. For example, an index located for the concurrent edge object will only satisfy the entire timing diagram if all indices between that for the falling transition on b and the concurrent edge object have b set to zero.



Similarly, any time-bounds on the portion of the diagram for which indices have been found must be verified at each step.

If there exists an object whose first occurrence after its enabling objects fails to satisfy the timing diagram, an *error index* must be defined. The error index is used by the iterative semantics to determine where the next search should start. Intuitively, the error index is one larger than the first index at which it was determined that the next object-index match cannot satisfy the timing diagram. If an index satisfying an object was found, but that index violated a time-bounds constraint, the error index is one larger than that located for the object. If some index violated a signal-pattern that needs to hold between objects, the error index is one larger than the index violating that pattern. If the end of the word is reached before an index is found satisfying some object, the error index is one larger than the last index in the word. The error index is left undefined if all objects are assigned indices.

The next definition formalizes both the next object-index match and the error index. The definition is given in terms of a general set of objects S so that one definition can be used for both the assume and guarantee portions of a timing diagram.

Definition 66

1. Let T be a timing diagram, W be a word, i be an index into W, and S be a set of objects of T. The *next object-index match* for T, W, i, and S is the S-object-index assignment I_S for T constructed as follows:

   (a) Assign indices to the head objects $b \in S$ such that $I_S(b) = j$ iff $j \geq i$, j satisfies b, and there does not exist index k such that $i \leq k < j$ and k satisfies b. If there is no index j meeting these criteria, leave $I_S(b)$ undefined.

   (b) For any transition object or concurrent edge object b whose enabling objects have been assigned indices in I_S, if there exists an index j such that j satisfies b and j is larger than the indices assigned to the enabling objects of b, assign j to $I_S(b)$ if

      i. j is the smallest index satisfying b and larger than the indices assigned to the enabling objects of b,

      ii. the previous condition is satisfied and I_S with $I(b) = j$ satisfies the signal pattern constraints of T relative to the objects assigned in I_S, and

      iii. the previous two conditions are satisfied and I_S with $I(b) = j$ satisfies the time-bound objects of T relative to the objects assigned in I_S.

If no such j can be found, leave $I_S(b)$ undefined.

(c) Repeat Step 1b until all indices that can be assigned values have been assigned values.

2. The *error index* of I is defined as follows:

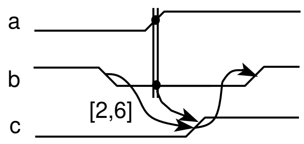
   (a) If there exists some object b such that no j exists that satisfies b and is larger than the indices assigned to the enabling objects of b, the error index is one larger than the last index in W.

   (b) If check 1(b)iii fails for some object b, the error index is one larger than the index j being tested in Step 1b.

   (c) If check 1(b)ii fails for some object b, the error index is one larger than the smallest index that violates the signal pattern constraints of T, relative to I.

   (d) Otherwise, the error index is undefined.

Example 24 Consider timing diagram



The word below, reproduced from Example 20, satisfies the timing diagram, yielding the next object-index match that assigns index 1 to the falling transition object on b, index 3 to the concurrent-edge object, index 5 to the rising transition object on c, and index 6 to the rising transition object on b. In this case, the error-index is undefined.

a	0	0	0	0	1	1	1	0	1
b	0	1	0	0	0	0	0	1	1
c	0	0	0	0	0	0	1	1	1
	0	1	2	3	4	5	6	7	8

The next object-index match in each of the following four words, however, results in a defined error-index. The word on the top left attempts to assign index 5 to the concurrent-edge object, but this index fails to match the signal pattern constraints of the time slice immediately preceding the concurrent-edge object; the error index in this case is 5, as determined by Condition 2c in the error-index definition (Defn. 66).

No index in the word on the top right satisfies the falling transition object on b; the error-index in this case is 9, as determined by Condition 2a in the error-index definition (Defn. 66). The word on the bottom left yields the same error-index, as no index satisfies the rising transition object on c.

The word on the bottom right yields a next object-index match with index 1 assigned to the falling transition object on b. Index 2 is located as the smallest index satisfying the rising transition object on c, but this violates the time-bound object between these two transition objects. The error index in this case is 2, as determined by Condition 2b in the error-index definition (Defn. 66).

a	0	0	0	0	0	0	1	1	1
b	1	1	0	0	1	0	0	1	1
c	0	0	0	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

a	0	0	0	0	1	1	1	1	1
b	0	0	0	0	1	1	1	1	1
c	0	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

a	0	0	0	1	1	1	1	1	1
b	1	1	0	0	0	0	1	1	1
c	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8

a	0	0	0	0	0	0	1	1	1
b	1	1	0	0	0	0	0	1	1
c	0	0	0	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

Sufficient framework now exists to define the two semantic relationships. The invariant semantics considers each index of a word in turn as the starting point for a next object-index search. Given index i, if the next object-index match for the objects in P_1 starting from i is defined for all objects, the next object-index match for all the objects in the diagram starting from i must be defined for all objects. If there exists an index for which this does not hold, the word does not invariantly satisfy the timing diagram. The error index is not required for the invariant semantics since matches are sought from every index into the word.

Definition 67 Let T be an anchor-free timing diagram with edge-explicit extension T' and let L be a language. Let P_1 and P_2 be a valid assume-guarantee partitioning for T' such that P_1 contains (explicitly or implicitly) at least one level object. T *invariantly describes* L, denoted $L \models_{\text{Inv}} T$, iff for every word $W \in L$,

1. If T' contains at least one successor edge object, then for all indicies i into W, if the next object-index match for T, W, P_1 and i is defined for all objects in P_1, then it can be extended to the next $P_1 \cup P_2$-object-index assignment I' and I' satisfies the constraints of T in W.
2. If T' contains no successor edge object,
   (a) For any waveform w consisting of only a level object b, all indices i of W satisfy b.
   (b) For any waveform w containing a transition object b, there are an infinite number of indices i of W that satisfy b.
   (c) For any concurrent edge object relating transition objects b_1 and b_2, index i satisfies b_1 iff it also satisfies b_2.

Given T, the set of all words that invariantly describe T is denoted $\mathcal{L}(T)^{\text{Inv}}$. In other words, $L \models_{\text{Inv}} T$ iff $L \subseteq \mathcal{L}(T)^{\text{Inv}}$.

The iterative semantics starts at an index into a word and locates the next object-index match for the objects in P_1. If it is undefined for some object in P_1, a new search is begun starting from the error index. If it is defined for all objects in P_1, the next object-index match for all of the objects in the timing diagram is sought from the same starting point. If it is undefined for some object, then that object must be in P_2, so the word does not satisfy the timing diagram. If it is defined for all object, a new search is started from the index immediately following the largest one assigned to any object, thus producing the concatentation effect. This process starts from the first index into the word and continues until either the word is determined not to satisfy the timing diagram or the end of the word is reached.

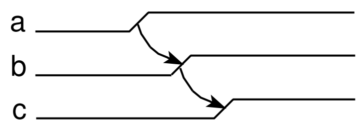
Definition 68 Let T be an anchor-free timing diagram, L be a language, W be a word, i be an index into W, and P_1 and P_2 be a valid assume-guarantee partitioning of T.

1. Let I be the next object-index match for T, W, i and P_1; I is called a *tested assignment* for T and W. T *iteratively describes* W *from* i, written $W, i \models T$, iff one of these cases holds:
   (a) i is greater than the length of W.
   (b) I being defined for all objects in P_1 implies that I can be extended to the next object-index match I' for T, W, and i and T iteratively describes W from $i' + 1$, where i' is the largest index in the range of I'.
   (c) I being undefined for some object in P_1 implies that T iteratively describes W from i' where i' is the error-index of I.
2. T *iteratively describes* L, written $L \models_{\text{Iter}} T$, iff for all words $W \in L$, $W, 0 \models T$.

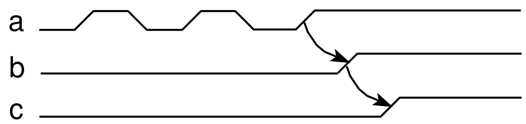
Given T, the set of all words that iteratively describe T is denoted $\mathcal{L}(T)^{\text{Iter}}$. In other words, $L \models_{\text{Iter}} T$ iff $L \subseteq \mathcal{L}(T)^{\text{Iter}}$.

When the iterative semantics are used with non-strict interpretations, there may exist indices matching the assume portion of a diagram that are not tested, as demonstrated in Example 25.

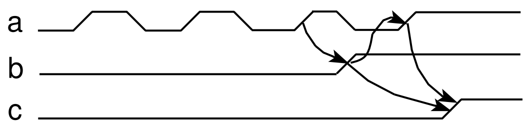
Example 25 Suppose this diagram is interpreted using non-strict semantics and that P_1 contains the rising transition object on a, but not those on b or c:



As defined, the iterative semantics would accept a word containing another rising transition on a before the rising transition on c; however, those rises occurring before the rise on c would not be tested. Any subsequent rising transition on a that preceded the rising transition on b would implicitly meet the property



but a rising transition on a that fell between those on b and c might not, if no subsequent rising transition occurred on b.



This point is raised as a note of caution when using the iterative semantics with non-strict interpretations of level objects.

Finally, semantics must be assigned to timing diagrams containing time anchors. Intuitively, time anchors focus timing diagrams on particular portions of words. However, only certain indices of words are ever tested against a timing diagram, based on the series of next object-index matches examined by the semantic relationships. Time anchor semantics therefore relate timing diagrams to particular next object-index matches. First, the set of next object-index matches checked for a particular timing diagram and word must be isolated.

Definition 69 Let T be a timing diagram, let W be a word, and let P_1 and P_2 be a valid assume-guarantee partitioning for T. The *tested object-index matches* for T, W, and P_1 is the set of all P_1-object-index assignments I such that I is a tested assignment for T and W if T is interpreted iteratively and I is the next object-index match for T, W, P_1, and some index i into W if T is interpreted invariantly.

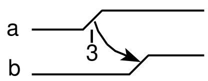
Definition 70 Let T be a timing diagram containing time anchors. Object-index assignment I is *anchor consistent* with T relative to a function ϕ from temporal variable in T to natural numbers iff for each time anchor (b, e) containing no free variables, $I(b) = \mathsf{TEval}(e)[\phi]$.

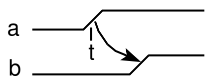
Definition 71 Let T be a timing diagram containing time anchors, let W be a word, and let P_1 and P_2 be a valid assume-guarantee partitioning for T.

1. Let ϕ be a function assigning natural numbers to all free variables appearing in time anchor expressions in T. T *anchor-describes* W *relative to* ϕ iff some tested object-index match, I, for T, W, and P_1 can be extended to an object-index match for P_1 and P_2 and I is anchor-consistent with T relative to ϕ.

2. Let L be a language. T *anchor-describes* L iff for all words W in L and all assignments ϕ to the free variables in T, T describes W relative to ϕ.

Example 26 Treated invariantly, the timing diagram on the left anchor-describes the word on the left (regardless of assignment ϕ since there are no free variables) because the next object-index match from index 2 finds a rising transition object on a at index 3. Treated iteratively, the diagram on the right anchor-describes the word on the right if ϕ assigns value 1 to t. If ϕ assigns value 3 to t, the diagram does not anchor-describe the word because there is no tested object-index match that assigns index 3 to the rising transition object on a.





a	0	1	0	0	1	1	1	1	1
b	0	0	1	0	0	1	1	1	1
	0	1	2	3	4	5	6	7	8

a	0	0	1	0	1	0	0	0	0
b	0	0	0	0	0	1	1	1	1
	0	1	2	3	4	5	6	7	8

The Logics

HHL supports several timing diagram logics based on three parameters: strict or non-strict interpretations, iterative or invariant semantics, and the complexity of time-bound expressions allowed in the syntax. Within a particular logic, subsets may be defined corresponding to different partitioning relationships. Logics are assumed defined only over well-formed timing diagrams. Two base logics are defined relative to semantic style:

- $\mathrm{TD}^{\mathrm{TI}}$: Interprets under $\models_{\mathrm{Inv}}$
- TD^{*}: Interprets under $\models_{\mathrm{Iter}}$

Each base logic can be customized to interpretations and time-bound complexity. The following notations specify customizations for time-bound complexity for base logic G. Notations relative to interpretation style are not given since those distinctions are not important for the results presented in Chapter 3.

- $G[\emptyset]$ — no time-bound expressions.
- $G[\mathcal{C} - \infty]$ — simple time-bound expressions using only temporal constants (no ∞).
- $G[\mathcal{C}]$ — simple time-bound expressions using only temporal constants and ∞.
- $G[\mathcal{N}]$ — all simple time-bound expressions.
- $G[\mathcal{N}, +]$ — complex time-bound expressions.

Unless otherwise specified, G denotes $G[\mathcal{N}, +]$.

2.2.4 CTL

Sentential temporal logic, a form of modal logic, is a popular formalism for hardware verification. Some temporal logics are *interval-based*, meaning that their formulas reflect what happens between states of a computation [41]. Logics reflecting what happens at states, rather than between them, fall into two general classes: linear-time temporal logics and branching-time temporal logics. Linear-time temporal logics view time as having a single future, or execution path [59]. Branching-time temporal logics view time as having many possible futures [29]. These views of futures are reflected in the formulas of the respective logics. Good surveys of temporal logic appear in [28] and [67].

HHL includes the branching-time logic CTL as one sentential representation. Timing diagrams are a form of linear temporal logic, as they depict only one possible future. Supporting a branching time logic gives HHL additional flexibility. Within the branching-time logic family, CTL is perhaps the most commonly used, both due to tools available for using it and its theoretical properties [23, 61].

Syntax

This definition is taken from [38].

Definition 72 The *formulas of CTL* are the state formulas $\langle \text{state\_f} \rangle$ as generated by the following grammars. An atomic proposition is any element of $\mathsf{SigNames}$.

$$\begin{array}{rcl}
\langle \text{state_f} \rangle & := & \langle \text{atomic proposition} \rangle \;\mid \\
 & & \neg \langle \text{state_f} \rangle \;\mid \\
 & & \langle \text{state_f} \rangle \wedge \langle \text{state_f} \rangle \;\mid \\
 & & \mathsf{A}(\langle \text{path_f} \rangle) \;\mid \\
 & & \mathsf{E}(\langle \text{path_f} \rangle)
\end{array}$$

$$
\begin{aligned}
\langle \text{path\_f} \rangle \quad := \quad & F\langle \text{state\_f} \rangle \quad | \\
& \mathsf{G}\langle \text{state\_f} \rangle \quad | \\
& \mathsf{X}\langle \text{state\_f} \rangle \quad | \\
& \langle \text{state\_f} \rangle \ \mathsf{U} \ \langle \text{state\_f} \rangle
\end{aligned}
$$

Semantics

The semantics of CTL is defined relative to a particular type of finite Kripke structure. Kripke structures consist of a set of states, a transition relation on the states, and a function from states to sets of signals indicating those signals that are assigned true in each state. It is standard to assume that each Kripke structure is defined relative to a set of atomic propositions; SigNames will be used for this purpose. This semantic definition is also taken from [38].

Definition 73 A *Kripke structure* is a tuple $\langle S, R, L\rangle$ where S is a set of states, $R : S \to S$ is a transition relation such that for all $s \in S$, there exists some $(s, s') \in R$, and $L : S \to 2^N$ indicates those propositions that are true in each state of S.

Definition 74 A *path* in a Kripke structure is a sequence $s = s_0, s_1, \ldots$ of states such that, for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. A path may be either finite or infinite, and is indexed by the natural numbers. Given path s, s^i refers to the suffix of s starting at index i, while s_i refers to the single state at index i.

Definition 75 Let $K = \langle S, R, L\rangle$ be a Kripke structure and let s_0 be a state in K. Let $n \in \mathsf{SigNames}$ and let f_1 and f_2 be formulas of CTL.

- $(K, s_0) \models n$ iff $n \in L(s_0)$
- $(K, s_0) \models \neg f_1$ iff $(K, s_0) \not\models f_1$

- $(K, s_0) \models f_1 \wedge f_2$ iff $(K, s_0) \models f_1$ and $(K, s_0) \models f_2$
- $(K, s_0) \models \mathsf{AX} f_1$ iff for all states s' such that $(s_0, s') \in R$, $(K, s') \models f_1$
- $(K, s_0) \models \mathsf{EX} f_1$ iff for some state s' such that $(s_0, s') \in R$, $(K, s') \models f_1$
- $(K, s_0) \models \mathsf{A}(f_1 \;\mathsf{U}\; f_2)$ iff for all paths $(s_0, s_1, s_2, \ldots)$, $\exists k \geq 0$ such that $(K, s_k) \models f_2$ and $\forall i, 0 \leq i < k$, $(K, s_i) \models f_1$
- $(K, s_0) \models \mathsf{E}(f_1 \;\mathsf{U}\; f_2)$ iff for some path $(s_0, s_1, s_2, \ldots)$, $\exists k \geq 0$ such that $(K, s_k) \models f_2$ and $\forall i, 0 \leq i < k$, $(K, s_i) \models f_1$

The remaining operators, $\mathsf{AF} f_1$, $\mathsf{EF} f_1$, $\mathsf{AG} f_1$, and $\mathsf{EG} f_1$ are abbreviations for $\mathsf{A}(\text{True} \;\mathsf{U}\; f_1)$, $\mathsf{E}(\text{True} \;\mathsf{U}\; f_1)$, $\neg\mathsf{EF}(\neg f_1)$, and $\neg\mathsf{AF}(\neg f_1)$, respectively.

Kripke structures can be viewed as abstractions over the device model. The device model captures the conditions under which transitions occur; Kripke structures place no conditions on state transitions. It is relatively straightforward to extract a Kripke structure from a device. Given a device D, the possible states of D are denoted by D_S and the set of possible input assignments for D as D_A.

Definition 76 The three components of a Kripke structure corresponding to a device D are defined as:

1. $S = D_S \times D_A$.
2. For all states $(s, a) \in S$, let $\langle D, i\rangle$ be a dynamic device in state s and let s' be the state of the dynamic device that follows from $\langle D, i\rangle$ given a. Then for all input assignments a', $((s, a), (s', a')) \in R$.
3. $L((s, a))$ is the set of all ports $p \in D$ such that the value on $i'(p) = 1$, where i' is the derived assignment for a dynamic device $\langle D, i\rangle$ in state s, given a.

Kripke structure $\langle S, R, L\rangle$ will be called the *derived Kripke structure for* D, and will be denoted K_D.

Lemma 6 *Given D, Kripke structure K_D is unique.*

Proof Let D be a device and consider $K_D = \langle S, R, L\rangle$. S is unique by construction. L is unique since the derived assignment for a dynamic device is unique by Lemma 1. D is deterministic by definition, so the dynamic device following from a dynamic device and an input assignment must be unique; R must therefore also be unique. Each component in unique, so K_D must be unique.

$\square$

Definition 77 Let D be a device and let F be a formula of CTL. Let ρ be a mapping from $\mathsf{SigNames}$ in F the ports of D. F *describes* dynamic device $\langle D, i\rangle$, denoted $\langle D, i\rangle \models F$, iff $(K_D, (s, i)) \models F_\rho$ where s is the state of $\langle D, i\rangle$ and F_ρ replaces each $\mathsf{SigName}$ in F with its corresponding port name under ρ. F *describes* D, denoted $D \models F$, iff for all assignments i, $\langle D, i\rangle \models F$.

Chapters 4 and 3 relate timing diagrams to CTL. There are close correlations between Kripke structures and languages. Given device D, each state of K_D generates a particular combination of values on the ports of D. Each combination of values is a letter in the language of D. A path through K_D therefore generates a word in D. Similarly, given a word in $\mathcal{L}(D)$, there must be at least one path through K_D that generates it, since words are generated by runs of D and transitions in K_D are defined by unit-segments of runs.

2.2.5 Second-Order Logic

HHL includes a monadic second-order logic SOL so that assertions involving quantification over signal names can be established. Signals are viewed as functions from discrete time expressions to values $\{0,1\}$; signal names in $\mathsf{SigNames}$ are therefore viewed as denoting such functions. SOL includes the standard connectives of predicate logic.

Syntax

Definition 78

1. The *atomic formulas* of SOL contain all expressions of the form $t_1 < t_2$ and $t_1 = t_2$ for discrete time expressions t_1 and t_2, as well as all expressions of the form $N(t)$ where $N \in \mathsf{SigNames}$ and t is a discrete time expression.

2. The *formulas* of SOL is the smallest set containing the atomic formulas and closed under the following:

   - If B_1 and B_2 are formulas, $\neg B_1$ and $B_1 \wedge B_2$ are formulas.
   - If B is a formula and N is a signal variable that is free in B, $\forall N B$ is a formula.
   - If B is a formula and t is a temporal variable that is free in B, $\forall t B$ is a formula.

The *well-formed formulas* are all formulas that contain no free temporal or signal variables.

Semantics

The semantics of SOL are defined with respect to runs of devices.

Definition 79 Let $\langle D_{Y_0}, \ldots \rangle$ be a run of device D, let F_1 and F_2 be formulas in SOL, let ρ map elements of $\mathsf{SigNames}$ in F_1 and F_2 to ports in D, and let ϕ map free temporal variables in F_1 and F_2 to natural numbers. Let $N \in \mathsf{SigNames}$, let X be a signal variable, let t_1 and t_2 be discrete time expressions, and let t be a temporal variable.

$$\langle D_{Y_0}, \ldots \rangle \models N(t_1)[\rho, \phi] \text{ iff } \rho(N) \text{ has value 1 in } D_{Y_e} \text{ where } e \text{ is } \mathsf{TEval}(t_1)[\phi]$$

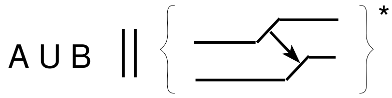


Figure 2.5: Example expression in the calculus.

$\langle D_{Y_0}, \ldots \rangle \models (t_1 = t_2)[\rho, \phi]$ iff $\mathsf{TEval}(t_1)[\phi] = \mathsf{TEval}(t_2)[\phi]$

$\langle D_{Y_0}, \ldots \rangle \models (t_1 < t_2)[\rho, \phi]$ iff $\mathsf{TEval}(t_1)[\phi] < \mathsf{TEval}(t_2)[\phi]$

$\langle D_{Y_0}, \ldots \rangle \models \neg F_1[\rho, \phi]$ iff $\langle D_{Y_0}, \ldots \rangle \not\models F_1[\rho, \phi]$

$\langle D_{Y_0}, \ldots \rangle \models (F_1 \wedge F_2)[\rho, \phi]$ iff $\langle D_{Y_0}, \ldots \rangle \models F_1[\rho, \phi]$ and $\langle D_{Y_0}, \ldots \rangle \models F_2[\rho, \phi]$

$\langle D_{Y_0}, \ldots \rangle \models (\forall X F_1)[\rho, \phi]$ iff $\langle D_{Y_0}, \ldots \rangle \models F_1[\rho', \phi]$ for each function ρ' from elements of $\mathsf{SigNames}$ to ports of D that agrees with ρ on all but the value for X

$\langle D_{Y_0}, \ldots \rangle \models (\forall t F_1)[\rho, \phi]$ iff $\langle D_{Y_0}, \ldots \rangle \models F_1[\rho, \phi']$ for each for each function ϕ' from temporal variables to natural numbers that agrees with ϕ on all but the value for t

Let D be a device and let F be a SOL formula. Let ρ map elements of $\mathsf{SigNames}$ in F_1 and F_2 to ports in D and let ϕ map free temporal variables in F_1 and F_2 to natural numbers. *F describes D relative to ρ and ϕ*, written $D \models F[\rho, \phi]$, iff $\langle D_{Y_0}, \ldots \rangle \models F[\rho, \phi]$ for all runs $\langle D_{Y_0}, \ldots \rangle$ of D.

2.2.6 Calculus of Temporal Representations

CTR was introduced in Chapter 1. An example CTR expression appears in Figure 2.5. CTR is similar to the Extended Temporal Logics of [85]. These logics allow automata as operators inside temporal logic, and show that the expressiveness of these logics is equivalent to the ω-regular languages. As will be discussed in Chapter 3, CTR differs from Extended Temporal Logic with respect to expressibility since HHL timing diagrams are non-regular.

Syntax

The syntax of CTR includes both the timing diagrams defined above and a subset of linear temporal logic formulae. Formally, the syntax of CTR is defined as follows:

Definition 80

1. The *sentential elements*, denoted SE, contains $\mathsf{SigNames}$ and is closed under $\neg$ and $\wedge$.
2. The *terms* of CTR contain SE and the well-formed timing diagrams. If E_1 and E_2 are terms of CTR, then so are $E_1; E_2$, ${E_1}^+$, ${E_1}^\omega$, $E_1 \parallel E_2$, $E_1 + E_2$ and $\bigcirc(E_1)$. If E_1 is not superscripted by $+$ or ω, then $E_1 \mathsf{U} E_2$ is also a term.

Intuitively, the CTR operators denote concatenation (;), parallel composition ($\|$), nondeterministic choice ($+$), next-time ($\bigcirc$), at least one iteration (superscript $+$), and infinite iteration (superscript ω).

Semantics

The semantics of CTR is given with respect to the language model. These semantics will be quite similar to the semantics given for timing diagrams, augmented as necessary to accommodate the operators of CTR. Timing diagrams are interpreted in CTR relative to the semantics of either $\mathrm{TD}^{\mathrm{TI}}$or TD^*; the semantic relationship to use would be specified by the user.

Definition 81 Let W be a word, E be a CTR formula, i be an index into W and k be either an index into W or ∞ such that $i \leq k$. Assume n denotes an element of $\mathsf{SigNames}$, each l_i denotes an element of *SE*, *TD* denotes a timing diagram, each T_i is a term of CTR, and T' is a non-timing diagram term of CTR (it may be complex terms containing timing diagrams, but it may not consist solely of a single timing diagram).

E describes W in the range $i \ldots k$, written $W_{i \ldots k} \models E$, under the following conditions.

$W_{i \ldots k} \models n$ iff $W_i(n) = 1$

$W_{i \ldots k} \models \neg l$ iff $W_{i \ldots k} \not\models l$

$W_{i \ldots k} \models l_1 \wedge l_2$ iff $W_{i \ldots k} \models l_1$ and $W_{i \ldots k} \models l_2$

$W_{i \ldots k} \models \mathrm{TD}$ iff $W_i \ldots W_k \models \mathrm{TD}$.

$W_{i \ldots k} \models \mathrm{TD}^+$ iff there exists some $i \leq j \leq k$ such that $W_{i \ldots j} \models T'^{\omega}$ non-vacuously

$W_{i \ldots k} \models \mathrm{TD}^{\omega}$ iff $W_i \ldots W_k \models \mathrm{TD}$.

$W_{i \ldots k} \models T_1 ; T_2$ iff $\exists j \geq i$ such that $W_{i \ldots j} \models T_1$, $W_{i \ldots j-1} \not\models T_1$, and $W_{j+1 \ldots k} \models T_2$

$W_{i \ldots k} \models T_1 + T_2$ iff $W_{i \ldots k} \models T_1$ or $W_{i \ldots k} \models T_2$

$W_{i \ldots k} \models T_1 \parallel T_2$ iff $W_{i \ldots k} \models T_1$ and $W_{i \ldots k} \models T_2$

$W_{i \ldots k} \models T_1 \mathsf{U} T_2$ iff $\exists j \geq i$ such that $W_{i \ldots j} \models T_1^{\omega}$ and $W_{j+1 \ldots k} \models T_2$

$W_{i \ldots k} \models \bigcirc(T_1)$ iff $W_{i+1 \ldots k} \models T_1$

$W_{i \ldots k} \models l_1^{\omega}$ iff $\forall i \leq j \leq k, W_{j \ldots j} \models l_1$

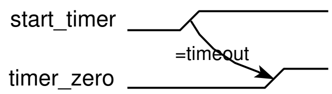
$W_{i \ldots k} \models T'^{\omega}$ iff $i \ldots k$ can be split into j consecutive ranges such that each $i_h \ldots k_h \models T'$.

$W_{i \ldots k} \models T'^{+}$ if there exists some $i \leq j \leq k$ such that $W_{i \ldots j} \models T'^{\omega}$

CTR formula E *describes* language L iff for all words $W \in L$, $W, 0 \ldots \infty \models E$. Given device D, E *describes* D, written $D \models E$ iff $\mathcal{L}(D) \models E$.

2.2.7 Component Abstraction

Abstractions hide details about designs. They specify components whose properties are important, but whose implementations are irrelevant. For example, the design example presented in Chapter 5 includes a timer component. For purposes of the proofs in that chapter, all that is necessary is that the timer module used exhibit the following behavior:



where timeout represents a constant amount of discrete time. The abstract component treats timeout as a parameter. Support for parameterized time constraints is an important issue in verification because concrete time constraints are often too specific for general-purpose verification [3].

Definition 82 An *abstract component* is a tuple $\langle I, O, R, \mathsf{Labels}\rangle$ where I and O are sets of objects called the *input points* and *output points*, Labels assigns names to each element of $I \cup O$, and R is a set of circuit diagrams, ASM charts, timing diagrams, CTL formulas, SOL formulas, and CTR terms.

Abstract components can be used within circuit sketches. This requires icons for depicting the abstract components. The function Icons is therefore extended with an icon mapping to each new abstract component. It is assumed that each icon contains the correct number of depicted input and output ports for the corresponding component.

In order for a set of formulas to be useful in an abstract component, they must relate to the input and output places specifying the interface of the component. The well-formedness assumption for abstract components requires that the formulas all depend upon these interface signals.

Definition 83 An abstract component $\langle I, O, R, \mathsf{Labels}\rangle$ is *well-formed* iff all elements r of R satisfy the following criteria:

1. If r is a timing diagram, each waveform must be labeled by a name assigned in Labels.
2. If r is a circuit diagram, each input interface and output interface port must be labeled by a name assigned in Labels.
3. If r is an ASM chart, the set of external signals must be a subset of the names assigned in Labels.

4. If r is a CTL formula, the propositions used in a must be a subset of the names assigned in Labels.

5. If r is an SOL formula, the signals used in a must be a subset of the names assigned in Labels.

6. If r is a CTR term, the signals used to label timing diagrams and as atomic propositions must be a subset of the names assigned in Labels.

Abstract components can be used either as stand-alone objects or they can be encorporated into circuit diagrams; the first use can be interpreted as a form of the second. As a result, the definition of circuit diagram syntax and semantics must be extended to allow for the use of abstract components. The following definitions formalize the necessary extensions:

Definition 84 An *extended circuit sketch* is a tuple $\langle P, I, O, W, K, \mathsf{Labels}\rangle$ in which K is a (possibly empty) set of extended circuit sketches and abstract components, each of whose I and O components are subsets of P. All other components of the tuple are as given in Definition 22.

In the case that K consists of only a single abstract component, and no circuit sketches, it is assumed that the same I and O are shared by both the abstract component and the enclosing circuit sketch. The same readability assumptions apply to circuit sketches with and without abstract components. In preparation for assigning semantics to circuit sketches containing abstract components, the notion of a primitive circuit sketch must also be extended:

Definition 85 Let $C = \langle P, I, O, W, K, \mathsf{Labels}\rangle$ be an extended circuit sketch. C is called *primitive* iff every element of K is either a basic circuit sketch or an abstract component.

The semantics of extended circuit sketches are also given relative to devices via depiction maps. However, with abstract components there is no notion of structural implementation because the internal structure need not be specified.

Definition 86 Let A be an abstract component and let D be a static device. *A abstractly describes D* iff for each element $r \in R$

1. If r is a circuit sketch or ASM chart, $D \models_b r$
2. If r is an SOL or CTL formula, $D \models r$
3. If r is a timing diagram or CTR formula, $\mathcal{L}(D) \models r$

Definition 87 Let $C = \langle P, I, O, W, K, \mathsf{Labels} \rangle$ be a primitive extended circuit sketch and let D be a static device.

1. *C describes D*, written $D \models C$ iff there is a depiction map σ from C to D such that for each abstract component $k = \langle I_k, O_k, R_k, \mathsf{Labels}_k \rangle$ in K, the restriction of σ to $I_k \cup O_k$ encapsulates a subdevice D' of D such that D' abstractly describes k.
2. D is a *behavioral implementation* of C iff $D \models C$ and every port of D is either in the range of C or in some subdevice encapsulated by some abstract component of C.

2.3 HHL

Definition 88 The *terms* of HHL, also called *formulas*, contain exactly

- The extended circuit diagrams,
- The ASM charts,
- The terms of CTL,
- The terms of SOL,
- The well-formed timing diagrams, and

- The terms of CTR.

The modeling relationships on HHL terms are as defined for each individual representation. The individual semantic relationships interact through semantic consequence. Semantic consequence indicates when one formula may be derived from a set of formulas. Intuitively, formula f is a semantic consequence of set of formulas F iff any model of all formulas in F is also a model of f. It is clear from Figure 2.2 that semantic consequence in HHL must be defined relative to the language model. The semantics of circuit diagrams, ASM charts, SOL, and CTL must therefore be extended to the language model.

Definition 89 Let f be a circuit diagram, ASM chart, SOL formula, or CTL formula and let L be a language. $L \models_t f$ iff L is the language of some device D such that $D \models_t f$, where $\models_t$ is one of the modeling relationships defined between D and f (such as $\models_b$ or $\models_s$).

Circuit diagrams and ASM charts have multiple semantic relationships, some structural and some behavioral. The semantic consequence definition chooses which relationship to use based on context. Circuit diagrams, for example, are usually compared to one another at a behavioral, rather tha structural, level. As a result, when a circuit diagram is a semantic consequence of another circuit diagram (or a set containing another circuit diagram), behavioral modeling is used. When a non-circuit diagram is a consequence of a circuit diagram, structural modeling is used; behavioral relationships between circuit diagrams and non-circuit diagrams can be derived through the behavioral relationships within the individual representations. This decision simplifies many of the soundness proofs in Chapter 4. As a result of this distinction in modeling relationships, the definition of semantic consequence provided below has three cases: one when a circuit diagram is a consequence of a set containing a circuit diagram, one when an ASM chart is a consequence of a set containing an ASM chart, and one for all other cases.

The definition of semantic consequence assumes a correlation between the elements of $\mathsf{SigNames}$ appearing in formulas and the ports of any device used in the definition. This mapping, ρ must be consistent with the signal maps, depiction maps, and similar mappings used to relate objects to ports in the semantic definitions for each representation. This restriction guarantees that when the same names are used in several formulas, they refer to the same ports or signals in the underlying models. When comparing a language defined on $\mathsf{SigNames}$ to one defined on ports, the languages are assumed compared with respect to ρ.

Definition 90 Let F be a finite set of formulas $F_1, \ldots, F_k$ and let f be a formula. Let ϕ be a mapping from free temporal variables in $F \cup \{f\}$ to natural numbers. f is a *semantic consequence* of F, written $F \models f$ iff for every device D and every run R_D of D, whenever $\mathcal{W}(R_D)$ is anchor-described by every time-anchored timing diagram and every CTR formula in F, D is described by every anchor-free timing diagram and every CTL formula in F, and R_D is described by every grounded ASM chart and every SOL formula in F relative to ϕ and ρ, where ρ is a mapping from elements of $\mathsf{SigNames}$ in $F \cup \{f\}$ to ports of D then

1. If f is a circuit diagram and some F_i is a circuit diagram, then D is behaviorally described by f whenever D is behaviorally described by each F_i that is a circuit diagram, and D is functionally structurally described by every time-free ASM chart in F.

2. If f is a time-free ASM chart and some F_i is an ASM chart, then D is behaviorally described by f whenever D is a behavioral implementation of every time-free ASM chart in F, and D is a structural implementation of every circuit diagram in F.

3. Otherwise, f describes D whenever D is structurally described by every circuit diagram in F and D is functionally structurally described by every time-free ASM chart in F.

3

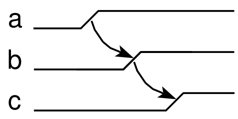
Theoretical Results

3.1 Representing Timing Diagrams in CTL

Timing diagrams have occasionally been given semantics via translation into sentential temporal logics [74] [75]. As discussed in Chapter 1, this method of assigning semantics is problematic because the correctness of the translation cannot be verified without an existing timing diagram semantics. This section presents a translation algorithm from temporally unambiguous diagrams in $\mathrm{TD}^{\mathrm{TI}}[\mathcal{C}-\infty]$ into CTL and verifies its semantic correctness. Timing diagrams in $\mathrm{TD}^{\mathrm{TI}}[\mathcal{N}]$ are not translated because, as discussed in Section 3.3, they can express non-regular properties whereas CTL can only express regular properties [38].

The algorithm generates expressions of the form "in all states, if the assume portion is satisfied, then the entire diagram is satisfied." The translation is straightforward for timing diagrams containing no time bound objects, as shown in Example 27.

Example 27 Let the assume portion of the diagram be the rising transition object on a. The first CTL expression is generated for non-strict semantics and the second for strict semantics:



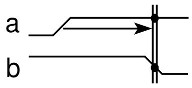
$$\mathsf{AG}(\text{rising}(a) \rightarrow \text{rising}(a) \wedge \mathsf{AX}(\mathsf{AF}(\text{rising}(b) \wedge \mathsf{AX}(\mathsf{AF}(\text{rising}(c))))))$$

$$\mathsf{AG}(\text{rising}(a) \rightarrow \text{rising}(a) \wedge \mathsf{AXA}[\neg\text{rising}(b)\ \mathsf{U}\ (\text{rising}(b) \wedge \mathsf{AXA}[\neg\text{rising}(c)\ \mathsf{U}\ \text{rising}(c)])])$$

where rising(w) expands to $(\neg w \wedge \mathsf{AX}(w))$. The strict semantics expression uses U instead of AF because the former cannot force the signal patterns to hold.

For non-strict diagrams containing concurrent edge objects, a mixture of AF and U operators is used, depending upon whether there are any patterns to constrain during a particular time slice.

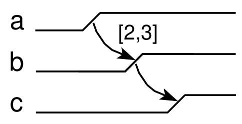
Example 28 Let the assume portion of the diagram be the rising transition object on a and suppose the diagram is interpreted non-strictly.



$$\mathsf{AG}(\text{rising}(a) \rightarrow \text{rising}(a) \wedge \mathsf{AXA}(a\ \mathsf{U}\ (\text{falling}(b) \wedge a)))$$

Time bound objects complicate matters because n either AF nor U can capture durational information. The algorithm must therefore use the next time operator AX to test bounds satisfaction; the possible time slice durations within the timing diagram are enumerated explicitly in the translated formula as shown in Example 29.

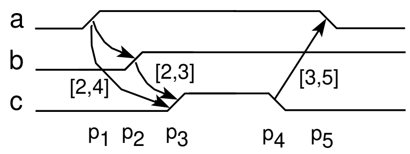
Example 29 Let the assume portion of the diagram be the rising transition on a and assume the diagram is interpreted under the non-strict semantics.



$$\begin{aligned}&\mathsf{AG}(\text{rising}(a) \rightarrow\\ &\quad(\text{rising}(a) \wedge \mathsf{AX}(\neg\text{rising}(b) \wedge \mathsf{AX}((\text{rising}(b) \wedge \mathsf{AXAF}(\text{rising}(c)))\vee\\ &\qquad\qquad\qquad\qquad(\neg\text{rising}(b) \wedge \mathsf{AX}(\text{rising}(b) \wedge \mathsf{AXAF}(\text{rising}(c))))))))\end{aligned}$$

The algorithm takes the combinations of durations satisfying the time-bound expressions in tabular form. Each table covers a range of time points related by time bound expressions. The table contains one column for each time point in the range except the first one. Each row indicates a valid combination of time slice durations, where the value in column p_i is the duration from p_{i-1} to p_i.

Example 30 The time bound object (p_1, p_3) initially generates the following table:



p_2	p_3
1	1
1	2
1	3
2	1
2	2
3	1

Each row with a 1 in the p_3 column violates the time-bound expression on (p_2, p_3). These rows must be removed, leaving two tables relevant to the diagram:

p_2	p_3
1	2
1	3
2	2

p_5
3
4
5

These tables do not support time bound expressions containing ∞. Handling ∞ requires a different type of table because the list of all allowable durations is not finite. Although the present algorithm does not handle diagrams containing ∞, it should be extendible to all $\mathrm{TD}^{\mathrm{TI}}[\mathcal{C}]$ diagrams.

The following table-generation algorithm initializes each table with the durations allowed by the time-bound expression spanning the most time points in the range. Rows of this table are then removed until all rows are consistent with all time-bound expressions over time points in the range. Each time point should appear in the header of at most one table. The lemma following the algorithm establishes that each valid combination of durations is reflected by some row in the table.

Algorithm 3 Let T be a temporally unambiguous timing diagram. Let P be a partition of the time points of T such that

1. Each cell is a contiguous range of time points,
2. There does not exist a time-bounded successor edge object in T that spans time points in more than one cell, and
3. No cell can be subdivided into smaller cells that meet the previous two requirements.

For each cell C of P, generate a table with columns labeled successively with p_{i+1} through p_j as follows, where p_i and p_j are the miminal and maximal time points in C, respectively:

1. Let n be the sum of all constant upper bounds in time-bound expressions over points within p_i through p_j inclusive. For each k such that $j - i < k < n$, add a row to the table for each permutation of each combination of $j - i$ non-zero numbers that sums to k.
2. For every successor edge object $(p_{i'}, p_{j'})$ such that $i \leq i'$ and $j' \leq j$ and containing time-bound expression $[l', u']$, remove any row from the table in which the sum of the entries under $p_{i'+1}$ through $p_{j'}$ is smaller than l' or larger than u'.

Lemma 7 *Let T be a timing diagram. Let W be a word and let I be the next object-index match from any index in W. Let b_j be the largest object for which I is defined. Consider those tables generated by Algorithm 3 covering objects that are all defined in I. There must exist a row from* *each table such that for all objects b_k up to and including b_j such that there is a constraint spanning time slice (p_{k-1}, p_k) and p_k is covered in a table, $I(b_k) - I(b_{k-1})$ is the value for p_k in those rows.*

Proof Since I is defined for all objects up to and including b_j, I must satisfy all time-bound expressions ending on objects up to b_j (Defn. 66). For each b_k up to and including b_j, $I(b_k) - I(b_{k-1})$ must therefore be a valid value within the restrictions of the time-bound expressions ending at or before b_j; in particular, the sum of all distances $I(b_k) - I(b_{k-1})$ must be no larger than the sum of all upper bounds appearing in time-bound expressions. Each table was initialized with all of the possible non-zero differences between the time slices that resulted in a sum no larger than the upper bound sum. Since I is defined for all objects up to and including b_j, the differences between time points in I must constitute an initial row of the table. The only rows removed were those that failed to satisfy a constraint. Since I satisfies all constraints, the row corresponding to I must appear in the final table.

$\square$

The translation algorithm is presented in pseudocode; a Scheme implementation of it appears in Appendix A. GenCTL is the main procedure. It takes a time point i to process, the maximum time point to process, and the tables of time bound information. The timing diagram is left implicit in the pseudocode. GenConstr generates CTL for the constraints, relative to the information in the tables. Its *count* parameter indicates which length duration is being processed for the time slice ending at time point i; the i, *max*, and tables parameters are the same as for GenCTL. GenObj(j) generates the obvious CTL formula for the object at time point j: w for a high-level object, $\neg w$ for a low-level object, $\neg w \wedge \mathsf{AX}(w)$ for a rising transition object, $w \wedge \mathsf{AX}(\neg w)$ for a falling transition object, and a conjunction of these for any concurrent-edge object. The signal patterns required of slice (p_j, p_{j+1}) are symbolized by SigPat_j.

The calls to GenCTL from GenConstr send *modified_tables* to the *tables* argument. Rather than generate one separate subformula for each possible combination of durations in the tables, the algorithm generates one subformula for each table value for the current time slice. Within the subformula for a particular value is the formula for the remainder of the diagram, relative to the portion of the table with that value in the first column. For example, if the following table exists when processing time point p_i

p_{i+1}	p_{i+2}	p_{i+3}
2	1	2
2	1	3
3	1	2

then when *count* is two, the boxed off portion of the table is passed as *modified_tables* on the call to GenCTL.

Algorithm 4

GenConstr(i, max, count, tables) =

  if an entry for count appears in the column labeled $i + 1$ in a table

    if count is the largest entry in the column labeled $i + 1$

      AX(GenCTL($i + 1$, max, modified_tables))

    else

      AX(GenCTL($i + 1$, max, modified_tables) $\vee$

        (SigPat$_i$ $\wedge$ $\neg$GenObj($i + 1$) $\wedge$ GenConstr(i, max, count + 1, tables)))

    endif

  else

    AX(SigPat$_i$ $\wedge$ $\neg$GenObj($i + 1$) $\wedge$ GenConstr(i, max, count + 1, tables))

  endif

end GenConstr

GenCTL(i, max, tables) =

if $i > \max$

  true

elseif $i = \max$

  $\mathrm{GenObj}(i) \wedge \mathrm{SigPat}_{i-1}$

elseif there exists a column for $i+1$ in some table

  $\mathrm{GenObj}(i) \wedge \mathrm{SigPat}_{i-1} \wedge \mathrm{GenConstr}(i, \max, 1, \text{tables})$

elseif SigPat_{i-1} is true

  $\mathrm{GenObj}(i) \wedge \mathsf{AX}\ \mathsf{AF}(\mathrm{GenCTL}(i+1, \max, \text{tables}))$

else

  $\mathrm{GenObj}(i) \wedge \mathrm{SigPat}_{i-1} \wedge \mathsf{AXA}[(\mathrm{SigPat}_i \wedge \neg\mathrm{GenObj}(i+1))\ \mathsf{U}$

  $\mathrm{GenCTL}(i+1, \max, \text{tables})]$

endif

end GenCTL

Definition 91 Given a temporally unambiguous timing diagram T, define CTL_T to be

$$\mathsf{AG}(GenCTL(1, split, tables) \rightarrow GenCTL(1, max, tables))$$

where *split* is the split point between P_1 and P_2, *max* is the index of the maximal time point in T, and *tables* are the tables generated for T by Algorithm 3.

Chapter 2 discussed the correlation between words and paths through Kripke structures. Given a word W and a path $s_0, \ldots, s_n$ that generates it, if W_i satisfies an object, then s_i must model that object. This correlation is used extensively in the following lemma and theorem, which establish the correctness of the translation algorithm.

Lemma 8 *Let D be a device, T be a timing diagram, W be a word, and I be the next object-index assignment for T, W, and i. Let $s_0, \ldots, s_n$ be a path in K_D corresponding to W. Let b_{j-1} be an object such that there exists an index larger than i_{j-1} that satisfies b_j, no intermediate index violates*

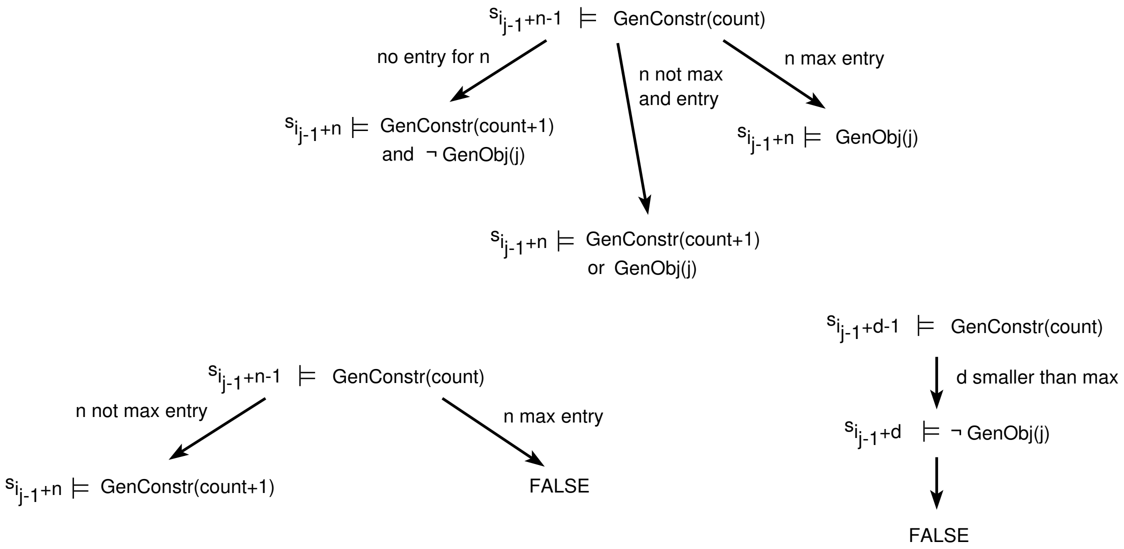


Figure 3.1: Summary of next-state requirements in CTL translation algorithm. An arrow from one statement to another indicates that the source question is reduced to the target question when the condition on the arrow is satisfied.

the signal patterns of (p_{j-1}, p_j)*, and the tables generated by Algorithm 3 contain a heading for* p_j*.* $(K_D, s_{i_{j-1}})$ *models* $\mathrm{GenConstr}(j-1, \max, 1, \mathrm{tables})$ *only if* $d = i_j - i_{j-1}$ *appears as an entry under the column for* p_j*.*

Proof Figure 3.1 depicts how "does $(K_D, s_{i_{j-1}+n-1})$ model $\mathrm{GenConstr}(j-1, \max, 1, \mathrm{tables})$" reduces to a question about $(K_D, s_{i_{j-1}+n})$. The top diagram provides a template for this question with certain information irrelevant to the proof omitted. By definition of d, all states larger than $s_{i_{j-1}}$ and smaller than $s_{i_{j-1}+d}$ model $\neg\mathrm{GenObj}(j)$ and SigPat_{j-1}. In addition, $s_{i_{j-1}+d}$ must model $\mathrm{GenObj}(j)$. The lower two diagrams update the template based on this information: the left diagram holds for any $n < d$, the right is based on there being no table entry for d. From the diagrams, any reduction of the question "does $(K_D, s_{i_{j-1}+n-1})$ model $\mathrm{GenConstr}(j-1, \max, 1, \mathrm{tables})$" reduces to false, so $(K_D, s_{i_{j-1}})$ cannot model $\mathrm{GenConstr}(j-1, \max, 1, \mathrm{tables})$.

$\square$

Theorem 4 *Let* D *be a device and let* T *be a temporally unambiguous* $\mathrm{TD}^{\mathrm{TI}}[\mathcal{C}-\infty]$ *timing diagram.*

$\mathcal{L}(D) \models T$ *iff* $D \models \mathrm{CTL}_T$.

Proof From the definitions of $\models T$ (Defn. 67), $\models \mathrm{CTL}_T$ (Defn. 77), and CTL_T, this is equivalent to proving $(K_D, s) \models \mathsf{AG}(\mathrm{GenCTL}(1, \mathrm{split}, \mathrm{tables}) \rightarrow \mathrm{GenCTL}(1, \max, \mathrm{tables}))$ for all s iff $W \models T$ for all W in $\mathcal{L}(D)$.

1. Assume $W \models T$ for all W in $\mathcal{L}(D)$. Let s be any state in K_D. Let $s_0, s_1, \ldots, s_n$ be any path in K_D such that s_0 is s. Let W be the word extracted from $s_0, \ldots, s_n$. By the definition of AG (Defn. 75), $(K_D, s_i) \models \mathrm{GenCTL}(1, \mathrm{split}, \mathrm{tables}) \rightarrow \mathrm{GenCTL}(1, \max, \mathrm{tables})$ must be established for all $0 \leq i \leq n$. Expanding the definition of $\mathrm{GenCTL}(1, \mathrm{split}, \mathrm{tables})$, any state for which $\mathrm{GenObj}(1)$ is false will satisfy this formula. Let s_i, $1 \leq i \leq n$, be any state such that $(K_D, s_i) \models \mathrm{GenObj}(1)$. By the definition of $\mathrm{GenObj}(1)$, W_i must therefore satisfy b_1. Let I be the next object-index match for T, W, and i; $I(b_1)$ must be i.

   (a) Assume I is defined for all objects in T. Let i_j denote $I(b_j)$ for all objects b_j in T. $(K_D, s_{i_{\max-1}})$ must model $\mathrm{GenCTL}(\max - 1, \max - 1, \mathrm{tables})$. This follows from the definition of $\mathrm{GenCTL}(\max-1, \max-1, \mathrm{tables})$ and the definitions of GenObj and SigPat. Let k be any index $1 < k < \max - 1$. Assume that for all j larger than k, if $I(b_j) = i$, then $(K_D, s_i) \models \mathrm{GenCTL}(j, \max - 1, \mathrm{tables})$.

   If there are no columns for $k + 1$ in the tables, $\mathrm{GenCTL}(k, \max - 1, \mathrm{tables})$ is either $\mathrm{GenObj}(k) \wedge \mathrm{SigPat}_{k-1} \wedge \mathsf{AXA}[(\mathrm{SigPat}_i \wedge \neg \mathrm{GenObj}(k+1)) \mathsf{U} \mathrm{GenCTL}(k+1, \max-1, \mathrm{tables})]$ or $\mathrm{GenObj}(i) \wedge \mathsf{AX}\ \mathsf{AF}(\mathrm{GenCTL}(i + 1,\ \max,\ \mathrm{tables}))$. By construction, there are no constraints in T that cross time slice (p_i, p_{i+1}). The value of $I(b_{k+1})$ is therefore based only on satisfying b_{k+1} and not violating the signal patterns of (p_i, p_{i+1}) between $I(b_k)$ and $I(b_{k+1})$. Let i be $I(b_k)$ and let i' be $I(b_{k+1})$. $(K_D, s_i) \models \mathrm{GenObj}(k) \wedge \mathrm{SigPat}_{k-1}$ since $I(b_k)$ is i, and by the definitions of GenObj and SigPat. By definition of I (Defn. 66), i' is the smallest index larger than i that satisfies b_{k+1}, and all indices from $i + 1$ through i' satisfy the signal patterns of time slice (p_k, p_{k+1}). Since $(K_D, s'_i) \models \mathrm{GenCTL}(k + 1, \max - 1, \mathrm{tables})$ by assumption, (K_D, s_i) must model one of the two possible values for $\mathrm{GenCTL}(k, \max - 1, \mathrm{tables})$, where $i = I(b_k)$.

Now assume there are constraints for k in the tables. The above argument established that (K_D, s_k) models each of GenObj(k) and SigPat$_{k-1}$. That (K_D, s_k) models GenConstr(i, max, 1, tables) must still be established. Let d be the difference between b_{k+1} and b_k. Since I satisfies the constraints of T, d must be a valid value for the distance between b_k and b_{k+1}. Hence, there must be a table with header p_{k+1} and value d for some of the rows. Unwinding the recursive definition of GenConstr, (K_D, s_k) can model it if (K_D, s_{k+1}) through (K_D, s_{k+d-1}) all model both SigPat$_k$ and $\neg$GenObj($k+1$) and (K_D, s_{k+d}) models GenCTL on $k+1$ with the updated tables. This last requirement is satisfied by the inductive hypothesis and the fact that $I(b_k) + d$ equals $I(b_{k+1})$ by definition. The requirements on each intermediate state are satisfied by the correlation of W and path $s_0, \ldots, s_n$ and the definition of $I(b_{k+1}$ relative to $I(b_k)$, so $(K_D, s_i) \models$ GenCTL($k, \max - 1$, tables), where $i = I(b_k)$.

(b) Assume I is not defined for all objects in T. Since W does describe T from i, by Definition 66 I is not defined for some object in P_1. Let b_j be the last object for which I is defined. Let i_k denote $I(b_k)$ for all $1 \leq k \leq j$. (K_D, s_{i_j}) cannot model GenCTL($j, \max - 1$, tables). By the correlation between W_{i_j} and s_{i_j}, (K_D, s_{i_j}) models GenObj(j) and SigPat$_j$.

If there are no constraints crossing time slice (p_j, p_{j+1}), then $I(b_{j+1})$ must be undefined either because there is no index into W larger than i_j that satisfies b_{j+1} or there is some index that violates the signal patterns of (p_j, p_{j+1}) before an index satisfying b_{j+1} is located. In the former case, there cannot exist a state s further down on the path from (K_D, s_{i_j}) such that (K_D, s) models GenObj($j+1$). Therefore (K_D, s_{i_j+1}) cannot model $\mathsf{A}[(\text{SigPat}_j \wedge \neg \text{GenObj}(j+1)) \mathsf{U} (\text{GenCTL}(j+1, \max - 1, \text{tables}))]$. Similarly, (K_D, s_{i_j+1}) cannot model this expression in the second case because a state exists that violates SigPat$_j$ before a state is found that satisfies GenObj($j+1$). In either case, (K_D, s_{i_j}) cannot model GenCTL($j, \max - 1$, tables).

If there is a constraint crossing time slice (p_j, p_{j+1}), then $I(b_{j+1})$ could be undefined for either of the two reasons given above or because a constraint failed to be satisfied. By unwinding the definition of GenCTL, every path that models the formula must satisfy

SigPat$_j$ until GenObj(j) is satisfied, so a similar argument shows that (K_D, s_{i_j}) cannot model GenCTL(j, max $-$ 1, tables). Assume the former held but a constraint failed that crossed (p_j, p_{j+1}). By construction, the tables capture all possible combinations of values between events that would satisfy the given constraints. Let d be the difference between the index found that satisfies b_{j+1} and i_j. Since d is not a valid value, there is no entry for d in the table under heading $j+1$. By Lemma 8, (K_D, s_{i_j}) cannot model GenConstr(j, max $-$ 1, 1, tables), so it cannot model GenCTL(j, max $-$ 1, tables).

By unwinding the definition of GenCTL, (K_D, s_i) cannot model GenCTL(1, split, tables). (K_D, s_i) therefore does model CTL$_T$.

2. Assume $W \not\models T$ for some W in $\mathcal{L}(D)$. Then there exists some index i into W such that the next object-index match I for P_1, W, and i cannot be extended to the next object-index match I' for T, W, and i. Let W' be the suffix of W starting with position $I(b_1)$. Let $s_0, \ldots, s_n$ be a path in K_D that is consistent with W'. Since I' is defined for all objects in P_1, an argument similar to that of Part 1a establishes that (K_D, s_0) models GenCTL(1, split, tables). Since I' is not defined for all objects in T, an argument similar to that of Part 1b establishes that (K_D, s_0) does not model GenCTL(1, max $-$ 1, tables). (K_D, s_0) therefore cannot model CTL$_T$.

□

Although timing diagrams appear more compact than their respective CTL translations, the difference in formula size is not theoretically significant. The size of a CTL formula is the count of its operands and operators [28]. The size of a timing diagram is the count of its transition objects, concurrent edge objects, successor edge objects, and time-bound expressions. These are captured using two parameters: the number of waveforms (n) and the number of time points (p). Relative to these parameters, the worst case number of each type of object is $O(n)O(p)$, $O(p)$, $O(p^2)$, and $O(p^2)$, respectively. In practice, the $O(p^2)$ estimates for the number of successor edge objects and time-bound expressions are expected to be closer to linear, as the worst case figures are only reached if time-bounds are placed between every pair of objects. The remainder of this comparison therefore treats $O(n)O(p)$ as the size of a timing diagram.

Consider a timing diagram with no time-bound expressions. GenConstr will never be invoked, so the formula analysis relies only on GenCTL. Both GenObj and SigPat are $O(n)$, since the object may be a concurrent edge object on all the waveforms and there may be signal pattern constraints on all waveforms. GenCTL(max, max, tables) is therefore $O(n)$. When there are no constraints, GenCTL(i, max, tables) is $O(n)$ plus GenCTL($i+1$, max, tables). Since GenCTL is invoked for each time point other than the minimal and maximal points, GenCTL(1, max, tables) is $O(p)O(n)$. The relative sizes between timing diagrams and their CTL formulas are therefore essentially the same in this case.

Consider a timing diagram with time-bound expressions. Let $u_1, \ldots, u_p$ be the largest values appearing in the table columns for each time point. GenCTL(max, max, tables) is still $O(n)$. GenConstr(i, max-1, u_j, tables) is AX(GenCTL($i+1$, max, tables)), so it is also $O(n)$. GenConstr(i, max-1, k, tables), where $k < u_j$, is $O(n)$ plus GenConstr(i, max-1, $k+1$, tables), so GenConstr(i, max-1, 1, tables) is $u_jO(n)$. GenCTL(max-1, max, tables) has order the worst of GenConstr(max-1, max, 1, tables) and GenCTL(i+1, max tables). GenConstr is possibly worse, due to the u_j factor, so GenCTL(max-1, max, tables) is also $u_jO(n)$. Iterating the analysis, the recurrence for GenCTL(1, max, tables) is $(\Sigma_{i=1}^{p} u_p)O(n)$. Letting u_m be the largest of $u_1, \ldots, u_p$, GenCTL(1, max, tables) is $u_mO(p)O(n)$. Thus the formula size blowup depends on the upper bounds appearing in the time-bound expressions; while the blowup is only linear in the upper bound magnitude, in practice this difference could render the translated formulas cumbersome to read.

The subset of CTL representable in timing diagrams is still undetermined. The subset is expected to lie in $\forall$CTL, the subset of CTL that allows the A path quantifier, but not the E path quantifier [37]. This makes intuitive sense since timing diagrams express linear, rather than branching, properties. However, even within $\forall$CTL, there are formulas that cannot be represented in timing diagrams, such as AF rising(a) $\rightarrow$ rising(b). There is no way to relate rising transitions on a and b such that b can occur either simultaneously with a or following a, as is required by the semantics of AF, using only one timing diagram. This is expressible in HHL using a CTR formula disjoining the diagram where the rising transitions are connected with a concurrent edge object and the diagram with a successor edge object from the rising transition on a to the one on b. The single timing diagram expressibility question therefore remains open for CTL.

3.2 Semantic Properties of Timing Diagrams

Chapter 2, Section 2.2.3 showed that a timing diagram with multiple tail objects could be divided into a set of single-tail diagrams. This section establishes two relationships between the original diagram and the single-tail diagrams: whether the intersection of languages modeling the single-tail diagrams model the original diagram, and whether any language modeling the original diagram must model each single-tail diagram. The two properties are stated formally as follows:

Definition 92 Let G be a timing diagram logic with semantic modeling relationship $\models_G$. G has the *single-tail intersection property* iff for all timing diagrams T in G, if $T_1, \ldots, T_n$ are the single-tail diagrams arising from T (Defn. 58), then for all languages $L_1, \ldots, L_n$, if $L_i \models_G T_i$ for each i such that $1 \leq i \leq n$ then $L_1 \cap \ldots \cap L_n \models_G T$.

Definition 93 Let G be a timing diagram logic with semantic modeling relationship $\models_G$. G has the *single-tail subdiagram property* iff for all timing diagrams T in G and all languages L such that $L \models_G T$, if $T_1, \ldots, T_n$ are the single-tail diagrams arising from T (Defn. 58), then $L \models_G T_i$ for each i such that $1 \leq i \leq n$.

Theorem 5 $\mathrm{TD}^{\mathrm{TI}}[\mathcal{C}]$ *has the single-tail intersection property.*

Proof Let T be a timing diagram and let $T_1, \ldots T_n$ be all the single-tail diagrams of T. Let $L_1, \ldots, L_n$ be languages such that forall $1 \leq i \leq n$, $L_i \models_{\mathrm{Inv}} T_i$. Assume $L_1 \cap \ldots \cap L_n \not\models_{\mathrm{Inv}} T$. Then by definition there exists some tail object t for which an object-index assignment for its enabling objects E cannot be extended to include t. Let T_t be the single-tail diagram for t. By definition, L_t such that $L_t \models_{\mathrm{Inv}} T_t$ is one of the languages in the intersection $L_1 \cap \ldots \cap L_n$. By definition of intersection, $L_1 \cap \ldots \cap L_n$ must be able to extend all object-index assignments for E to include t, so $L_1 \cap \ldots \cap L_n \models_{\mathrm{Inv}} T$.

□

Theorem 6 $\mathrm{TD}^{\mathrm{TI}}[\mathcal{N}]$ *does not have the single-tail intersection property.*

Proof Consider the following $\mathrm{TD}^{\mathrm{TI}}[\mathcal{N}]$ diagram and its two single-tail diagrams:

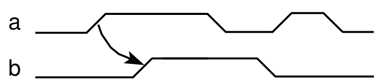
The bounds in the single-tail diagrams are meaningless since the variable n appears only once in each diagram. The first single-tail diagram requires that every rising transition on a be followed by a rising transition on b. The second single-tail diagram requires that every rising transition on a be followed by a rising transition on c. The intersection of these languages, therefore, would include the following, which is not in the language of the original diagram.

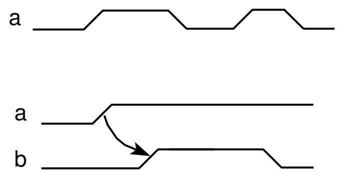
a	0	1	0	0	$\cdots$
b	0	0	1	0	$\cdots$
c	0	0	0	1	$\cdots$
	0	1	2	3	$\cdots$

□

Theorem 7 $\mathrm{TD}^*[\emptyset]$ *does not have the single-tail intersection property.*

Proof Consider the following TD^*diagram and its single-tail diagrams:





The language containing only the following word is described by each of the single-tail diagrams, but not by the original diagram:

a	0	1	0	1	0	1	0	1	0	1	1	1	$\cdots$
b	0	0	1	1	1	1	0	0	1	1	0	0	$\cdots$
	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$

$\square$

The two single-tail diagrams locate their second iterations at different indices. The upper single-tail diagram starts its second iteration at index 4. The lower single-tail diagram starts its second iteration at index 6. This misalignment prevents the intersected language from describing the original diagram because in the original, all next iterations must be aligned between the two single-tail diagrams.

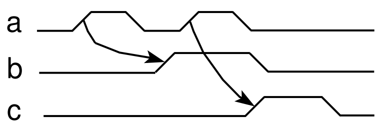
Theorem 8 $\mathrm{TD}^{\mathrm{TI}}$ *has the single-tail subdiagram property.*

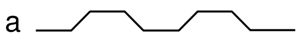
Proof Let T be a $\mathrm{TD}^{\mathrm{TI}}$timing diagram and let $T_1, \ldots T_n$ be all the single tail subdiagrams of T. Let L be a language such that $L \models_{\mathrm{Inv}} T$. Assume that for some T_i, $L \not\models_{\mathrm{Inv}} T_i$. Then there exists an object-index assignment for the enabling objects of T_i that cannot be extended to include the tail object t of T_i. This however, contradicts the definition of $L \models_{\mathrm{Inv}} T$ (Defn. 67), so $L \models_{\mathrm{Inv}} T_i$ and the property must hold.

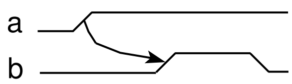
$\square$

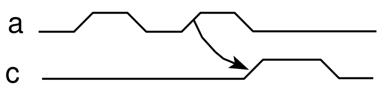
Theorem 9 $\mathrm{TD}^*[\emptyset]$ *does not have the single-tail subdiagram property.*

Proof Consider the following diagram and its single-tail diagrams:









The timing diagram on the top would describe a word in which every other rising transition on a was followed by a rising transition on b. The middle single-tail diagram, however, requires that every rising transition on a be followed by a rising transition on b. A word of the form

a	0	1	0	1	0	...
b	0	0	1	0	0	...
	0	1	2	3	4	...

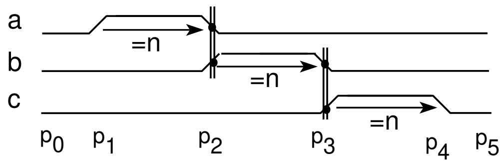
would therefore satisfy the original, but not the single-tail diagram.

□

3.3 Decidability And Timing Diagrams

Decidability determines the theoretical boundary between algorithmic and interactive verification. Automated language containment verification relies on the problems of language emptiness, containment, and equivalence [55]. These problems are all decidable for regular languages, emptiness is decidable for context-free languages but undecidable for context-sensitive languages, and containment and equivalence are undecidable for context free languages [45]. As a result, existing language containment verification systems support only regular languages.

This section discusses the decidability of verification problems for timing diagram languages. Timing diagram languages cut across the Chomsky hierarchy. Section 3.1 identified a subset of timing diagrams corresponding to regular languages, namely those in $\mathrm{TD}^{\mathrm{TI}}[\mathcal{C} - \infty]$. The following diagram is context-sensitive, corresponding to context-sensitive language $a^n b^n c^n$:



Syntactic classifications have not been proven of HHL timing diagram languages with respect to the Chomsky hierarchy. However, timing diagrams containing no temporal variables in time-bound expressions should be regular, while those containing more than one use of a temporal variable are likely either context-free or context-sensitive.

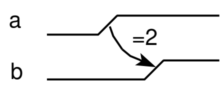
Timing diagram languages resemble *bounded languages*, those with form ${w_1}^*{w_2}^* \ldots {w_n}^*$, where each w_i is a word [33]. In the case of timing diagrams, each w_i represents a time slice. Language containment is decidable for bounded languages, regardless of their position in the Chomsky hierarchy [48]. Unfortunately, this result does not apply directly to timing diagrams. The bounded language correlation captures a single instance of a timing diagram, but it cannot capture the multiple instances allowed by the iterative and invariant semantics.

Fortunately, decision procedures for certain problems can be constructed from the single-instance results; other decision procedures arise from timing diagram semantics. Section 3.3.1 proves that emptiness is decidable for timing diagram languages. Section 3.3.2 discusses the *regular containment problem*, that of whether a regular language is contained in a timing diagram language. This problem is proven decidable for temporally unambiguous timing diagrams and finite regular languages. The decision procedure provides a first step in providing algorithmic support for testing timing diagram properties against circuit diagrams and ASM charts, both of which correspond to regular languages [45, 81]. Section 3.3.3 discusses the applicability of the regular containment decision method to the *timing diagram containment problem*, the general language containment problem on timing diagram languages. This property is useful for determining when one timing diagram satisfies the requirements of another.

3.3.1 The Emptiness Problem

Timing diagram languages are always non-empty. By Definition 67, every TD^{TI} timing diagram has at least one level object assume portion P_1, either explicitly or implicitly. The language of any timing diagram includes those words that fail to satisfy its assume portion. TD^{TI} languages are always non-empty because a word can always be found to violate P_1. This same technique applies to TD^* diagrams with a level-object in P_1. Other TD^* diagrams either have P_1 empty or have only "don't-care" objects in P_1. These diagrams have non-empty languages because the definition of well-formed timing diagrams (Defn. 60) requires all such timing diagrams to have satisfiable time-bound expressions. There must therefore exist patterns of signals that are readable from the diagram that are in its language.

Example 31 Suppose T is the timing diagram

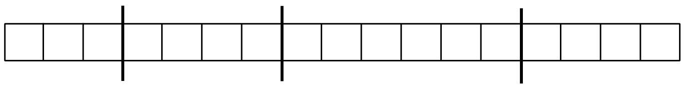


Interpreted as a TD^{TI} timing diagram with the rising transition object on a in P_1, any word in which a is constantly 0 is in the language of T. Interpreted as a TD^* timing diagram with P_1 empty, the following word is contained in the language of T:

a	0	1	1	1
b	0	0	0	1
	0	1	2	3

3.3.2 The Regular Containment Problem

The regular containment problem is decidable for temporally unambiguous timing diagrams and finite regular languages. The problem is still open for temporally ambiguous timing diagrams and ω-regular languages. Consider a finite automaton A accepting by final state and a timing diagram T interpreted under the iterative semantics. Let W be a word over the alphabets of both A and T. The iterative description test (Defn. 68) compares W against T from each of several indices; each test determines where the next test will begin. W can therefore be divided into subwords



where each long line indicates the start of a new object-index match search. Each delineated subword corresponds to a single instance of the description test. Intuitively, each subword could be tested for containment in $\mathcal{L}(T)$, and W would iteratively describe T if all of the subwords described T. In order for this approach to work, the long lines must divide W at the indices where each next object-index match search is begun; this requires precision in the definition of the single-instance language corresponding to T. If this language contains segments with additional characters at the end, the subword boundaries in W may not align with the indices tested for iterative description.

The decision procedure constructs two machines from T. The first, M_T, accepts certain words for which the next object-index match for P_1 is either defined for all objects and extendible to the next object-index match for T, or is undefined for some object in P_1. The second, M_{TBAD}, accepts those words for which the next object-index match is defined for all objects in P_1 but is not extendible to the next object-index match for T. Words accepted by M_{TBAD} do not iteratively describe T by Definition 68.

A simple modification of M_T, M_{T_F}, restricts M_T to accepting only those words for which the next object-index match for P_1 is defined for all objects and extendible to the next object-index match for T. There is no explicit construction algorithm for M_{T_F}; it is formed by removing one of the final states from M_T.

Words accepted by M_T iteratively describe T, but not all words that iteratively describe T are accepted by M_T. M_T accepts a word iff its final letter is the start of the subsequent object-index match search. A word containing two successive next object-index matches for T is not accepted by M_T. The justification for this will be clear shortly. No restriction is placed on the length of words accepted by M_{TBAD}.

Two relations are computed on the states of A using M_{T_F} and M_{TBAD}. The first relates states q and q' if A with start state q and final state q' accepts some word in M_{T_F}. The second identifies those states from which a word in M_{TBAD} is accepted. If any state in the second relation is reachable by chains of states in the first, then A accepts a word that is not iteratively described by T. In other words, M_{T_F} helps identify states where the boundary lines on words can be drawn and M_{TBAD} identifies those states that cannot start a segment without sacrificing containment.

Computing these relations requires that emptiness be decidable on the intersections of M_{T_F} or M_{TBAD} with A. A is not guaranteed to accept a bounded language, so the intersection of A with either machine language is not guaranteed to be bounded. The decision procedure therefore cannot rely on the results for bounded languages. Fortunately, the bounded languages are a subclass of another class containing the regular languages and for which containment, emptiness, and equivalence are decidable. This class, the finite unions of languages accepted by reversal-bounded, 1-counter machines, was recently proved by Ibarra to be the largest for which these three problems are known to be decidable [48]. Such machines contain a two-way read-only head and one counter which can hold any natural number. Reversal-bounded means that the input head makes at most a bounded number of changes of direction while reading any string. Without bounded reversals, the emptiness, containment, and equivalence problems are undecidable. This justifies the restriction of M_T, M_{T_F}, and M_{TBAD} to processing only single instances of timing diagrams: they must process only a finite number in order to maintain bounded reversals.

The decision procedure for the invariant semantics is much simpler and uses only M_{TBAD}. Since a word that is invariantly described by T must be described from every index, the test consists of checking whether any reachable state of A accepts a word accepted by M_{TBAD}.

The formal presentations of the decision procedures and their proofs begin with the definitions of counter machines, a summary of Ibarra's results, and some preliminary definitions needed for both constructions. The constructions and correctness proofs for M_T and M_{TBAD} appear in the subsequent sections.

Preliminaries

A two-way k-counter machine has a finite-state control, a two-way, read-only head over a finite, bounded-length input tape, and k counters, each of which can store any non-negative integer. Transitions are based on the current state, the letter being read, and whether each counter contains zero; the transition indicates a next state, which direction, if any, to move the input head, and whether to increment, decrement, or hold each counter. The formal definition is reproduced from [47].

Definition 94

1. A two-way k-counter machine M is a tuple $\langle k, K, \Sigma, \lhd, \rhd, \delta, q_0, F\rangle$ where K, Σ, $\lhd$, $\rhd$, q_0, and F are the states, inputs, left and right endmarkers, initial state, and accepting states, respectively. δ is a mapping from $K \times (\Sigma \cup \{\lhd, \rhd\}) \times \{0,1\}^k$ into $K \times \{-1,0,1\} \times \{-1,0,1\}^k$. Intuitively, the range of δ indicates which state to move into, which direction, if any, to move the head, and whether to increment, decrement, or hold each counter.

2. A *configuration* of M on an input $\lhd x \rhd$, for $x \in \Sigma^*$, is given by a $(k+3)$ tuple

$$(q, \lhd x \rhd, i, c_1, \ldots, c_k)$$

denoting the fact that M is in state q with the input head reading the i^{th} symbol of $\lhd x \rhd$, and $c_1, \ldots, c_k$ the values stored in the k counters.

3. Relation $\Rightarrow$ is defined between configurations as follows: $(q, \lhd x \rhd, i, c_1, \ldots, c_k) \Rightarrow (p, \lhd x \rhd, i + d, c_1 + d_1, \ldots, c_k + d_k)$ if a is the i^{th} symbol of $\lhd x \rhd$ and $\delta(q, a, \lambda(c_1), \ldots, \lambda(c_k))$ contains $(p, d, d_1, \ldots, d_k)$ where $\lambda(c_i) = 0$ if $c_i = 0$ and $\lambda(c_i) = 1$ if $c_i \neq 0$. $\Rightarrow^*$ denotes the transitive closure of $\Rightarrow$.

4. A string $x \in \Sigma^*$ is *accepted* by M if $(q_0, \lhd x \rhd, 1, 0, \ldots, 0) \Rightarrow^* (q, \lhd x \rhd, i, c_1, \ldots, c_k)$ for some $q \in F$, $1 \leq i \leq |\lhd x \rhd|$, and non-negative integers $c_1, \ldots, c_k$. The set of strings accepted by M form the *language* of M and will be denoted $\mathcal{L}(M)$.

Deterministic two-way counter machines are denoted 2DCM. Languages accepted by counter machines are characterized by two parameters: the number of counters required and the number of times the head changes direction on the input tape. Denoting these by c and r, respectively, the following results about 2DCM(c, r) are due to Ibarra [48]:

Theorem 10 (Ibarra, et al., 1993)

- *The emptiness problem for 2DCM(1, r) is decidable for every* $r \geq 1$.
- *For each c,* $\bigcup_r$ 2DCM(c, r) *is effectively closed under complementation, intersection, and union. In particular, the class of languages recognized by two-way deterministic reversal-bounded multicounter machines is effectively closed under boolean operations.*
- *The containment and equivalence problems for* $\bigcup_r$ 2DCM$(1, r)$ *are decidable.*
- *The class of 2DCM*$(1, \infty)$ *on bounded languages is effectively closed under complementation, intersection, and union.*

Both machine construction algorithms assume a particular representation for time bound objects. A variable v_i, called a *slice variable*, is associated with each time slice (p_i, p_{i+1}). An ordering can be imposed on the slice variables that matches the ordering on the time points. All constraints must be in one of the forms:

$$v_{a_1} + \ldots + v_{b_1} < v_{a_2} + \ldots + v_{b_2} + n$$

$$v_{a_1} + \ldots + v_{b_1} > v_{a_2} + \ldots + v_{b_2} + n$$

where n is a natural number constant and each v_i is a distinct slice variable. The following algorithm converts time bound objects to this representation. The lemmas following the algorithm establish that all time-bound expressions are captured by the results of this algorithm and that all resulting expressions are in the desired form.

Algorithm 5 Let T be a temporally unambiguous timing diagram and let $v_0, \ldots, v_{k-1}$ be the slice variables for T. The *constraints on* T is a set S of linear inequalities constructed as follows:

1. For each $(p_i, p_j) \in E_S$ assigned a time bound expression $[l, u]$, add inequalities $l < v_i + \ldots + v_{j-1} + 1$ and $u > v_i + \ldots + v_{j-1} - 1$ to S.

2. If there exists a non-slice variable x that appears on the less-than side of one inequality and on the greater-than side of another, rewrite all inequalities involving x so that the occurrences of x are all in positive form on one side of the inequality and all other terms are on the other side of the inequality. The number of occurrences of x appearing in inequality e is denoted $X_{\#}(e)$. For each pair of inequalities e_1 and e_2 both involving x, if the x's are on the less-than side of e_1 and on the greater-than side of e_2, and $X_{\#}(e_1) \leq X_{\#}(e_2)$ add inequality $e_1' < e_2' - (X_{\#}(e_2) - X_{\#}(e_1)) + 1$ to S, where e_i' is the non-x side of inequality e_i. When all such pairs have been processed, remove any inequalities involving x from S. Rewrite all expressions in S so that no variable (slice or non-slice) appears on both the left and right hand sides of any expression. Repeat this step until no non-slice variables exist that appear on the less-than side of one inequality and on the greater-than side of another inequality.

3. Remove any inequality involving a non-slice variable from S.

4. Rewrite each inequality in S such that

(a) Every slice variable appears as a positive value in the inequality.

(b) The inequality contains at most one positive constant and no negative constants.

Lemma 9 *All constraints on T are in proper constraint form.*

Proof After Step 1, all expressions in S are inequalities. By the end of Step 3, all expressions contain only slice variables and constants. The algorithm does not leave Step 2 unless no inequality has the same slice variable on both sides. Step 4 forces the other properties of proper constraint form to hold.

□

Lemma 10 *The constraints on T capture exactly the restrictions originally present in T.*

Proof Given an object-index assignment I for the time points of T, assume that time slice variable v_i represents the value $I(p_{i+1}) - I(p_i)$.[1] Under this assumption, it is easily proven that expression $I(p_{i+j}) - I(p_i)$ can be replaced with the expression $v_i + v_{i+1} + \cdots + v_{i+j-1}$ without altering the value of the overall expression.

Consider successor edge object (p_i, p_j) such that $B((p_i, p_j))$ is defined and equals $[l, u]$. The definitions for satisfying constraints and strictly satisfying constraints require that expressions $l \leq I(p_j) - I(p_i)$ and $I(p_j) - I(p_i) \leq u$ be satisfied. By the argument above, this is equivalent to writing expressions $l \leq v_i + \cdots + v_j$ and $v_i + \cdots + v_j \leq u$. After Step 1, the requirements expressed in S are therefore equivalent to those expressed by the time bound objects. The remaining steps in Algorithm 5 must be shown to preserve the requirements expressed in S.

A non-slice variable is only constrained if it is given both upper and lower bounds; otherwise, it is trivially satisfied in T. Step 2 combines each upper and lower bound relationship on a non-slice

[1] This property is proven of the construction algorithm in Section 3.3.2, Lemma 16.

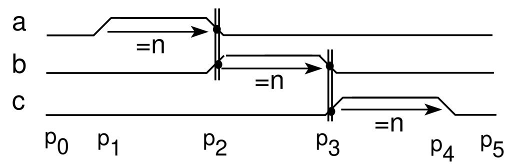
variable x into a single constraint on the terms that comprise x. After all pairs have been processed, the only inequalities still involving x are the expressions that were in S at the start of the step, albeit in rewritten form. Since all possible combinations of bounds on x were paired up, the original expressions involving x are redundant, and can be removed safely from S.

Any other inequality involving non-slice variables that remains in S by Step 3 could not have had both an upper and a lower bound in S, or it would have been processed in Step 2. Such expressions can be removed without any loss of information. Step 4 performs only information-preserving rewrites. Therefore, under the assumption that slice variables are properly computed relative to the assignments to time points, the constraints of T exactly capture the time bound objects of T.

$\square$

Algorithm 5 is conservative; it can generate more expressions than are necessary to capture the time-bound restrictions of T, as shown in Example 32.

Example 32 Algorithm 5 generates equations $v_1 = v_3$, $v_1 = v_2$, and $v_3 = v_2$ for this timing diagram, even though any two would suffice mathematically.



The conservative generation is necessary for the correct operation of the machines; the machines are designed to check the satisfaction of each time point in turn. Semantically, this check involves checking all time-bound expressions up to that time point. Algorithm 5 generates the extra constraints to capture this aspect of the semantics.

Lemma 11 *The constraints of any prefix of T up to p_i are contained in the constraints relative to time points up to and including p_i.*

Proof Let p_i be any time point in T and let T' be the prefix of T ending at p_i. The restrictions of every time-bound object in T' are captured in some constraint by Lemma 10. Let c be any constraint for T'. Since the restriction of every time-bound object in T is captured by its constraints, if c is not a constraint in T, then there must exist a time-bound object in $T - T'$ that uses the same slice variables as the time-bound objects that yielded c. However, Step 2 processes every pair of expressions that use the same time points, so the expressions that generated c for T' must also have generated it for T.

□

The language for a temporally unambiguous timing diagram is determined by the objects at its time points and the signal patterns required to hold between time points. The letters capturing these are consistent with the satisfaction definitions (Defns. 63 and 64) in Chapter 2. Three letters are defined below for each time point: a_i captures the signal pattern requirements for time slice (p_i, p_{i+1}), ap_{iinit} and ap_i captures the requirements for an index to satisfy an object.

Definition 95 Let T be a temporally unambiguous timing diagram.

1. The *assignment corresponding to time slice* (p_i, p_{i+1}) is a signal-value assignment a_i for T where for each signal name n,
   (a) If T is being interpreted under strict semantics
      i. If n contains a rising transition object or a high-level object at p_i, $a_i(n) = 1$.
      ii. If n contains a falling transition object or a low-level object at p_i, $a_i(n) = 0$.
   (b) If T is being interpreted under non-strict semantics
      i. If n has a low-level object spanning p_i and p_{i+1}, $p_i \neq p_0$, and there exist time points $p_a \leq p_i$ and $p_b \geq p_{i+1}$, $a_i(n) = 0$ if either

A. there exists a concurrent edge object at p_a involving the same low-level object on n that spans p_i and p_{i+1} and p_b is the smallest time point larger than p_{i+1} containing a transition object on n, or

B. there exists a concurrent edge object at p_b involving the same low-level object on n that spans p_i and p_{i+1} and p_a is the largest time point smaller than p_i containing a transition object on n.

ii. If n has a high-level object spanning p_i and p_{i+1}, $p_i \neq p_0$, and there exist time points $p_a \leq p_i$ and $p_b \geq p_{i+1}$, $a_i(n) = 1$ if either

A. there exists a concurrent edge object at p_a involving the same high-level object on n that spans p_i and p_{i+1} and p_b is the smallest time point larger than p_{i+1} containing a transition object on n, or

B. there exists a concurrent edge object at p_b involving the same high-level object on n that spans p_i and p_{i+1} and p_a is the largest time point smaller than p_i containing a transition object on n.

2. The *initial letter* for each time point p_i other than the maximal and minimal points is denoted ap_{iinit} and is a signal-value assignment such that:

(a) If T is being interpreted under strict semantics, for all time points other than p_1, ap_{iinit} be a_{i-1}. ap_{1init} is the signal value assignment over those signal names either with transition objects or involved in concurrent edge objects at p_1. Waveform w is assigned 0 if w has a rising transition object or a low-level object involved in a concurrent edge object at p_1. Waveform w is assigned 1 if w has a falling transition object or a high-level object involved in a concurrent edge object at p_1.

(b) If T is being interpreted under non-strict semantics

- If p_i contains a concurrent edge object, ap_{iinit} assigns 0 to each signal n in the concurrent edge object that contains either a rising transition object or a low-level object and ap_{iinit} assigns 1 to each signal n in the concurrent edge object that contains either a falling transition object or a high-level object.

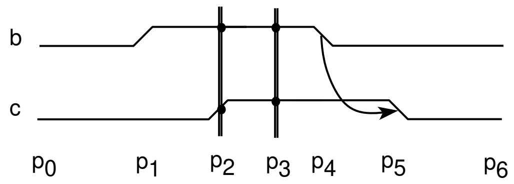
- If p_i does not contain a concurrent edge object, ap_{iinit} assigns 0 to each signal that contains a rising transition object at p_i and ap_{iinit} assigns 1 to each signal that contains a falling transition object at p_i.

3. The *second time point letter* for each time point p_i is denoted ap_i and is a signal-value assignment such that

   (a) If T is being interpreted under strict semantics, ap_i is the same as a_i.

   (b) If T is being interpreted under non-strict semantics, ap_i assigns 0 to each signal containing a falling transition object at p_i, assigns 1 to each signal containing a rising transition object at p_i, and includes the assignments made in a_{i+1} for all n.

Example 33 The labels across the bottom of this timing diagram denote time points. Suppose the diagram is interpreted non-strictly. Assignment a_2 sets both b and c to 1. Assignment a_4 is empty. Assignment $ap_{1\text{init}}$ sets b to 0 but leaves c unconstrained. Assignment $ap_{3\text{init}}$ sets both b and c to 1. Assignment ap_4 sets b to 0 and c to 1.



Each construction algorithm defines a machine in terms of its transitions. If a transition contains symbol x, one copy of the rule is created for any valid value for x. Letter a denotes any element of Σ unless otherwise constrained. The two machines are structurally quite similar. Each time point is sought in the word in turn. When an index matching the time point is located, all constraints relative to the time point are checked.

The M_T Construction Algorithm

Algorithm 6 Let T be a temporally unambiguous, anchor-free timing diagram. Let *split* be the split point of T and let $p_{\max}$ be the maximal time point of T. Let M_T be a 2DCM $\langle 1, K, \Sigma, \triangleleft, \triangleright, \delta, q_{p_0}, \{q_f, q_e\}\rangle$ where Σ is the set of all assignments to the signals in T, δ contains the transitions generated below, and K consists of all states appearing in some transition of δ.

- Garbage state rule. The garbage state loops infinitely to reject strings.
  - $(q_g, x, x) \rightarrow (q_g, 0, 0)$
- Final states rules. The machine enters q'_f when it has found the index that should start the next object-index match search. The machine accepts the word if this is the last index in the string, else it rejects the word.
  - $(q'_f, \triangleright, x) \rightarrow (q_f, 0, 0)$
  - $(q'_f, a, x) \rightarrow (q_g, 0, 0)$ for each $a \neq \triangleright$
- If an error is encountered while processing a constraint in the antecedent, the next object-index match search begins in the index immediately following the one satisfying the time point whose constraint failed. These transitions return the machine to that index, then enter q'_f.
  - $(qf_{ii}, x, x) \rightarrow (q'_f, 1, 0)$
  - $(qf_{ij}, ap_{i+1init}, x) \rightarrow (q2f_{ij}, 1, 0)$ for all $i > j$
  - $(q2f_{ij}, ap_{i+1}, x) \rightarrow (qf_{ij+1}, 0, 0)$ for all $i > j$
  - $(q2f_{ij}, a, x) \rightarrow (qf_{ij}, 0, 0)$ for all $a \neq ap_{i+1}$ and all $i > j$
  - $(qf_{ij}, a_i, x) \rightarrow (qf_{ij}, 1, 0)$ for all $i > j$
- Transitions to locate index satisfying time point p_{i+1}.
  - $(qp_i, ap_{i+1init}, x) \rightarrow (q2p_i, 1, 0)$
  - $(q2p_i, ap_{i+1}, x) \rightarrow (qp_{i+1}, 0, 0)$ if no constraints are relative to p_{i+1} and p_i is not $p_{\max-2}$

- $(q2p_i, ap_{i+1}, x) \rightarrow (q'_f, 0, 0)$ if no constraints are relative to p_{i+1} and p_i is $p_{\max-2}$
- $(q2p_i, ap_{i+1}, x) \rightarrow (qrp_i c_1, 0, 0)$ if there are constraints relative to p_{i+1}
- $(q2p_i, a, x) \rightarrow (qp_i, 0, 0)$ for all $a \neq ap_{i+1}$
- $(qp_i, a_i, x) \rightarrow (qp_i, 1, 0)$
- $(qp_i, a, x) \rightarrow (q'_f, 1, 0)$ for all a that are neither $ap_{i+1init}$ nor a_i, if $i \leq$ split
- $(qp_i, a, x) \rightarrow (q_g, 1, 0)$ for all a that are neither $ap_{i+1init}$ nor a_i, if $i >$ split

- Transitions for encountering right endmarker, generated for all i such that p_i is a time point smaller than $p_{\max-2}$:

  - $(q_{p_i}, \rhd, x) \rightarrow (q_e, 0, 0)$ if $i \leq$ split
  - $(q_{p_i}, \rhd, x) \rightarrow (q_g, 0, 0)$ if $i >$ split

- Transitions to generate to check constraints relative to time point p_i (the rules introduced in this section must be generated once for each constraint c_k relative to p_i):

  If slice variable v_j appears n times on greater-than side of c_k

  - $(qp_i c_{kjInc}, a, x) \rightarrow (qp_i c_{kjInc1}, 0, 1)$
  - $(qp_i c_{kjInch}, a, x) \rightarrow (qp_i c_{kjInch+1}, 0, 1)$ for all $1 \leq h < n$
  - $(qp_i c_{kjIncn}, ap_{j+1init}, x) \rightarrow (q2p_i c_{kjInc}, 1, 0)$
  - $(q2p_i c_{kjInc}, ap_{j+1}, x) \rightarrow (qp_i c_{kj+1Inc}, 0, 0)$
  - $(q2p_i c_{kjInc}, a, x) \rightarrow (qp_i c_{kjInc}, 0, 0)$ for each $a \neq ap_{j+1}$
  - $(qp_i c_{kjIncn}, a, x) \rightarrow (qp_i c_{kjInc}, 1, 0)$ for each $a \neq ap_{j+1init}$

  If slice variable v_j does not appear on greater-than side of c_k and v_j is smaller than v_{gtmax}

  - $(qp_i c_{kjInc}, ap_{j+1init}, x) \rightarrow (q2p_i c_{kjInc}, 1, 0)$
  - $(q2p_i c_{kjInc}, ap_{j+1}, x) \rightarrow (qp_i c_{kj+1Inc}, 0, 0)$
  - $(q2p_i c_{kjInc}, a, x) \rightarrow (qp_i c_{kjInc}, 0, 0)$ for all $a \neq ap_{j+1}$
  - $(qp_i c_{kjInc}, a_j, x) \rightarrow (qp_i c_{kjInc}, 1, 0)$

If v_j is larger than v_{gtmax} and no constant appears in c_k:

- $(qp_i c_{jkInc}, x, x) \rightarrow (qr2p_i c_k, 0, 0)$

If v_j is larger than v_{gtmax} and c_k is of the form $v_{a_1} + \ldots + v_{b_1} < v_{a_2} + \ldots + v_{b_2} + n$, where all v's are slice variables and n is a non-zero constant:

- $(qp_i c_{kjInc}, x, x) \rightarrow (qp_i c_{k\mathrm{const}1}, 0, 1)$
- $(qp_i c_{k\mathrm{const}h}, x, x) \rightarrow (qp_i c_{k\mathrm{const}h+1}, 0, 1)$ for each $1 \leq h < n$.
- $(qp_i c_{k\mathrm{const}n}, x, x) \rightarrow (qr2p_i c_k, 0, 0)$

If v_j is larger than v_{gtmax} and c_k is of the form $v_{a_1} + \ldots + v_{b_1} + n < v_{a_2} + \ldots + v_{b_2}$, where all v's are slice variables and n is a non-zero constant:

- $(qp_i c_{kjInc}, x, x) \rightarrow (qp_i c_{k\mathrm{const}1}, 0, -1)$
- $(qp_i c_{k\mathrm{const}h}, x, 1) \rightarrow (qp_i c_{k\mathrm{const}h+1}, 0, -1)$ for each $1 \leq h < n$.
- $(qp_i c_{k\mathrm{const}h}, x, 0) \rightarrow (qf_{i+1j}, 0, 0)$ for all $1 \leq h \leq n$, if $i \leq$ split
- $(qp_i c_{k\mathrm{const}h}, x, 0) \rightarrow (q_g, 0, 0)$ for all $1 \leq h \leq n$, if $i >$ split
- $(qp_i c_{k\mathrm{const}n}, x, 1) \rightarrow (qr2p_i c_k, 0, 0)$

Return to the beginning of the input to process less-than side

- $(qr2p_i c_k, \lhd, x) \rightarrow (qp_i c_{k0Dec}, 0, 0)$
- $(qr2p_i c_k, a, x) \rightarrow (qrp_i c_k, -1, 0)$ for each $a \neq \lhd$

If slice variable v_j appears on less-than side of c_k and $j \leq i$

- $(qp_i c_{kjDec}, a, 1) \rightarrow (qp_i c_{kjDec1}, 0, -1)$
- $(qp_i c_{kjDec}, a, 0) \rightarrow (qf_{i+1j}, 0, 0)$ if $i \leq$ split
- $(qp_i c_{kjDec}, a, 0) \rightarrow (q_g, 0, 0)$ if $i >$ split
- $(qp_i c_{kjDech}, a, 1) \rightarrow (qp_i c_{kjDech+1}, 0, -1)$ for all $1 \leq h < n$
- $(qp_i c_{kjDech}, a, 0) \rightarrow (qf_{i+1j}, 0, 0)$ for all $1 \leq h \leq n$ if $i \leq$ split

- $(qp_i c_{kjDech}, a, 0) \rightarrow (q_g, 0, 0)$ for all $1 \leq h \leq n$ if $i >$ split
- $(qp_i c_{kjDecn}, ap_{j+1init}, 1) \rightarrow (q2p_i c_{kjDec}, 1, 0)$
- $(q2p_i c_{kjDec}, ap_{j+1}, x) \rightarrow (qp_i c_{kj+1Dec}, 0, 0)$
- $(q2p_i c_{kjDec}, a, x) \rightarrow (qp_i c_{kjDec}, 0, 0)$ for all $a \neq ap_{j+1}$
- $(qp_i c_{kjDecn}, a, 1) \rightarrow (qp_i c_{kjDec}, 1, 0)$ for each $a \neq ap_{j+1init}$

If slice variable v_j does not appear on less-than side of c_k and $j \leq i$

- $(qp_i c_{kjDec}, ap_{j+1init}, x) \rightarrow (q2p_i c_{kjDec}, 1, 0)$
- $(q2p_i c_{kjDec}, ap_{j+1}, x) \rightarrow (qp_i c_{kj+1Dec}, 0, 0)$
- $(q2p_i c_{kjDec}, a, x) \rightarrow (qp_i c_{kjDec}, 0, 0)$ for all $a \neq ap_{j+1}$
- $(qp_i c_{kjDec}, a_j, x) \rightarrow (qp_i c_{kjDec}, 1, 0)$

If $j = i + 1$ and c_k is the last constraint to be checked for p_{i+1}

- $(qp_i c_{kjDec}, x, 0) \rightarrow (qf_{i+1j}, 0, 0)$ if $i \leq$ split
- $(qp_i c_{kjDec}, x, 0) \rightarrow (q_g, 0, 0)$ if $i >$ split
- $(qp_i c_{kjDec}, x, 1) \rightarrow (qp_{i+1}, 0, 0)$ if p_i is not $p_{\max-2}$
- $(qp_i c_{kjDec}, x, 1) \rightarrow (q'_f i, 0, 0)$ if p_i is $p_{\max-2}$

If $j = i + 1$ and c_k is not the last constraint to be checked for p_{i+1}

- $(qp_i c_{kjDec}, x, 1) \rightarrow (qrp_i c_{k+1}, -1, 0)$
- $(qp_i c_{kjDec}, x, 0) \rightarrow (qf_{i+1j}, 0, 0)$ if $i \leq$ split
- $(qp_i c_{kjDec}, x, 0) \rightarrow (q_g, 0, 0)$ if $i >$ split

Transitions to start checking next constraint associated with p_{i+1}

- $(qrp_i c_k, \triangleleft, x) \rightarrow (qrp_i c_{k\text{deplete}}, 0, 0)$
- $(qrp_i c_k, a, x) \rightarrow (qrp_i c_k, -1, 0)$ for all $a \neq \triangleleft$
- $(qrp_i c_{k\text{deplete}}, x, 1) \rightarrow (qrp_i c_{k\text{deplete}}, 0, -1)$
- $(qrp_i c_{k\text{deplete}}, x, 0) \rightarrow (qp_i c_{k0Inc}, 0, 0)$

Correctness

In order to simplify the correctness proofs, Figures 3.2 and 3.3 provide the transition diagrams for M_T; they are referred to as the "machine diagrams" throughout this section. In the diagrams, rounded rectangles denote blocks for which a transition diagram is defined. Ovals denote the particular states of M_T whose names appear inside them. Transitions from one block to another occur through the arrows entering and leaving the blocks. If a path does not reach the arrow leading out of a block, there is no way out of the block along that path. Ovals with no arrows leaving them denote final states.

All of the lemmas in this section are subproofs needed for the main result of the section, which classifies the language of M_T relative to timing diagram T.

Lemma 12 *Letters a_i and $ap_{i+1init}$ capture exactly the signal patterns of time slice (p_i, p_{i+1}).*

Proof This follows from the definitions of the slice constraints (page 86).

□

Lemma 13 *Let W be a word and let b be the object at time point p_j. If b contains a transition object, index i into W satisfies b iff letters ap_{jinit} and ap_j appear in indices i and $i+1$ of W, respectively. If b does not contain a transition object, index i satisfies b iff it contains letter a_j.*

Proof Let W be a word and let b be the object at time point p_j.

1. If T is interpreted under strict semantics and index i satisfies b, i must assign each waveform 0 if it has a low-level object or a rising transition object at p_j and 1 if it has a high-level object or falling transition object at p_j. If b contains a transition object, index $i+1$ must assign 1 to each waveform with a rising transition object at p_j and 0 to each waveform with a falling transition object at p_j. Under the strict semantics, ap_j is defined (Defn. 95) to be a_j. By

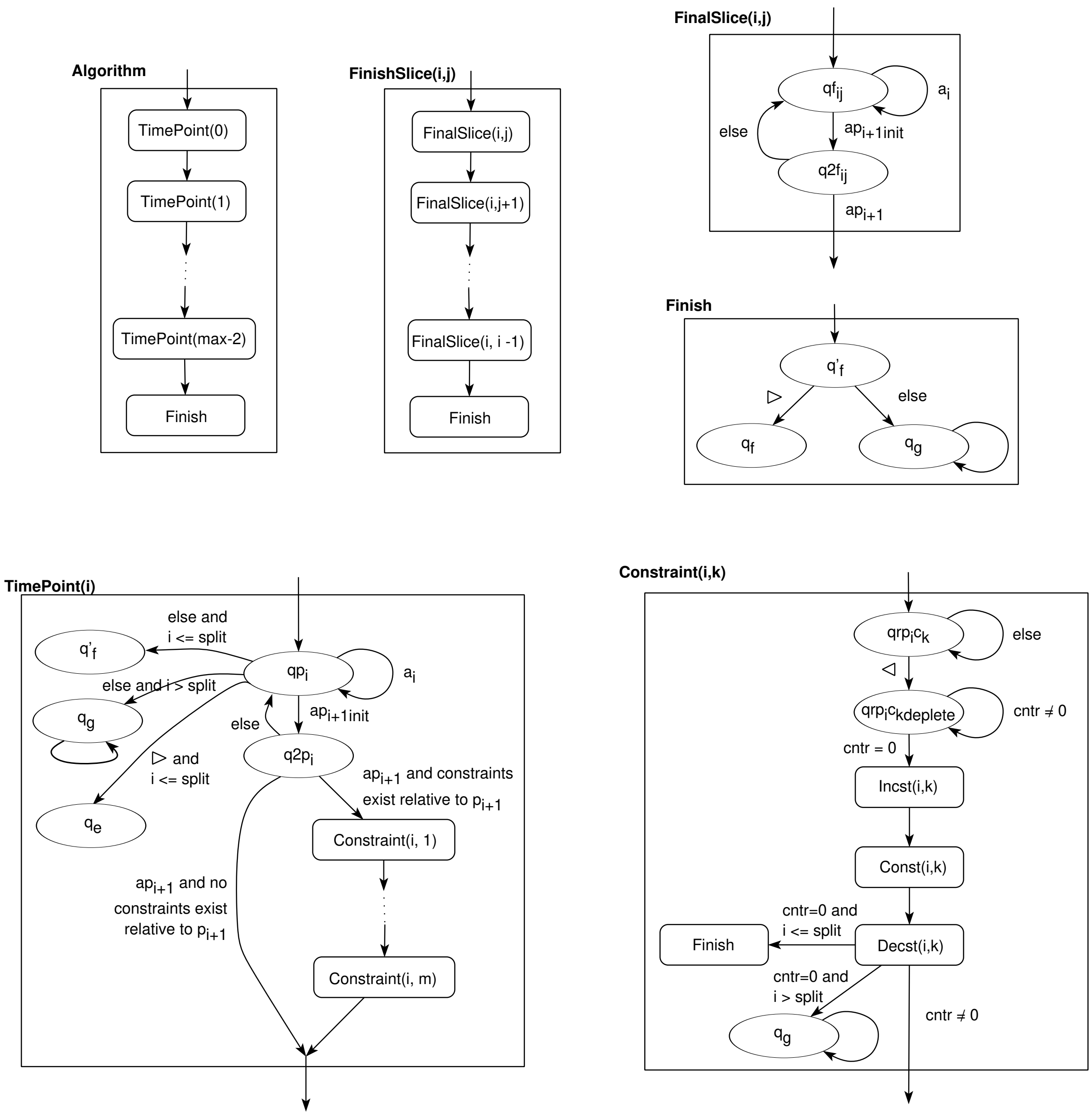


Figure 3.2: Transition diagrams for M_T.

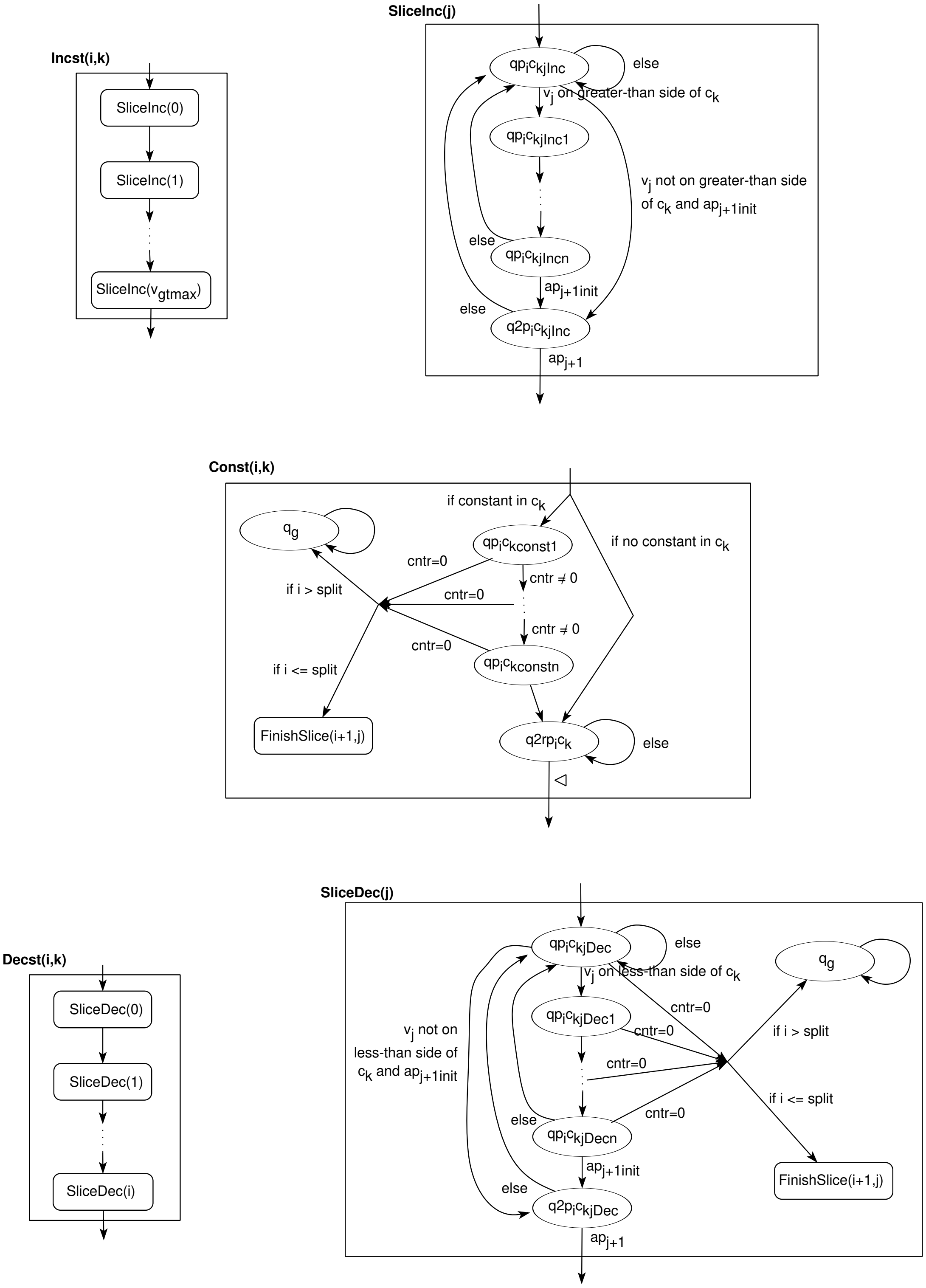


Figure 3.3: Additional transition diagrams for M_T.

Definition 95, a_j assigns 1 to each waveform with a rising transition object or high-level object at p_j and 0 to each waveform with a falling transition object or low-level object at p_j. Index $i+1$ must therefore contain letter a_j.

The definition of ap_{jinit} differs depending upon whether $j = 1$ or $j > 1$. If $j = 1$, then ap_{1init} is defined as the signal value assignment over those signal names either with transition objects or involved in concurrent edge objects at p_1. Waveform w is assigned 0 if w has a rising transition object or a low-level object involved in a concurrent edge object at p_1. Waveform w is assigned 1 if w has a falling transition object or a high-level object involved in a concurrent edge object at p_1. This matches the definition of what is required of index i, so the lemma holds.

Assume $j > 1$. Then by definition, ap_{jinit} is a_{i-1}. If some waveform has a high-level object at time point p_j, then at time point p_{j-1}, it must have had a high-level object or a rising transition object, due to the well-formedness constraints (Defn. 48). In either case, a_{i-1} is defined to be 1. A similar argument shows that if some waveform has a low-level object at p_j, then a_{i-1} must assign 0 to it. Index i and a_{i-1} must still be shown to agree on the values assigned to signals with don't-care values; those are undefined in each case, so the lemma holds.

2. If T is interpreted under non-strict semantics:

   (a) Assume b contains a transition object. Then by Definition 62, if index i satisfies b, then for all waveforms w involved in b, i must satisfy the object on w in time slice (p_{j-1}, p_j) and index $i+1$ must satisfy the objects on w in time slice (p_j, p_{j+1}). From Definition 95, ap_{jinit} has the same restrictions for index i. For each w containing a transition object, ap_j has the same restrictions that are on index $i+1$ by Definition 95. It remains to be shown that ap_j has the same restrictions as are on $i+1$ for those w involved at b containing level-objects. If w is involved at b and contains a level object, then index $i+1$ must assign 0 to w if the object is a low-level object and 1 to w if the object is a high-level object. Definition 95 states that ap_j includes all restrictions in a_j. By definition, a_j assigns values to precisely those level objects involved in a concurrent edge object at time point p_j and those values are identical to the ones required of $i+1$. The lemma therefore holds in this case.

(b) Assume b contains no transition object. Then by Definition 62, any index satisfying b must assign 1 to any waveform involved in b containing a high-level object and 0 to any waveform involved in b containing a low-level object. By Definition 95, letter a_j contains the same restrictions, so the lemma holds.

□

Lemma 14 *If M_T leaves block Constraint(j, m) it is reading the same index of word W as it was when it entered block Constraint($j, 1$).*

Proof Given j, M_T leaves Constraint(j, n) reading the same index, regardless of the value of n (assuming it leaves Constraint(j, n) at all). This is shown by the machine diagrams since M_T leaves Constraint(j, n) in the same index as it leaves Decst(j, n) and the transition structure of Decst(j, n) is the same regardless of the value of n. Proving that M_T leaves block Constraint$(j, 1)$ reading the same index as when it entered block Constraint$(j, 1)$ suffices.

The proof is by induction on j. M_T leaves block Constraint$(j, 1)$ reading the same index as when it leaves block SliceDec(j). Assume j is the smallest time point that has constraints relative to it. M_T always enters block SliceDec(0) at the beginning of W. The machine diagram indicates that SliceDec(k) has a similar transition structure to TimePoint(k) if there are no constraints relative to p_{k+1}. A straightforward inductive argument on m shows that M_T must therefore exit SliceDec(j) at the same time point that it entered Constraint$(j, 1)$.

Assume that M_T leaves block Constraint$(j, 1)$ reading the same index as when it entered block Constraint$(j, 1)$. Let j' be the next time point larger than j such that there are constraints defined relative to $j' + 1$. By the inductive assumption, M_T must enter TimePoint(k) reading the same index as when it enters Constraint$(k, 1)$ for any k smaller than j' such that there are constraints defined relative to p_{k+1}. The last portion of Constraint$(k, 1)$ is the sequence of SliceDec blocks, the first of which starts at the beginning of W. By the inductive hypothesis and the transition structure, M_T

therefore left SliceDec($j' - 1$) reading the same index as when it left TimePoint($j' - 1$). Since the transition structure for SliceDec(j') and the part of TimePoint up to Constraint are similar, M_T entered Constraint $(j', 1)$ at the index after the first that satisfied object b'_j. However, the transition structure shows that this is the same index M_T is at when it exits SliceDec(j'). M_T therefore enters and exits Constraint$(j', 1)$ reading the same index as when it exits SliceDec(j'), so M_T leaves block Constraint$(j', 1)$ reading the same index as when it entered block Constraint$(j', 1)$.

□

Lemma 15 *The next object-index match for T, W, 1, and P_1, denoted I, is defined for the object at p_j iff M_T reaches state qp_j. Furthermore, if M_T first reaches state qp_j while reading index i, $I(b_j) = i - 1$, where b_j is the object at time point p_j.*

Proof Assume $j = 1$. By Definition 56, the object at p_1 is the head object of T. By definition of I (Defn. 66), $I(b_1)$ is defined iff $I(b_1) \geq 1$, $I(b_1)$ satisfies b_1, and there is no index smaller than $I(b_1)$ that satisfies b_1. There cannot be any constraints relative to p_1 since p_1 is the head object of T. By Lemma 13, the machine diagrams, and the transition structure, if M_T reaches state qp_1 while reading index i, then index $i - 1$ is the first index in W that satisfies b_1. $I(b_1)$ must therefore equal $i - 1$. If M_T does not reach qp_1, then no index in W satisfies b_1. This base case therefore holds.

Assume the lemma holds for all time points up to p_j and that p_j is smaller than $p_{\max-2}$. Then all enabling objects for b_{j+1} have been assigned indices in I. If M_T reaches state $q2p_j$ and reads ap_{j+1} in index i, index $i - 1$ satisfies b_{j+1} from the machine diagrams and Lemma 13; $i - 1$ is the smallest index larger than $I(b_j)$ that satisfies b_{j+1}. In addition, all indices from $I(b_j) + 1$ through $i - 1$ satisfy the signal patterns of T in time-slice (p_j, p_{j+1}) by the machine diagrams and Lemma 12. If there are no constraints of T relative to p_{j+1}, the lemma holds. If there are constraints of T relative to p_{j+1}, the lemma holds iff M_T leaves block Constraint(j, m) in the machine diagrams when reading index i. This was established in Lemma 14 so this lemma holds.

□

Lemma 16 *If v_j appears on the greater-than side of constraint c_k, then if M_T enters block SliceInc(j), it increments in exactly $I(p_{j+1}) - I(p_j)$ indices while in SliceInc(j).*

Proof M_T enters SliceInc(0) at the beginning of W. The transition structure of block SliceInc is similar to the top part of block TimePoint if states used only to increment the counter are ignored. Lemma 14 establishes that M_T leaves TimePoint at the same index, regardless of which transition it takes out of state $q2p_i$ when reading ap_{i+1}. M_T therefore transitions from SliceInc(i) to SliceInc($i+1$) reading the same index as when it transitioned from TimePoint(i) to TimePoint($i+1$). Lemma 15 establishes that the transition from TimePoint(i) to TimePoint($i+1$) occurs at index $I(b_i)+1$. M_T therefore enters SliceInc(j) reading index $I(b_j)+1$ and leaves SliceInc(j) reading index $I(b_{j+1})+1$. While in block SliceInc(j), M_T increments in each index. M_T therefore must increment in exactly $I(p_{j+1}) - I(p_j)$ indices before leaving SliceInc(j).

□

Lemma 17 *When M_T leaves block Incst(i, k), the counter contains the sum of all slice variables on the greater-than side of constraint c_k.*

Proof M_T increments n times in each index while in SliceInc(j), where n is the number of occurrences of v_j on the greater-than side of c_k. Lemma 16 indicates that this is done for the number of indices falling within time slice (p_j, p_{j+1}). The machine diagrams show that the counter is unchanged for any slice variable that is not on the greater-than side of c_k. By construction, there is an instance of SliceInc for each time slice whose slice variable appears on the greater-than side of c_k, so the lemma holds.

□

Lemma 18 *If v_j appears on the less-than side of constraint c_k, M_T enters block SliceDec(j), and M_T leaves SliceDec(j), then M_T decrements in exactly $I(p_{j+1}) - I(p_j)$ indices while in SliceDec(j).*

Proof The proof is similar to that of Lemma 16.

□

Lemma 19 *If M_T enters block Constraint(j, k), it leaves that block iff constraint c_k associated with time point p_{j+1} is satisfied.*

Proof The machine diagram shows that the first step in Constraint(j, k) is to return to the beginning of W and deplete the counter. Lemma 17 establishes that the counter has properly processed the slice-variables on the greater-than side of c_k when M_T reaches block Const(i, k). If there is no constant in c_k, the transition structure shows that the counter is not affected. If there is a constant, the machine diagram and the transition structure show that the counter is incremented or decremented the correct number of times, depending upon which side of c_k contains the constant. If M_T does not leave block Const(i, k), then the counter must have reached zero. This can only happen if the constant is on the less-than side of c_k and the constant is larger than the value in the counter at the start of Const(i, k). Therefore, if the constraint fails while processing the constant, M_T does not leave Const(i, k), and hence does not leave Constraint(j, k).

If M_T does leave Const(i, k), it reaches Decst(i, k) with the counter accurately reflecting the slice-variables on the greater-than side of c_k and any constant in c_k. The transition structure and the machine diagrams indicate that M_T decrements n times in SliceDec(j), where n is the number of occurrences of v_j on the less-than side of c_k, as long as the counter does not reach zero. M_T does not affect the counter if v_j is not on the less-than side of c_k, so if M_T leaves Decst(i, k), the counter contains the value of the greater-than side of c_k minus the less-than side of c_k. Immediately after leaving Decst(i, k), M_T tests whether or not the counter contains zero. If it does, the constraint failed because the greater-than and less-than sides of c_k were equal. In this case, M_T does not leave Constraint(i, k), as seen in the machine diagram. If the counter does not contain zero, the greater-than side was strictly larger than the less-than side, so the constraint holds. M_T takes one of the other two transitions out of Decst(i, k), either of which takes M_T out of Constraint(i, k).

If M_T does not leave Decst(i,k), then M_T entered but did not leave some SliceDec(j). The machine diagram shows that this can only happen if the counter reached zero. Since M_T accurately counts the less-than side of c_k, the less-than side must have been larger than the greater-than side, so the constraint failed. M_T did not leave Constraint(i,k) in this case, so the lemma holds.

□

Lemma 20 *If M_T enters block Constraint($j,1$), it leaves block Constraint(j,m), where m is the number of constraints relative to p_{j+1}, iff all constraints relative to p_{j+1} are satisfied.*

Proof This follows immediately from Lemma 19.

□

Lemma 21 *M_T leaves block TimePoint(split-1) iff the next object-index match for T, W, 1, and P_1 is defined for all objects in P_1.*

Proof This follows from Lemma 15.

□

In the following theorem, the length of W refers to the length of the word between the endmarkers: the endmarkers themselves are not included.

Theorem 11 *Let T be a temporally unambiguous timing diagram and let W be a word. M_T accepts W iff either*

1. *The next object-index match for T, W, 1, and P_1 is defined for all objects in P_1, the next object-index match for T, W, 1, and $P_1 \cup P_2$, I' is defined for all objects in $P_1 \cup P_2$, and the length of W is one more than the highest index in the range of I'.*

2. *The next object-index match for T, W, 1, and P_1, I, is undefined for some object in P_1 and the length of W is less than or equal to the error index of I.*

Proof By arguments similar to those for Lemma 21, M_T leaves TimePoint$(\max - 2)$ iff the next object-index match for T, W, 1, and $P_1 \cup P_2$ is defined for all objects in $P_1 \cup P_2$. By an argument similar to Lemma 15, M_T leaves TimePoint$(\max - 2)$ reading index $I(b_{\max-1}) + 2$. This follows since the input head is advanced one spot to the right on either transition out of TimePoint$(\max - 2)$, but was not advanced on either transition out of TimePoint$(\max - 2)$. The transition structure of state q'_f shows that M_T transitions to final state q_f from q'_f iff the input head is on the right endmarker. Otherwise, M_T enters the garbage state. If M_T moved to q_f (and hence accepted W), the length of W was therefore one more than the largest index in the range of I', since the largest index in the range of I' was that assigned to object $b_{\max-1}$. If M_T moved to q_g (and hence did not accept W), then W was longer than one more than the highest index in the range of I'.

Assume I is undefined for some object in P_1. If that object was the head object (b_1), then there did not exist an index in W that satisfied b_1. By Definition 66, the error index of I is therefore one larger than the length of W. Since no index satisfied b_1, M_T would have encountered the right endmarker in state qp_0, entered final state q_e, and accepted W.

If I is defined for the head object, let b_j be the earliest object for which I is undefined. By the definition of I (Defn. 66), one of the following three cases must hold:

1. There does not exist an index larger than $I(b_{j-1})$ that satisfies b_j. The argument for this case is the same as that for the head object, only M_T would have entered state q_e from state qp_{j-1}.

2. There exists an index i larger than $I(b_{j-1})$ that satisfies b_j, but there also exists index i' larger than $I(b_{j-1})$ and smaller than i that violates the signal patterns of time slice (p_{j-1}, p_j). By Lemma 12, letters a_{j-1} and ap_{jinit} capture the signal patterns of time slice (p_{j-1}, p_j). From the machine diagram for TimePoint, any letter other than a_{j-1} or ap_{jinit} would result in a transition to q'_f, unless the right endmarker was encountered. Since i is larger than i', the right

endmarker cannot be encountered in i', so M_T transitions to q'_f and is reading index $i'+1$ when it gets there. Since M_T checks every index, i' must be the smallest index larger than $I(b_{j-1})$ that violates the signal patterns of time slice (p_{j-1}, p_j). By Definition 66, the error index in this case is $i'+1$. At this point M_T accepts W iff it is reading the right endmarker in q'_f. If M_T is reading the right endmarker, the length of W must have been $i'+1$, which equals the error index.

3. There exists an index i larger that $I(b_{j-1})$ that satisfies b_j, all indices from $I(b_{j-1})$ through i satisfy the signal patterns of time slice (p_{j-1}, p_j), but some constraint relative to p_j is not satisfied. In this case the error index is defined to be $i+1$ (Defn. 66). By Lemma 15, M_T must have reached state qp_{j-1}. Since index i exists and there exist constraints relative to p_j, M_T must reached block Constraint$(j, 1)$. Since some constraint relative to p_j is not satisfied, by Lemma 20 there must exist a k such that M_T entered block Constraint(j, k), but did not leave it. From the machine diagrams, M_T must have entered some block FinishSlice(j, j'), where j' is the slice M_T was processing when the counter reached zero. From the machine diagrams, M_T transitions to block Finish when it has located the object at time point b_j. Specifically, M_T is reading index $i+1$ in state qpf_{jj-1} when it sees ap_j. From the transition structure, M_T moves one position to the right and enters q'_f. M_T accepts W iff the input head is on the right endmarker. Therefore, M_T accepts W iff the length of W equals the error index.

The above cases are the only ones in which M_T accepts W. M_T accepts W iff it reaches either state q_e or state q_f while processing W. From the machine diagrams, M_T can only reach q_e if it fails to find an index satisfying some object b_j in P_1 after indices are found for the enabling objects of b_j. This case is covered above.

M_T can only reach state q_f through state q'_f. From the machine diagrams, M_T can only reach q'_f in one of the following cases:

1. It reaches some state qf_{ii}. This occurs only when a constraint fails on objects within P_1.
2. It is in state qp_i and finds letter a that is neither $ap_{i+1init}$ nor a_i, where p_i is a time point in

P_1.

3. It finds the index satisfying $b_{\max-1}$ of P_2 and there are no constraints relative to $p_{\max-1}$.

4. It finds the index satisfying $b_{\max-1}$ of P_2 and has successfully checked all constraints relative to $p_{\max-1}$.

Each of these cases is covered by the cases in the lemma, so the lemma holds.

□

The M_{TBAD} Construction Algorithm

M_{TBAD} has the same transitions as M_T with the exception that transitions to the final and garbage states from the processing states are swapped.

Algorithm 7 Let T be a temporally unambiguous, anchor-free timing diagram. Let *split* be the split point of T and let $p_{\max}$ be the maximal time point of T. Let M_{TBAD} be a 2DCM $\langle 1, K, \Sigma, \lhd, \rhd, \delta, q_{p_0}, \{q_f\}\rangle$ where Σ is the set of all assignments to the signals in T, δ contains the transitions generated below, and K consists of all states appearing in some transition of δ.

- Garbage state rule. The garbage state loops infinitely to reject strings.
  - $(q_g, x, x) \rightarrow (q_g, 0, 0)$
- Transitions to locate index satisfying time point p_{i+1}.
  - $(qp_i, ap_{i+1init}, x) \rightarrow (q2p_i, 1, 0)$
  - $(q2p_i, ap_{i+1}, x) \rightarrow (qp_{i+1}, 0, 0)$ if no constraints are relative to p_{i+1} and p_i is not $p_{\max-2}$
  - $(q2p_i, ap_{i+1}, x) \rightarrow (q'_f, 0, 0)$ if no constraints are relative to p_{i+1} and p_i is $p_{\max-2}$
  - $(q2p_i, ap_{i+1}, x) \rightarrow (qrp_ic_1, 0, 0)$ if there are constraints relative to p_{i+1}

- $(q2p_i, a, x) \rightarrow (qp_i, 0, 0)$ for all $a \neq ap_{i+1}$
- $(qp_i, a_i, x) \rightarrow (qp_i, 1, 0)$
- $(qp_i, a, x) \rightarrow (q_g, 1, 0)$ for all a that are neither $ap_{i+1init}$ nor a_i, if $i \leq$ split
- $(qp_i, a, x) \rightarrow (q_f, 1, 0)$ for all a that are neither $ap_{i+1init}$ nor a_i, if $i >$ split

- Transitions for encountering right endmarker, generated for all i such that p_i is a time point smaller than $p_{\max-2}$:
  - $(q_{p_i}, \rhd, x) \rightarrow (q_g, 0, 0)$ if $i \leq$ split
  - $(q_{p_i}, \rhd, x) \rightarrow (q_f, 0, 0)$ if $i >$ split

- Transitions to generate to check constraints relative to time point p_i (the rules introduced in this section must be generated once for each constraint c_k relative to p_i):

  If slice variable v_j appears n times on greater-than side of c_k

  - $(qp_i c_{kjInc}, a, x) \rightarrow (qp_i c_{kjInc1}, 0, 1)$
  - $(qp_i c_{kjInch}, a, x) \rightarrow (qp_i c_{kjInch+1}, 0, 1)$ for all $1 \leq h < n$
  - $(qp_i c_{kjIncn}, ap_{j+1init}, x) \rightarrow (q2p_i c_{kjInc}, 1, 0)$
  - $(q2p_i c_{kjInc}, ap_{j+1}, x) \rightarrow (qp_i c_{kj+1Inc}, 0, 0)$
  - $(q2p_i c_{kjInc}, a, x) \rightarrow (qp_i c_{kjInc}, 0, 0)$ for each $a \neq ap_{j+1}$
  - $(qp_i c_{kjIncn}, a, x) \rightarrow (qp_i c_{kjInc}, 1, 0)$ for each $a \neq ap_{j+1init}$

  If slice variable v_j does not appear on greater-than side of c_k and v_j is smaller than v_{gtmax}

  - $(qp_i c_{kjInc}, ap_{j+1init}, x) \rightarrow (q2p_i c_{kjInc}, 1, 0)$
  - $(q2p_i c_{kjInc}, ap_{j+1}, x) \rightarrow (qp_i c_{kj+1Inc}, 0, 0)$
  - $(q2p_i c_{kjInc}, a, x) \rightarrow (qp_i c_{kjInc}, 0, 0)$ for all $a \neq ap_{j+1}$
  - $(qp_i c_{kjInc}, a_j, x) \rightarrow (qp_i c_{kjInc}, 1, 0)$

  If v_j is larger than v_{gtmax} and no constant appears in c_k:

- $(qp_ic_{jkInc}, x, x) \rightarrow (qr2p_ic_k, 0, 0)$

If v_j is larger than v_{gtmax} and c_k is of the form $v_{a_1} + \ldots + v_{b_1} < v_{a_2} + \ldots + v_{b_2} + n$, where all v's are slice variables and n is a non-zero constant:

- $(qp_ic_{kjInc}, x, x) \rightarrow (qp_ic_{k\text{const}1}, 0, 1)$
- $(qp_ic_{k\text{const}h}, x, x) \rightarrow (qp_ic_{k\text{const}h+1}, 0, 1)$ for each $1 \leq h < n$.
- $(qp_ic_{k\text{const}n}, x, x) \rightarrow (qr2p_ic_k, 0, 0)$

If v_j is larger than v_{gtmax} and c_k is of the form $v_{a_1} + \ldots + v_{b_1} + n < v_{a_2} + \ldots + v_{b_2}$, where all v's are slice variables and n is a non-zero constant:

- $(qp_ic_{kjInc}, x, x) \rightarrow (qp_ic_{k\text{const}1}, 0, -1)$
- $(qp_ic_{k\text{const}h}, x, 1) \rightarrow (qp_ic_{k\text{const}h+1}, 0, -1)$ for each $1 \leq h < n$.
- $(qp_ic_{k\text{const}h}, x, 0) \rightarrow (q_g, 0, 0)$ for all $1 \leq h \leq n$, if $i \leq$ split
- $(qp_ic_{k\text{const}h}, x, 0) \rightarrow (q_f, 0, 0)$ for all $1 \leq h \leq n$, if $i >$ split
- $(qp_ic_{k\text{const}n}, x, 1) \rightarrow (qr2p_ic_k, 0, 0)$

Return to the beginning of the input to process less-than side

- $(qr2p_ic_k, \triangleleft, x) \rightarrow (qp_ic_{k0Dec}, 0, 0)$
- $(qr2p_ic_k, a, x) \rightarrow (qrp_ic_k, -1, 0)$ for each $a \neq \triangleleft$

If slice variable v_j appears on less-than side of c_k and $j \leq i$

- $(qp_ic_{kjDec}, a, 1) \rightarrow (qp_ic_{kjDec1}, 0, -1)$
- $(qp_ic_{kjDec}, a, 0) \rightarrow (q_g, 0, 0)$ if $i \leq$ split
- $(qp_ic_{kjDec}, a, 0) \rightarrow (q_f, 0, 0)$ if $i >$ split
- $(qp_ic_{kjDech}, a, 1) \rightarrow (qp_ic_{kjDech+1}, 0, -1)$ for all $1 \leq h < n$
- $(qp_ic_{kjDech}, a, 0) \rightarrow (q_g, 0, 0)$ for all $1 \leq h \leq n$ if $i \leq$ split
- $(qp_ic_{kjDech}, a, 0) \rightarrow (q_f, 0, 0)$ for all $1 \leq h \leq n$ if $i >$ split

- $(qp_ic_{kjDecn}, ap_{j+1init}, 1) \rightarrow (q2p_ic_{kjDec}, 1, 0)$
- $(q2p_ic_{kjDec}, ap_{j+1}, x) \rightarrow (qp_ic_{kj+1Dec}, 0, 0)$
- $(q2p_ic_{kjDec}, a, x) \rightarrow (qp_ic_{kjDec}, 0, 0)$ for all $a \neq ap_{j+1}$
- $(qp_ic_{kjDecn}, a, 1) \rightarrow (qp_ic_{kjDec}, 1, 0)$ for each $a \neq ap_{j+1init}$

If slice variable v_j does not appear on less-than side of c_k and $j \leq i$

- $(qp_ic_{kjDec}, ap_{j+1init}, x) \rightarrow (q2p_ic_{kjDec}, 1, 0)$
- $(q2p_ic_{kjDec}, ap_{j+1}, x) \rightarrow (qp_ic_{kj+1Dec}, 0, 0)$
- $(q2p_ic_{kjDec}, a, x) \rightarrow (qp_ic_{kjDec}, 0, 0)$ for all $a \neq ap_{j+1}$
- $(qp_ic_{kjDec}, a_j, x) \rightarrow (qp_ic_{kjDec}, 1, 0)$

If $j = i + 1$ and c_k is the last constraint to be checked for p_{i+1}

- $(qp_ic_{kjDec}, x, 0) \rightarrow (q_g, 0, 0)$ if $i \leq$ split
- $(qp_ic_{kjDec}, x, 0) \rightarrow (q_f, 0, 0)$ if $i >$ split
- $(qp_ic_{kjDec}, x, 1) \rightarrow (qp_{i+1}, 0, 0)$ if p_i is not $p_{\max-2}$
- $(qp_ic_{kjDec}, x, 1) \rightarrow (q_gi, 0, 0)$ if p_i is $p_{\max-2}$

If $j = i + 1$ and c_k is not the last constraint to be checked for p_{i+1}

- $(qp_ic_{kjDec}, x, 1) \rightarrow (qrp_ic_{k+1}, -1, 0)$
- $(qp_ic_{kjDec}, x, 0) \rightarrow (q_g, 0, 0)$ if $i \leq$ split
- $(qp_ic_{kjDec}, x, 0) \rightarrow (q_f, 0, 0)$ if $i >$ split

Transitions to start checking next constraint associated with p_{i+1}

- $(qrp_ic_k, \lhd, x) \rightarrow (qrp_ic_{k\text{deplete}}, 0, 0)$
- $(qrp_ic_k, a, x) \rightarrow (qrp_ic_k, -1, 0)$ for all $a \neq \lhd$
- $(qrp_ic_{k\text{deplete}}, x, 1) \rightarrow (qrp_ic_{k\text{deplete}}, 0, -1)$
- $(qrp_ic_{k\text{deplete}}, x, 0) \rightarrow (qp_ic_{k0Inc}, 0, 0)$

Correctness

Given the structural similarities between M_{TBAD} and M_T, several lemmas pertaining to M_T apply to M_{TBAD}; in particular, Lemmas 14 through 21 all hold of M_{TBAD}.

Theorem 12 *Let T be a temporally unambiguous timing diagram and let W be a word. M_{TBAD} accepts W iff the next object-index match for T, W, 1, and P_1 is defined for all objects in P_1, but the next object-index match for T, W, 1, and $P_1 \cup P_2$ is undefined for some object in P_2.*

Proof M_{TBAD} accepts W iff M_{TBAD} reaches state q_f. The transition structure of M_{TBAD} indicates that it cannot accept a word unless it reaches state $q'p_0$. By Lemma 21, M_{TBAD} accepts W only if the next object-index match for T, W, 1, and P_1 is defined for all objects in P_1.

Assume M_{TBAD} accepts W. Let I' be the next object-index match for T, W, 1, and $P_1 \cup P_2$. I' must be defined for all objects in P_1 since I is defined for all objects in P_1 and M_{TBAD} uses the same transition structure and the same starting point to locate matches for objects in P_1 and P_2. M_{TBAD} reaches q_f iff one of the following cases holds:

1. M_{TBAD} is in state $q'p_i$ and reads a letter that violates the signal patterns of time slice (p_i, p_{i+1}). Since I' is defined for all objects in P_1, object b_i must lie in P_2. M_{TBAD} cannot reach state $q'p_{i+1}$ in this case, so I' must be undefined for object b_{i+1} by Lemma 15. Object b_{i+1} must lie in P_2 since P_1 and P_2 form a valid assume-guarantee partitioning for T (Defn. 65).

2. M_{TBAD} is in state $q'p_i$ and reads the right endmarker, where p_i is smaller than $p_{\max-1}$. M_{TBAD} therefore does not reach state $q'p_{i+1}$ and the argument is the same as in the previous case.

3. The counter reaches zero while M_{TBAD} is processing constraint c_k for time point p_i. All constraints relative to time points in P_1 are satisfied or M_{TBAD} would not have reached $q'p_0$ by Lemma 20. Time point p_i therefore lies in P_2. By Lemma 19, some constraint relative to

p_i in T is not satisfied. By the definition of the next object-index match (Defn. 66), I' cannot be defined for b_i.

Assume the next object-index match for T, W, 1, and P_1 is defined for all objects in P_1, but the next object-index match for T, W, 1, and $P_1 \cup P_2$ is undefined for some object b_j in P_2. Then one of the following cases must hold:

1. There does not exist an index larger than $I'(b_{j-1})$ that satisfies b_j. Then M_{TBAD} would have encountered the right endmarker in state $q'p_{j-1}$, which results in a transition to q_f.
2. There exists an index i larger than $I(b_{j-1})$ that satisfies b_j, but there also exists index i' larger than $I(b_{j-1})$ and smaller than i that violates the signal patterns of time slice (p_{j-1}, p_j). By Lemma 12, letters a_{j-1} and ap_{jinit} capture the signal patterns of time slice (p_{j-1}, p_j). M_{TBAD} transitions to q_f upon reading any other letter in $q'p_{j-1}$, so it reaches q_f when at index i'.
3. There exists an index i larger that $I(b_{j-1})$ that satisfies b_j, all indices from $I(b_{j-1})$ through i satisfy the signal patterns of time slice (p_{j-1}, p_j), but some constraint relative to p_j is not satisfied. By Lemma 10, some constraint c_k relative to p_j is not satisfied. M_{TBAD} therefore does not leave some Constraint$(j-1, k')$ by Lemma 20 (k' and k are not necessarily the same, since M_{TBAD} may fail to satisfy c'_k, also relative to p_j, before it tests c_k). The transition structure shows that M_{TBAD} can only fail to leave Constraint$(j-1, k')$ if it reaches q_f.

□

Decidability of the Regular Containment Problem

In deciding the regular containment problem, a finite automaton language and a timing diagram language must be intersected . As the intersection of two 2DCM is effectively constructable [48], the first step is to produce a 2DCM from a regular automaton. This is a straightforward construction; the transitions of the finite automaton are merely augmented with a counter whose value is never changed.

Theorem 13 *Let A be a deterministic finite automaton that accepts by final state. There exists a deterministic 1-counter machine that accepts the same language as A.*

Proof Let $A = \langle K, \Sigma, \delta_A, q_0, F\rangle$ be a deterministic finite automaton. Define

$$M = \langle 1, K, \Sigma, \lhd, \rhd, \delta_M, q_0, F\rangle$$

where for each transition $(q_1, a) \to q_2$ in δ_A, transition $(q_1, a, 0) \to (q_2, 1, 0)$ is in δ_M and δ_M contains no additional transitions. Let x be in Σ^*. It must be established that x is accepted by A iff x is accepted by M.

By definition [45], a string x is accepted by A iff $\delta_A(q_0, x) = q_f$ for some $q_f \in F$. The extension of δ_A to strings is defined inductively as $\delta_A(q, \epsilon) = q$, where ϵ represents the empty string and $\delta_A(q, wa) = \delta_A(\delta_A(q, w), a)$. The sequence of states $q_0, q_1, \ldots, q_f$ A takes on x can be extracted from the definition of δ_A where $(q_i, x_i + 1) = q_{i+1}$, x_i is the letter in the i^{th} position of x (counting first position as 1, not 0), and $q_f \in F$. Since A is assumed to be deterministic, this sequence is unique.

Recalling from Definition 94, a string $x \in \Sigma^*$ is *accepted* by M if $(q_0, \lhd x \rhd, 1, 0, \ldots, 0) \Rightarrow^* (q, \lhd x \rhd, i, c_1, \ldots, c_k)$ for some $q \in F$, $1 \leq i \leq |\lhd x \rhd|$, and non-negative integers $c_1, \ldots, c_k$.

Assume x is accepted by A. Let $q_0, \ldots, q_f$ be the state sequence extracted from A for x. The starting configuration of M is $(q_0, \lhd x \rhd, 1, 0)$ by Definition 94. For all i, if A contained transition $(q_1, a) \to q_2$ in δ_A, then δ_M contains transition $(q_1, a, 0) \to (q_2, 1, 0)$. M must therefore have configuration chain

$$(q_0, \lhd x \rhd, 1, 0) \Rightarrow^* (q_f, \lhd x \rhd, i, 0)$$

where i is the length of x. q_f is in F by assumption, so x is accepted by M.

Assume x is accepted by M. Then by Definition 94, M must have configuration chain

$$(q_0, \lhd x \rhd, 1, 0) \Rightarrow^* (q_f, \lhd x \rhd, i, 0)$$

where q_f is in F. Since all transitions in δ_M advance the counter head by exactly one position by definition, i must be the length of x. Consider transition $(q_j, \lhd x \rhd, j+1, 0) \Rightarrow (q_{j+1}, \lhd x \rhd, j+2, 0)$. By definition, this transition is only in δ_M if transition $(q_j, a) \rightarrow q_{j+1}$ is in δ_A, where a is the i^{th} letter in x, so $\delta_A(q_0, x) = q_f$, so x is accepted by A.

□

In the following decision procedures, whenever a finite automaton is intersected with a 2DCM, the automata is assumed converted to a 2DCM by the above procedure before the intersection occurs.

A 2DCM must have bounded reversals if containment on it is to be decidable. This lemma establishes that both M_T and M_{TBAD} have a bounded number of reversals, and that number is dependent upon T.

Lemma 22 *Let T be a timing diagram. The number of reversals made in either M_T or M_{TBAD} is bounded by a finite number based on T.*

Proof In the worst case, M_T makes two reversals for every constraint in T while processing P_1, two reversals for every constraint in T while processing all of T, and two reversals to get from processing P_1 to processing all of T. The same holds for M_{TBAD}. Showing that the number of constraints on T is bounded by a finite number is sufficient. The number of time-bound objects in T is finite. Each step in Algorithm 5 iterates a finite number of times and adds a finite number of expressions to the set S. The number of constraints on T is therefore finite and could be computed from the number of time-bound objects in T.

□

Both decision procedures alter the start and final states of an automaton. Given automaton A, the notation $A|_{(q,q')}$ denotes the version of A with start state q and single final state q'. The next two sections establish the decidability results for the iterative and invariant semantics, respectively.

Iterative Decidability

The iterative decidability test was described at the beginning of the section. Given a timing diagram T and a finite automaton A, relations Avoid and Reach are computed on the states of A. A state is in Avoid if a word in M_{TBAD} is accepted from it, while a pair of states are in Reach if a word in M_{T_F} is accepted from one to the other.

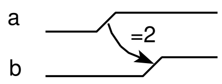
Algorithm 8 Let $A = \langle K_A, \Sigma, \delta_A, q_0, F\rangle$ be a finite automaton. Let T be a timing diagram and let P_1 and P_2 be a valid assume-guarantee partitioning for T. Let M_T be the machine constructed by Algorithm 6 for T and let M_{TBAD} be the machine constructed by Algorithm 7 for T. Let M_{T_F} be the restriction of M_T to having only state q_f, rather than both q_f and q_e, as a final state.

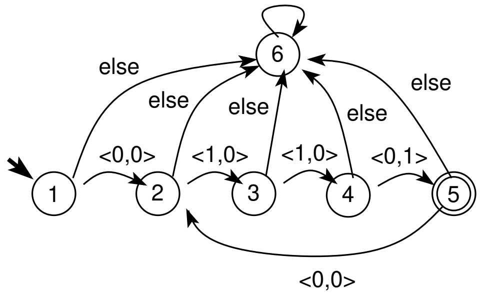
1. For each pair of states (q, q_f) in A such that $q_f \in F$, test whether A with start state q and final state q_f can generate a word in M_{TBAD}. Let Avoid be the set of states q such that $A|_{(q,q_f)} \cap M_{TBAD} \neq \emptyset$.

2. For each pair of states (q, q') in A, test whether A with start state q and final state q' can generate a word in M_{T_F}. Define predicate Reach to contain all pairs of states (q, q') such that $A|_{(q,q')} \cap M_{T_F} \neq \emptyset$. Let $\mathcal{CR}$ denote the transitive closure of Reach.

3. If there exists a state q in Avoid such that $(q_0, q) \in \mathcal{CR}$, $\mathcal{L}(A) \not\subseteq \mathcal{L}(M)^*$. If no such state exists, $\mathcal{L}(A) \subseteq \mathcal{L}(M)^*$.

Example 34 Suppose timing diagram T is interpreted iteratively under strict level objects with P_1 empty.





The language of T is

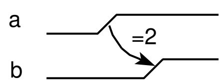
$$\left(\left(\langle 0,1\rangle+\langle 1,0\rangle+\langle 1,1\rangle\right)^{*}\ \langle 0,0\rangle^{+}\ \langle 1,0\rangle\ \langle 1,0\rangle\ \left(\langle 0,1\rangle+\langle 1,1\rangle\right)\right)^{*}$$

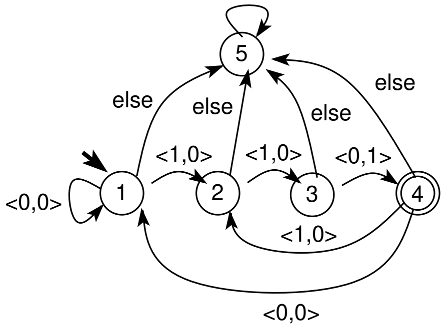
where each pair $\langle x,y\rangle$ denotes that $a=x$ and $b=y$. The language of finite automaton A is

$$\left(\langle 0,0\rangle\ \langle 1,0\rangle\ \langle 1,0\rangle\ \langle 0,1\rangle\right)^{*}$$

which is contained in the language of T. Algorithm 8 constructs $\mathsf{Avoid}=\{2,3,4\}$ and $\mathsf{Reach}\{(1,5),(5,5),(1,6),(2,6),(3,6),(4,6),(5,6),(6,6)\}$. The only states accessible from the start state (1) in the closure of Reach are 5 and 6. As neither of these is in Avoid, the algorithm determines that $\mathcal{L}(A)\subseteq\mathcal{L}(T)$.

Example 35 The language of timing diagram T is as in Example 34.





The language of finite automaton A is now

$$(\langle 0,0\rangle^* \langle 1,0\rangle \langle 1,0\rangle \langle 0,1\rangle)^*$$

which is not contained in the language of T. Algorithm 8 constructs $\mathsf{Avoid} = \{1,2,3,4\}$ and $\mathsf{Reach}\{(1,4),(4,4),(1,5),(2,5),(3,5),(4,5),(5,5)\}$. The start state is listed in Avoid, so the algorithm determines that $\mathcal{L}(A) \not\subseteq \mathcal{L}(T)$.

These last results establish that the above algorithm provides a decision procedure for the regular containment problem for iteratively interpreted timing diagrams. The following lemma establishes that the division of a given word into subwords with respect to the iterative description test is unique; this result is needed in the proof of the subsequent theorem. The first theorem establishes that the algorithm correctly determines containment. The second theorem establishes that all steps of the algorithm are decidable.

Lemma 23 *Let W be a word that iteratively describes timing diagram T. W has a unique decomposition $w_1 w_2 \cdots w_n$ such that the first index in each w_i for $i > 1$ is*

1. *one larger than the largest index used in the next object-index match for T, W, and the first index of w_{i-1} if this object-index match is defined for all objects in T*
2. *the error index (Defn. 66) of the next object-index match for T, W, and the first index of w_{i-1} if this object-index match is undefined for some object in P_1.*

Proof By the definition of iterative description (Defn. 68), the first object-index match is sought from the first index of W. The definition further shows that the decision as to where to begin the next search is deterministic, based upon whether or not the current search matched all objects in P_1. The indices determining the boundaries between each w_i and w_{i+1} are precisely those used to begin the next search in Definition 68, so the lemma holds.

□

Theorem 14 *Let $A = \langle K_A, \Sigma, \delta_A, q_0, F\rangle$ be a finite automaton. Let T be a timing diagram and P_1 and P_2 be a valid assume-guarantee partitioning for T. Algorithm 8 correctly determines whether $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Iter}}$.*

Proof

1. Assume $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Iter}}$. Every word accepted by A has a unique decomposition into $w_1 w_2 \cdots w_n$ as described in Lemma 23. By Theorem 11, each w_i plus the first character of w_{i+1} is accepted by M_T if $i < n$ and w_n is accepted by M_T. Each word accepted by A therefore generates a chain of states that lies in Reach. Theorems 11 and 12 indicate that no word is accepted by both M_T and M_{TBAD}. Hence there cannot exist state q such that (q_0, q) in $\mathcal{CR}$ and q is in Avoid because the decomposition of each word is unique and generates chains of states entirely in Reach. The algorithm therefore correctly determines that $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Iter}}$.

2. Assume $\mathcal{L}(A) \not\subseteq \mathcal{L}(T)^{\mathrm{Iter}}$. Then A must accept some word W such that one of the next object-index matches for P_1 located by the definition of iterative description could not be extended to the next object-index match for T. Let i be the index at which the search for this next object-index match started. Lemma 12 shows that W^i is accepted by M_{TBAD}. The state q_i that A was in when it processed the i^{th} index of W is therefore in Avoid. Given any word in W, the choice of where to begin the search for each successive object-index match is deterministic. All searches prior to that starting in index i must have succeeded, or the search from i would not have been attempted. In addition, none of the previous searches failed due to reaching the end of the input. Since M_T only enters final state q_e when it encounters the end of the input, all other accepting runs of M_T must end in final state q_f.

   The portion of W up to index i iteratively describes T. By Lemma 23, it has a unique decomposition as described in the lemma. By Theorem 11, each subword in this decomposition is accepted by M_{T_F}. The respective start and final states for each subword are therefore related in Reach, so there must be a path in $\mathcal{CR}$ from the start state to q_i; the algorithm therefore correctly determines that $\mathcal{L}(A) \not\subseteq \mathcal{L}(T)^{\mathrm{Iter}}$.

□

Theorem 15 *Algorithm 8 provides a decision procedure for the problem $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Iter}}$, where A is a finite automaton accepting by final state and T is a timing diagram.*

Proof Theorem 14 proves that Algorithm 8 correctly determines whether $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Iter}}$; it remains to show that every step of the algorithm is decidable. M_T and M_{TBAD} are constructable by Algorithms 6 and 7, respectively, and each has bounded reversals by Theorem 22. The counter machine corresponding to A is constructable by Theorem 13; the construction algorithm shows that this machine has no reversals. The intersection of two reversal-bounded, 1-counter machines is effectively constructable and is also a reversal-bounded, 1-counter machine by Theorem 10. The emptiness tests on the intersection machines are decidable by Theorem 10. The number of states in A is finite by definition, so Reach must be finite. Closure computation is decidable. Algorithm 8 is therefore a decision procedure.

□

Invariant Decidability

The algorithm for invariant decidability is as described at the beginning of this section: each state of A is checked for generating a word in M_{TBAD}. If such a state is found, it is checked for reachability in A.

Algorithm 9 Let $A = \langle K_A, \Sigma, \delta_A, q_0, F \rangle$ be a finite automaton. Let T be a timing diagram and P_1 and P_2 be a valid assume-guarantee partitioning for T. Let M_{TBAD} be the machine constructed by Algorithm 7 for T.

1. For each pair of states (q, q_f) in A such that $q_f \in F$, test whether A with start state q and final state q_f can generate a word in M_{TBAD}. Let Avoid be the set of states q such that $A|_{(q,q_f)} \cap M_{TBAD} \neq \emptyset$.

2. Check whether any state in Avoid is reachable from the start state in A. If so, $\mathcal{L}(A) \not\subseteq \mathcal{L}(M)^{\mathrm{Inv}}$. Otherwise, $\mathcal{L}(A) \subseteq \mathcal{L}(M)^{\mathrm{Inv}}$.

These two theorems establish that the above algorithm provides a decision procedure for the regular containment problem for invariantly interpreted timing diagrams. The first theorem establishes that the algorithm correctly determines containment. The second theorem establishes that all steps of the algorithm are decidable.

Theorem 16 *Let $A = \langle K_A, \Sigma, \delta_A, q_0, F \rangle$ be a finite automaton. Let T be a timing diagram and P_1 and P_2 be a valid assume-guarantee partitioning for T. Algorithm 9 correctly determines whether $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Inv}}$.*

Proof

1. Assume $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Inv}}$. Then every word accepted by A invariantly describes T. By the definition of invariant description (Defn. 67), every suffix of every word accepted by A invariantly describes T. Hence there does not exist a state of A reachable from the start state from which a word can be generated such that the next object-index match for P_1 is defined for all objects in P_1, but for which the next object-index match for $P_1 \cup P_2$ is undefined for some object in P_2. By Theorem 12, any states in Avoid cannot be reachable from the start state of A, so Algorithm 9 correctly determines that containment holds.

2. Assume $\mathcal{L}(A) \not\subseteq \mathcal{L}(M)^{\mathrm{Inv}}$. Then there exists some word W that is accepted by A, but that does not invariantly describe T. By Definition 67, there exists some index i into W from which the next object-index match for P_1 is defined for all objects in P_1, but for which the next object-index match for $P_1 \cup P_2$ is undefined for some object in P_2. Let W^i denote the suffix of W starting with index i. By Theorem 12, W^i is accepted by M_{TBAD}. Let q_i be the state A was in when it processed index i. State q_i must be in Avoid by construction (Algorithm 9, Step 1). The existence of W proves that q_i is reachable from the start state of A, so Algorithm 9 correctly determines that containment fails.

□

Theorem 17 *Algorithm 9 provides a decision procedure for the problem $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Inv}}$, where A is a finite automaton accepting by final state and T is a timing diagram.*

Proof Theorem 16 proves that Algorithm 9 correctly determines whether $\mathcal{L}(A) \subseteq \mathcal{L}(T)^{\mathrm{Inv}}$; it remains to show that every step of the algorithm is decidable. M_{TBAD} is constructable by Algorithm 7 and has a bounded number of reversals by Theorem 22. The counter machine corresponding to A is constructable by Theorem 13; the construction algorithm shows that this machine has no reversals. The intersection of two reversal-bounded, 1-counter machines is effectively constructable and is also a reversal-bounded, 1-counter machine by Theorem 10. The emptiness test on the intersection machine is decidable by Theorem 10. State reachability is decidable for finite automata. Algorithm 9 is therefore a decision procedure.

□

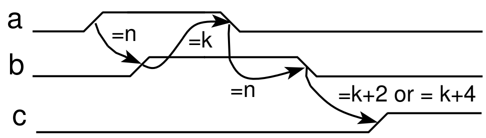
3.3.3 The Timing Diagram Containment Problem

Unfortunately, the same approach does not decide the timing diagram containment problem. The above decision procedures rely on there being only finitely many states in a finite automaton. A 2DCM does not have a finite number of states due to the possible values in the counters. The above approach might indicate when containment failed to hold, but it could not establish containment since an enumeration of the states is impossible.

It is possible to build a 2DCM that accepts the language of any temporally unambiguous timing diagram (invariant or iterative). Intuitively, the machine is a straightforward modification of M_T that begins a subsequent check in the appropriate index into the word after the previous check has been completed. Unfortunately, this machine is not guaranteed to have bounded reversals since the number of possible checks to be made on a word is bounded only by the size of the word. In addition, the machines for diagrams using non-strict interpretations may be forced to use two counters, one to check constraints as in M_T and one to store the index at which the current check started. This arises because non-strictly interpreted timing diagrams cannot necessarily be scanned backwards to find the start of the current search, while strictly interpreted timing diagrams can. As the results of Theorem 10 do not apply to machines with either unbounded reversals or more than one counter, the methods of the previous section cannot be used to provide a decision procedure here.

Certain instances of the timing diagram containment problem should be decidable. For example, if timing diagram T_1 corresponds to a regular language, $T_1 \subseteq T_2$ should be decidable since the regular containment problem is decidable. A decision procedure for these instances would require an algorithm for constructing a finite-state automaton from a timing diagram. Similar algorithms exist from several temporal logics into automata [11] [84] [85], so such a construction seems feasible.

The timing diagram containment problem is still open when T_1 is not regular. There exist 1-counter non-reversal-bounded languages that cannot be captured in any timing diagram. Consider "timing diagram"



Timing diagram is used in quotation marks above because this diagram is not well-formed due to the invalid time-bound expression on the last successor edge object. Assume for the moment that this were a well-formed timing diagram. Then a 1-2DCM could be constructed to accept its language using a similar approach to the one described for well-formed timing diagrams. As argued above, this machine would not have bounded reversals; the alternation of the variables on the successor edge objects would force reversals in a 1-2DCM accepting this language. There does not exist a single timing diagram that has the same language as this "timing diagram" because any valid time-bound on the last successor edge object would have to include the invalid value $k+3$. This diagram therefore provides an example of a 1-2DCM that does not have bounded reversals and is not representable as a timing diagram. Therefore, the undecidability of containment for 1-counter non-reversal bounded languages does not prove the undecidability of containment for timing diagram languages.

4

Rules of Inference

Rules of inference provide a syntactic view of consequence; their structure is therefore similar to that of semantic consequence. Inference rules are defined from a (possibly empty) set of formulas $F_1, \ldots, F_k$ to a single formula f and are denoted as either

$$F_1, \ldots, F_k \vdash f \qquad \text{or} \qquad \frac{F_1, \ldots, F_k}{f}$$

An inference rule is *sound* relative to a semantic relationship $\models$ iff it is supported by $\models$: that is, iff $F_1, \ldots, F_k \vdash f$ implies $F_1, \ldots, F_k \models f$. A set of inference rules is *complete* relative to a semantic relationship $\models$ iff $F_1, \ldots, F_k \models f$ implies $F_1, \ldots, F_k \vdash f$.

Each diagrammatic representation has multiple semantic relationships: $\models_b$ and $\models_s$ for circuit diagrams, $\models_b$, $\models_s$, and $\models_{\mathrm{fs}}$ for ASM charts, and $\models_{\mathrm{Inv}}$ and $\models_{\mathrm{Iter}}$ for timing diagrams. Each inference rule is defined relative to certain relationships; those relationships are tested in the corresponding soundness proofs. The soundness proofs all share a similar structure: any model described by all of the F_i's is shown to be described by f. Completeness is not addressed for HHL, although portions of the logic, such as circuit diagrams, are expected to be complete [32].

Inference rules are either *primitive* or *derived*. Primitive rules provide a basis for the inference system. Derived rules are shorthand for sequences of primitive rules. This chapter presents primitive rules, unless otherwise noted. The soundness of a derived rule follows from the soundness of its associated primitive rules.

HHL contains several general types of inference rules: *Observe* rules transfer information from a diagrammatic representation to a sentential representation, *Apply* rules transfer information from a sentential representation to a diagrammatic representation, *Algebraic* rules reflect equivalence-preserving transformations within a single representation, and *Methods of Proof* are templates for reasoning steps that apply to any representation. *Observe*, *Apply*, and the methods of proof are based on similar rules in *Hyperproof* [7]. Algebraic rules are standard in most logical calculi. The remaining rules in HHL are particular to the hardware domain, and operate over multiple syntactic representations.

The rules are presented in several sections, grouped by the representations they relate. Each presentation contains an informal description of the rule, sample instances of the rule, and its formal, syntactic-level definition. The definitions use the term *obtained from*; when object A is obtained from object B, A is initialized to be identical to B, then the outlined series of steps are applied to transform A into the desired object.

4.1 Circuit Diagram to Circuit Diagram

The rules in this section obtain one circuit diagram from another through steps such as "adding a gate" and "removing a gate". In the case of the latter, the removal of a gate may leave some wires hanging unattached. Rather than add the fine details of cleaning up dangling wires in the individual rules, the following "cleanup step" is assumed applied after every inference rule in this section:

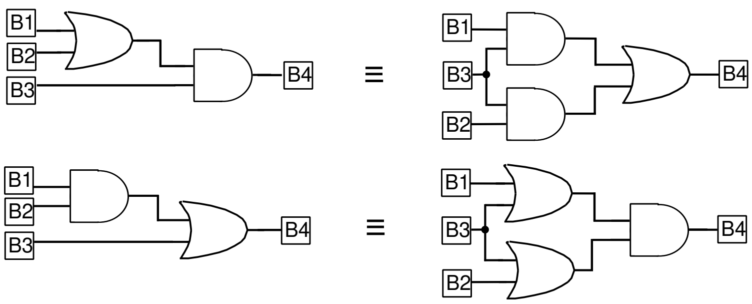
Definition 96 Let C be a circuit diagram. For any wire $(p_1, p_2) \in C$, if either p_1 is not the right element of some component in C or p_2 is not the left element of some component in C, remove (w_1, w_2) from C. For any component with port $p \in O$ such that p is not the left endpoint of some wire in C, remove the component from C. For any component with port $p \in I$ such that p is not the right endpoint of some wire in C, remove the component from C. The iteration of these steps applied to C is called the *cleanup step*.

The circuit diagram inference rules are all algebraic. They are presented as equivalences of the form $F_1 \equiv F_2$; this notation is shorthand for $F_1 \vdash F_2$ and $F_2 \vdash F_1$. The first four rules correspond to equivalences of boolean algebra: distributivity, commutativity, identity, and excluded middle. All other rules corresponding to boolean equivalences are derivable from this set [40]. The circuit diagram inference rules are all defined relative to the behavioral implementation relationship, $\models_b$ (Defn. 28).

Rule 1 The distributivity rules correspond to boolean equivalences

$$(a \vee b) \wedge c \equiv (a \wedge c) \vee (b \wedge c)$$

$$(a \wedge b) \vee c) \equiv (a \vee c) \wedge (b \vee c)$$



Circuit diagram C' is derivable from circuit diagram C by *distributivity* iff one of these cases holds:

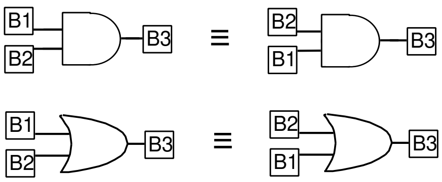
1. C contains $r = \mathsf{OrIcon}(p_1, p_2, p_3)$, $a = \mathsf{AndIcon}(p_4, p_5, p_6)$, and a branch-free wire (p_3, p_4) and C' is obtained from C by removing a and r, adding $\mathsf{OrIcon}(p_4, p_5, p_6)$, $\mathsf{AndIcon}(p_1, p_2, p_3)$, and $\mathsf{AndIcon}(p_7, p_8, p_9)$ to C', where p_7, p_8, and p_9 are new connection points, adding wire (p_a, p_7), where (p_a, p_5) is a wire in C, adding wire (p_b, p_8), where (p_b, p_2) is a wire in C, removing wire (p_a, p_5), and adding wire (p_9, p_5).

2. C contains $a = \mathsf{AndIcon}(p_1, p_2, p_3)$, $r = \mathsf{OrIcon}(p_4, p_5, p_6)$, and a branch-free wire (p_3, p_4) and C' is obtained from C by removing a and r, adding $\mathsf{AndIcon}(p_4, p_5, p_6)$, $\mathsf{OrIcon}(p_1, p_2, p_3)$, and $\mathsf{OrIcon}(p_7, p_8, p_9)$ to C' where p_7, p_8, and p_9 are new connection points, adding wire (p_a, p_7), where (p_a, p_5) is a wire in C, adding wire (p_b, p_8), where (p_b, p_2) is a wire in C, removing wire (p_a, p_5), and adding wire (p_9, p_5).

Rule 2 The commutativity rules correspond to boolean equivalences

$$a \wedge b \equiv b \wedge a$$

$$a \vee b \equiv b \vee a$$



Circuit diagram C' is derivable from circuit diagram C by *commutativity* iff one of these cases holds:

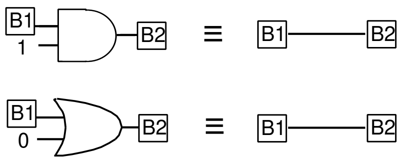
1. C contains $\mathsf{OrIcon}(p_1, p_2, p_3)$ and C' is obtained from C by replacing each wire (p_a, p_1) with a wire (p_a, p_2) and by replacing each wire (p_b, p_2) that is also in C with a wire (p_b, p_1).

2. C contains $\mathsf{AndIcon}(p_1, p_2, p_3)$ and C' is obtained from C by replacing each wire (p_a, p_1) with a wire (p_a, p_2) and by replacing each wire (p_b, p_2) that is also in C with a wire (p_b, p_1).

Rule 3 The identity rules correspond to boolean equivalences

$$a \wedge \mathsf{true} \equiv a$$

$$a \vee \mathsf{false} \equiv a$$



Circuit diagram C' is derivable from circuit diagram C by *identity* iff one of these cases holds:

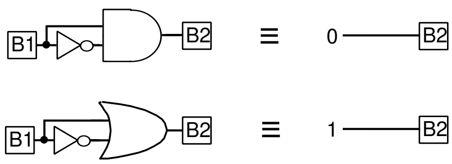
1. C contains r=$\mathsf{OrIcon}(p_1, p_2, p_3)$ with p_2 wired to ground and C' is obtained from C by removing r, replacing all occurrences of p_3 with p_a, where (p_a, p_1) is a wire in C, and removing (p_a, p_1).

2. C contains $a = \mathsf{AndIcon}(p_1, p_2, p_3)$ with p_2 wired to ground and C' is obtained from C by removing a, replacing all occurrences of p_3 with p_a, where (p_a, p_1) is a wire in C, and removing (p_a, p_1).

Rule 4 The excluded middle rules correspond to boolean equivalences

$$a \wedge \neg a \equiv \mathsf{false}$$

$$a \vee \neg a \equiv \mathsf{true}$$

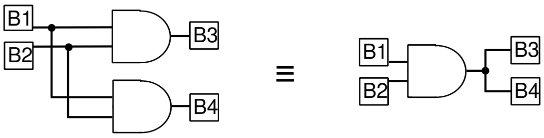


Circuit diagram C' is derivable from circuit diagram C by *excluded middle* iff one of these cases holds:

1. C contains $i = \mathsf{InverterIcon}(p_1, p_2)$, $r = \mathsf{OrIcon}(p_3, p_4, p_5)$, branch-free wire (p_2, p_4), and wires (p_a, p_3) and (p_a, p_1), and C' is obtained from C by removing r and i and adding a wire (p_p, p_b) for each p_b such that (p_5, p_b) is in C, where p_p is the power point.

2. C contains $i = \mathsf{InverterIcon}(p_1, p_2)$, $a = \mathsf{AndIcon}(p_3, p_4, p_5)$, branch-free wire (p_2, p_4), and wires (p_a, p_3) and (p_a, p_1), and C' is obtained from C by removing a and i and adding a wire (p_g, p_b) for each p_b such that (p_5, p_b) is in C, where p_g is the ground point.

Rule 5 The branching rules are used to move the branching points of wires through gates; points may be moved forwards or backwards through gates.

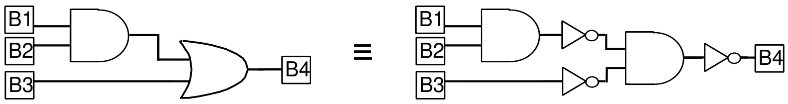


Circuit diagram C' is derivable from circuit diagram C by *branching* iff one of the following cases holds:

1. C contains wires (p_a, p_b) and (p_a, p_c) where p_a is an internal output port for a component M with one output interface port and C' is obtained from C by adding a copy M' of M, removing wire (p_a, p_c), adding wire (p_d, p_c) where p_d is the output interface port of M', and for each wire (p_1, p_2) where p_2 is an input interface port to M adding wire (p_1, p_2') where p_2' is the corresponding input interface port to M'.

2. C contains components M_1 and M_2 such that for every input interface port p_1 of M_1 and its corresponding input interface port p_2 of M_2, there exist wires (p_a, p_1) and (p_a, p_2) in C and C' is obtained from C by removing M_2 and for each output interface port p_b of M_1, adding wire (p_b, p_c), where p_c was wired to the output interface port in M_2 corresponding to p_b in C.

Rule 6 The inlining rules allow the icon for a component to be replaced with the body of that component. These are substitution rules for circuit diagrams. The following example uses OrIcon (Page 7).



1. Circuit diagram C' is derivable from circuit diagram C by *component inlining* iff C contains some non-basic circuit icon C_1 and C' is obtained from C by removing the icon and replacing it with the circuit diagram for C_1.

2. Circuit diagram C' is derivable from circuit diagram C by *component abstraction* iff C contains a subdiagram isomorphic to the body of circuit sketch C_1 and C' is obtained from C by removing the subdiagram and replacing it with the icon mapping to C_1.

4.1.1 Soundness

Based on Lemma 3, these rules are sound if the circuit diagrams used in the rules are behaviorally equivalent. These soundness proofs establish behavioral equivalence by showing that the equations relating ports in assignments are the same for the outputs of each diagram in the rule. The equations used in the proofs are obtained from the formal statements of the rules given above.

Definition 2 gives the output equations for AndGates, Inverters, and UnitDelays. The output equation corresponding to an OrGate is derived from these and the definition of the or gate on page 38; portions of assignments corresponding to OrIcons must reflect $i(p_3) = i(p_1) + i(p_2)$, where p_3 maps to the output port of the OrIcon, p_1 and p_2 map to the input ports of the OrIcon, and i is the assignment from ports to values in the device. In each soundness proof, the b_i's refer to the ports pictured in the instances of the corresponding rule.

Lemma 24 *The distributivity rule is sound relative to* $\models_b$.

Proof The two cases of the rule are considered in turn.

1. And-distributivity: Assume C' is obtained from C by the first case of distributivity. The output equation for C and its derivation are on the left; the output equation for C' and its derivation are on the right.

$$i(b_4) = i(b_3) * (i(b_1) + i(b_2))$$

$$i(p_6) = i(p_4) * i(p_5)$$
$$i(p_3) = i(p_1) + i(p_2)$$
$$i(p_4) = i(p_3)$$
$$i(b_1) = i(p_1)$$
$$i(b_2) = i(p_2)$$
$$i(b_3) = i(p_5)$$
$$i(b_4) = i(p_6)$$

$$i(b_4) = (i(b_1) * i(b_3)) + (i(b_3) * i(b_2))$$

$$i(b_1) = i(p_1)$$
$$i(b_3) = i(p_2)$$
$$i(b_3) = i(p_7)$$
$$i(b_2) = i(p_8)$$
$$i(b_4) = i(p_6)$$
$$i(p_3) = i(p_1) * i(p_2)$$
$$i(p_9) = i(p_7) * i(p_8)$$
$$i(p_6) = i(p_4) + i(p_5)$$
$$i(p_4) = i(p_3)$$
$$i(p_5) = i(p_9)$$

2. Or-distributivity: Assume C' is obtained from C by the second case of distributivity. The output equation for C and its derivation are on the left; the output equation for C' and its derivation are on the right.

$$i(b_4) = (i(b_1) + i(b_3)) * (i(b_3) + i(b_2))$$

$$i(b_4) = i(b_3) + (i(b_1) * i(b_2))$$

$$i(p_6) = i(p_4) + i(p_5)$$
$$i(p_3) = i(p_1) * i(p_2)$$
$$i(p_4) = i(p_3)$$
$$i(b_1) = i(p_1)$$
$$i(b_2) = i(p_2)$$
$$i(b_3) = i(p_5)$$
$$i(b_4) = i(p_6)$$

$$i(b_1) = i(p_1)$$
$$i(b_3) = i(p_2)$$
$$i(b_3) = i(p_7)$$
$$i(b_2) = i(p_8)$$
$$i(b_4) = i(p_6)$$
$$i(p_3) = i(p_1) + i(p_2)$$
$$i(p_9) = i(p_7) + i(p_8)$$
$$i(p_6) = i(p_4) * i(p_5)$$
$$i(p_4) = i(p_3)$$
$$i(p_5) = i(p_9)$$

□

Lemma 25 *The commutativity rule is sound relative to* $\models_b$.

Proof

1. The output equation for C is $i(b_3) = i(b_1) + i(b_2)$ and the output equation for C' is $i(b_3) = i(b_2) + i(b_1)$. These are equivalent by commutativity of addition.

2. The output equation for C is $i(b_3) = i(b_1) * i(b_2)$ and the output equation for C' is $i(b_3) = i(b_2) * i(b_1)$. These are equivalent by commutativity of multiplication.

□

Lemma 26 *The identity rule is sound relative to* $\models_b$.

Proof

1. The output equation for C is $i(b_2) = i(b_1) + 0$, which is equivalent to the output equation for C', $i(b_2) = i(b_1)$.

2. The output equation for C is $i(b_2) = i(b_1) * 1$, which is equivalent to the output equation for C', $i(b_2) = i(b_1)$.

$\square$

Lemma 27 *The excluded-middle rule is sound relative to* $\models_b$.

Proof

1. The output equation for C is $i(b_2) = i(b_1) + 1 - i(b_1)$, derived from the following rules:

$$
\begin{aligned}
i(b_2) &= i(b_5) \\
i(b_5) &= i(p_3) + i(p_4) \\
i(p_4) &= i(p_2) \\
i(p_2) &= 1 - i(p_1) \\
i(p_1) &= i(b_1) \\
i(p_3) &= i(b_1)
\end{aligned}
$$

   The output equation for C' is $i(b_2) = 1$, which is equivalent to the output equation for C.

2. The output equation for C is $i(b_2) = i(b_1) * (1 - i(b_1))$, derived from the following rules:

$$
\begin{aligned}
i(b_2) &= i(b_5) \\
i(b_5) &= i(p_3) * i(p_4) \\
i(p_4) &= i(p_2) \\
i(p_2) &= 1 - i(p_1) \\
i(p_1) &= i(b_1)
\end{aligned}
$$

$$i(p_3) = i(b_1)$$

The output equation for C' is $i(b_2) = 1$. Under the restriction that the range of i is $\{0, 1\}$, this is equivalent to the output equation for C.

□

Lemma 28 *The branching rules are sound relative to* $\models_b$.

Proof

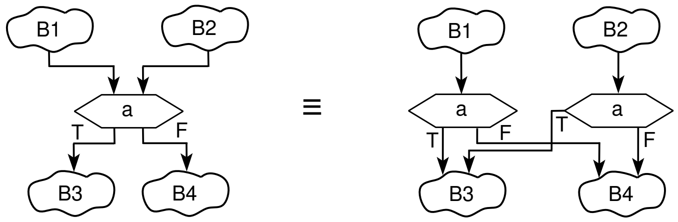
1. By definition, two gates of the same type must have the same output equations relative to their input ports. Since gates g and g' take the same inputs, their output equations must be identical, the derivation of C' from C does not affect the value on port p_2 at any time, so C and C' must be behaviorally equivalent.
2. By definition, two gates of the same type that are wired to the same inputs in each respective position must have identical output equations, so the value on port p_2 is the same whether it is wired to the output of g or the output of g'. C and C' must therefore be behaviorally equivalent.

□

4.2 ASM Chart to ASM Chart

In defining the rules between ASM charts, the charts appearing in examples use a blob-like shape to denote objects of unknown type. The names appearing inside the blobs are used only for reference purposes, and do not denote particular signal names. Like the circuit diagram rules, these ASM chart rules are all algebraic and are all defined relative to $\models_b$, the behavioral implementation relationship. Each rule is defined relative to ASM charts $M = \langle T, B, O, A, A_T, A_F, \mathsf{Signals}, \mathsf{Labels}, \mathsf{CurrStates}\rangle$ and $M' = \langle T', B', O', A', A'_T, A'_F, \mathsf{Signals}', \mathsf{Labels}', \mathsf{CurrStates}'\rangle$.

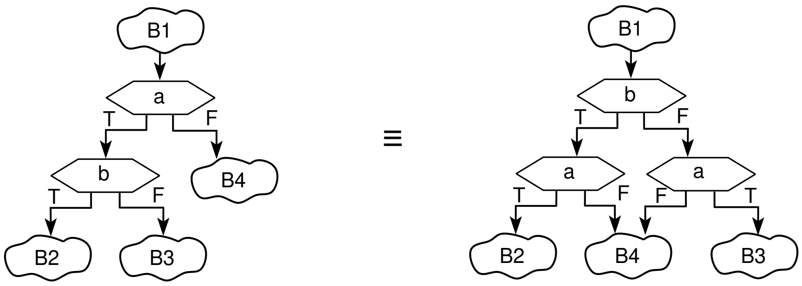
Rule 7 If multiple paths lead into a single conditional-branch object, that object can be duplicated, with some of the paths directed to each copy. Likewise, a duplicated conditional-branch object can be collapsed into a single copy.



1. ASM chart M' is derivable from ASM chart M by *test duplication* iff M contains conditional test object b and objects B_1 and B_2 such that (B_1, b) and (B_2, b) are arrow objects in M, and M' is obtained from M by adding a new conditional branch object b' , setting $\mathsf{Signals}(b')$ to $\mathsf{Signals}(b)$, adding (b', B_3) to A'_T where B_3 is the object such that $(b, B_3) \in A_T$, adding (b', B_4) to A'_F where B_4 is the object such that $(b, B_4) \in A_F$, and replacing arrow (B_2, b) with (B_2, b').

2. ASM chart M' is derivable from ASM chart M by *test collapsing* iff M contains two conditional branch objects b and b' such that $\mathsf{Signals}(b) = \mathsf{Signals}(b')$, there exist objects b_3 and b_4 such that (b, b_3) and (b', b_3) are both in A_T and (b, b_4) and (b', b_4) are both in A_F and M' is obtained from M by removing b', and replacing each arrow (B_2, b') with an arrow (B_2, b).

Rule 8 The order of tests along a conditional output path or next-state transition path are irrelevant. This rule supports the reordering of conditional branch objects along a path.

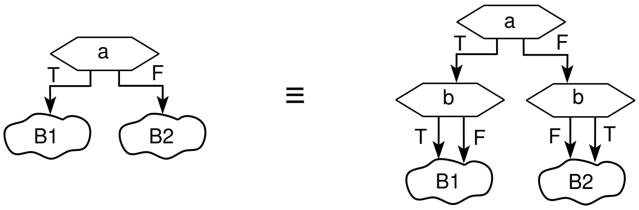


ASM chart M' is derivable from ASM chart M by *test reordering* iff either

1. M contains conditional branch objects b and b' such that (b, b') is in A_T or (b, b') is in A_F, no other arrows of the form (obj, b') exist, and M' is obtained from M by adding new conditional branch object b'' such that $\mathsf{Signals}(b) = \mathsf{Signals}(b'')$, removing arrow (b, b'), adding arrow (b', b) to A'_T, adding arrow (b', b'') to A'_F, adding arrow (b'', B_4) of the same type as arrow (b, B_4) where B_4 is not b', removing arrows (b', B_2) and (b', B_3), and adding arrows (b, B_2) and (b'', B_3) to A'_T if (b, B_4) is in A'_F, otherwise adding arrows (b, B_2) and (b'', B_3) to A'_F.

2. M contains conditional branch objects b, b', and b'' such that $\mathsf{Signals}(b) = \mathsf{Signals}(b'')$, $(b', b) \in A_T$, $(b', b'') \in A_F$, no other arrow objects of the form (obj, b) or (obj, b'') exist, there exist arrows (b, B_4) and (b'', B_4) either both in A_T or both in A_F, and M' is obtained from M by replacing arrow object (B_1, b') with arrow object (B_1, b) of the same type, removing arrows (b', b), (b', b''), (b, B_2), (b'', B_3), and (b'', B_4), removing b'', and adding arrow (b, b') with the opposite type of (b, B_4), adding arrow (b', B_2) to A'_T and (b', B_3) into A'_F.

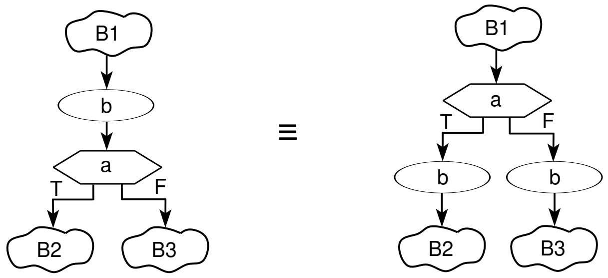
Rule 9 Signals that are not tested along a conditional output or next-state transition path are irrelevant to that path. This rule allows irrelevant tests to be added to a path, where both cases of the irrelevant test follow the path in the original ASM chart. This rule also allows irrelevant tests to be removed from a path.



1. ASM chart M' is derivable from ASM chart M by *test addition* iff M contains conditional branch object b and M' is obtained from M by adding new conditional branch objects b' and b'' such that $\mathsf{Signals}(b')$ equals $\mathsf{Signals}(b'')$, $\mathsf{Signals}(b')$ is disjoint from $\mathsf{Signals}(\text{obj})$ for any conditional branch object or conditional output object obj in the same state as b, $\mathsf{Signals}(b')$ is already an external signal in M, replacing true arrow (b, B_1) with arrow object (b, b'), replacing false arrow (b, B_2) with arrow object (b, b''), and adding (b', B_1) to each of A_T and A_F and adding (b'', B_2) to each of A_T and A_F.

2. ASM chart M' is derivable from ASM chart M by *test deletion* iff M contains conditional branch objects b, b', and b'' such that $(b, b') \in A_T$, $(b, b'') \in A_F$, (b', B_1) is in both A_T and A_F, (b'', B_2) is in both A_T and A_F, no objects other than b have arrows into either b' or b'', and M' is obtained from M by removing b' and b'' and all arrow objects involving them, and adding (b, B_1) to A_T and (b, B_2) to A_F.

Rule 10 This rule is like a distributivity rule for conditional output objects over conditional branch objects. A conditional output that does not depend upon a particular test may be propagated to both branches of that test. Similarly, a conditional output that holds regardless of the value of a particular test may be moved to just before that test.



ASM chart M' is derivable from ASM chart M by *output propagation* iff either of the following cases holds:

1. M contains conditional output object b and conditional branch object b' such that (b, b') is an arrow, there does not exist a path to b' that does not pass through b, and M' is obtained from M by adding new conditional output object b'' with the same $\mathsf{Signals}$ as b', replacing each arrow object (B_1, b) with (B_1, b'), removing arrow (b, b'), replacing true arrow (b', B_2) with (b', b), replacing false arrow (b', B_3) with (b', b''), adding arrow (b, B_2) and adding arrow (b'', B_3).

2. M contains conditional output objects b and b'' and conditional branch object b' such that $\mathsf{Signals}(b)$ equals $\mathsf{Signals}(b'')$, $(b', b) \in A_T$, $(b', b'') \in A_F$, no other arrow objects (b_a, b) or (b_a, b'') exist, and M' is obtained from M by replacing all arrow objects (B_1, b') with (B_1, b), removing arrow objects (b, B_2) and (b'', B_3), replacing true arrow (b', b) with (b', B_2), replacing false arrow (b', b'') with (b', B_3), and removing b'' and all remaining arrow objects defined on b''.

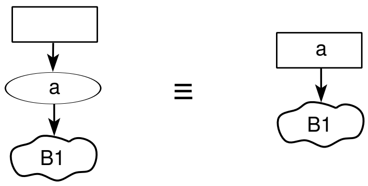
Rule 11 Two consecutive conditional output objects may be collapsed into one, provided no paths pass through one object and not the other. Similarly, a conditional branch object containing more than one signal name can be split into two conditional branch objects, each of which takes a non-empty subset of the signals from the original.

1. ASM chart M' is derived from ASM chart M by *output collapse* iff M contains conditional branch objects b and b' such that there is an arrow (b, b'), there does not exist an arrow (b'', b') for any other object b'', and M' is obtained from M by adding the contents of $\mathsf{Signals}(b')$

to $\mathsf{Signals}(b)$, removing arrow (b, b'), removing b', removing arrow (b', B_1) and adding arrow (b, B_1).

2. ASM chart M' is derived from ASM chart M' by *output duplicate* iff M contains a conditional branch object b containing more than one element in $\mathsf{Signals}$ and M' is obtained from M by adding a new conditional branch object b', partitioning the elements of $\mathsf{Signals}(b)$ between $\mathsf{Signals}(b)$ and $\mathsf{Signals}(b')$ such that neither $\mathsf{Signals}(b)$ nor $\mathsf{Signals}(b')$ is empty, removing arrow (b, B_1), adding arrow (b', B_1), adding an arrow (b, b'),

Rule 12 Conditional outputs that rely on no tests can be moved into states, and vice versa. This rule essentially converts certain Mealy outputs to Moore outputs, and Moore outputs to Mealy outputs.

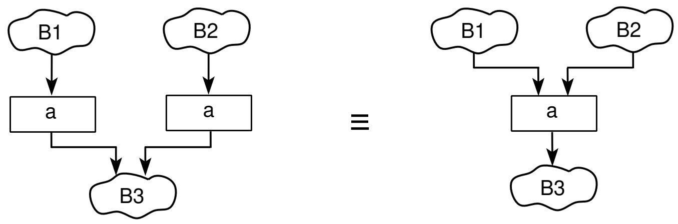


ASM chart M' is derivable from ASM chart M by *output state* if either

1. M contains a state object b and a conditional output object b' such that there exists an arrow (b, b'), there does not exist any other arrow object (b'', b') and M' is obtained from M by replacing arrow (b', B_1) with arrow (b, B_1), adding $\mathsf{Signals}(b')$ to $\mathsf{Signals}(b)$, and removing b'.

2. M contains a state object b such that $\mathsf{Signals}(b)$ is non-empty and M' is obtained from M by adding new conditional output object b', replacing arrow (b, B_1) with (b', B_1), adding arrow (b, b'), and partitioning the elements of $\mathsf{Signals}(b)$ between $\mathsf{Signals}(b)$ and $\mathsf{Signals}(b')$ such that $\mathsf{Signals}(b')$ is not empty.

Rule 13 This rule allows states to be duplicated, and allows duplicated states to be collapsed.



1. ASM chart M' is derivable from ASM chart M by *state merge* iff there exist state objects b and b' with arrows (b, B_3) and (b', B_3) such that $\mathsf{Signals}(b)$ equal $\mathsf{Signals}(b')$, and M' is obtained from M by removing arrow (b', B_3), removing b', and replacing all remaining occurrences of b' with b.

2. ASM chart M' is derivable from ASM chart M by *state split* iff M contains a state b with distinct arrows (B_1, b) and (B_2, b) and M' is obtained from M by adding a new state object b', adding arrow (b', B_3) such that there exists arrow (b, B_3), setting $\mathsf{Signals}(b')$ to $\mathsf{Signals}(b)$, and replacing some, but not all, arrows (B_2, b) with (B_2, b').

4.2.1 Soundness

The soundness proofs in this section are based upon Lemma 5. Those rules that preserve the number of states in the chart are proven treating function τ from the lemma as the identity function. Those rules that alter the number of states in the chart have their particular τ stated in the respective proof. Once τ is given, it suffices from the lemma to prove that the next-state transitions and output conditions are unaffected after the application of each rule.

Lemma 29 *The test duplication rule is sound relative to* $\models_b$.

Proof The next-state and output transitions of M' must be shown equivalent to the next-state and output transitions of M. The only potential differences between M and M' lie in the next-state and output transitions involving conditional test object b and object b_2. Let $t = \langle t_s, t_t, c_t, c_f \rangle$ be a next-state transition in M that involves b and b_2. Then by assumption, there must exist a next-state transition path $p = j_0, \ldots, j_k$ in M such that t is the next-state transition corresponding to p, j_i equals b for some $0 < i < k$, and j_{i-1} equals b_2.

Given the structure of M, j_{i+1} is either object b_3 or object b_4. If $j_{i+1} = b_3$, then $a \in c_t$ since (b, b_3) is in A_T. By construction, M' contains a next-state transition path $p' = j_0, \ldots, b_2, b', b_3, \ldots, j_k$. By construction of M', $(b', b_3) \in A_T$, so the next-state transition t' corresponding to p' has $a \in c_t$. Since all other objects are position-wise the same between the p and p', t' must be identical to t.

If $j_{i+1} = b_4$, then $a \in c_f$ since (b, b_4) is in A_F. By construction, M' contains a next-state transition path $p' = j_0, \ldots, b_2, b', b_4, \ldots, j_k$. By construction of M', $(b', b_4) \in A_F$, so the next-state transition t' corresponding to p' has $a \in c_f$. Since all other objects are position-wise the same between the p and p', t' must be identical to t.

A similar argument establishes that any output condition involving b in M has an identical output condition in M'.

It remains to show that every next-state transition or output condition in M' is also in M. By construction, any transition or condition path involving b' has a corresponding transition or condition path in M where b is used in place of b'. Since the arrows leaving b are identical to the arrows leaving b', these transition and condition paths cannot generate next-state transitions or output conditions that are not in M. Any transition or condition path in M' that does not involve b' also appears in M by construction.

□

Lemma 30 *The test collapsing rule is sound relative to* $\models_b$.

Proof The next-state and output transitions of M' must be shown equivalent to the next-state and output transitions of M. The only potential differences between M and M' lie in the next-state and output transitions involving conditional test object b'. Let $t = \langle t_s, t_t, c_t, c_f \rangle$ be a next-state transition in M that involves b'. Then there must exist a next-state transition path $p = j_0, \ldots, j_k$ in M such that t is the next-state transition corresponding to p and j_i equals b' for some $0 < i < k$.

M contains some object b that is distinct from b' such that (b, b_3) and (b', b_3) are both in A_T and (b, b_4) and (b', b_4) are both in A_F. For every flow path p' in M containing b', there must exist another flow path p in M that is identical to p' with the exception of the use of b in place of b'. All of these flow paths involving b are contained in M'. M' is constructed by redirecting every flow path $j_0, \ldots, j_k$ where $j_k = b'$ to be a flow path $j_0, \ldots, j_k$ where $j_k = b$. Hence, for every flow path containing b' in M there is a flow path containing b in M' such that the two flow paths are equivalent.

A similar argument establishes this property for output conditions.

□

Lemma 31 *The test reordering rule is sound relative to* $\models_b$.

Proof By construction, M and M' have the same states. It therefore suffices to prove that whenever a next-state transition $\langle t_s, t_t, c_t, c_f \rangle$ is satisfied under an external signal assignment A in M, there exists a next-state transition $\langle t_s, t_t, c'_t, c'_f \rangle$ satisfied by A in M' and that output condition $\langle t_s, n, c_t, c_f \rangle$ is satisfied under an external signal assignment A in M iff there exists an output condition $\langle t_s, n, c'_t, c'_f \rangle$ satisfied by A in M'. The proof is given with respect to next-state transitions, though the proof for the output conditions is analogous. Each case of the test reordering rule is be treated in turn:

1. In the first case, the only potential differences between M and M' lie in transitions or output conditions that involve one or both of conditional branch objects b and b'. Since by assumption, M does not contain any arrows into b' other than arrow (b, b'), only flow paths that contain b need to be considered.

Let $p = j_0, \ldots, j_k$ be a next-state transition path in M such that b is object j_i. Let $\langle j_0, j_k, c_t, c_f\rangle$ be the next state transition corresponding to p. There are now three cases to consider:

(a) Object j_{i+1} is b' and (b, b') is in A_T.

If (b', j_{i+2}) is in A_T, then $\mathsf{Signals}(b)$ and $\mathsf{Signals}(b')$ are both in c_t. By construction, M' contains next-state transition path $j_0, \ldots, j_{i-1}, b', b, j_{i+2}, \ldots j_k$ such that the type of (j_{i-1}, b') in M' is the same as the type of (j_{i-1}, b) in M, (b', b) is in A'_T, and arrow (b, j_{i+2}) is in A'_T. The next-state transition corresponding to this path therefore has $\mathsf{Signals}(b)$ and $\mathsf{Signals}(b')$ both in the true conditions, with the conditions of the remainder of the path unaffected. As a result, this next-state transition must be satisfied whenever $\langle j_0, j_k, c_t, c_f\rangle$ is satisfied.

If (b', j_{i+2}) is in A_F, then $\mathsf{Signals}(b)$ is in c_t and $\mathsf{Signals}(b')$ is in c_f. By construction, M' contains next-state transition path $j_0, \ldots, j_{i-1}, b', b'', j_{i+2}, \ldots j_k$ such that the type of (j_{i-1}, b') in M' is the same as the type of (j_{i-1}, b) in M, (b', b'') is in A'_F, and arrow (b'', j_{i+2}) is in A'_T. This path produces a next-state transition with $\mathsf{Signals}(b')$ in c_f and $\mathsf{Signals}(b'')$ in c_t. Since $\mathsf{Signals}(b'')$ is equal to $\mathsf{Signals}(b)$ by construction and all other elements of c_t and c_f are preserved in the transformation, this transition is satisfied whenever the original is satisfied.

(b) Object j_{i+1} is b' and (b, b') is in A_F.

If (b', j_{i+2}) is in A_T, then $\mathsf{Signals}(b)$ is in c_f and $\mathsf{Signals}(b')$ is in c_t. By construction, M' contains next-state transition path $j_0, \ldots, j_{i-1}, b', b, j_{i+2}, \ldots j_k$ such that the type of (j_{i-1}, b') in M' is the same as the type of (j_{i-1}, b) in M, (b', b) is in A'_T, and arrow (b, j_{i+2}) is in A'_F. The next-state transition corresponding to this path therefore has $\mathsf{Signals}(b)$ in the false conditions and $\mathsf{Signals}(b')$ in the true conditions, with the conditions of the remainder of the path unaffected. As a result, this next-state transition must be satisfied whenever $\langle j_0, j_k, c_t, c_f\rangle$ is satisfied.

If (b', j_{i+2}) is in A_F, then $\mathsf{Signals}(b)$ and $\mathsf{Signals}(b')$ are both in c_f. By construction, M' contains next-state transition path $j_0, \ldots, j_{i-1}, b', b'', j_{i+2}, \ldots j_k$ such that the type of (j_{i-1}, b') in M' is the same as the type of (j_{i-1}, b) in M, (b', b'') is in A'_F, and arrow (b'', j_{i+2}) is in A'_F. This path produces a next-state transition with $\mathsf{Signals}(b')$ and $\mathsf{Signals}(b'')$ both in c_f. Since $\mathsf{Signals}(b'')$ is equal to $\mathsf{Signals}(b)$ by construction and all other elements of c_t and c_f are preserved in the transformation, this transition is satisfied whenever the original is satisfied.

(c) Object j_{i+1} is not b'.

In this case, $\mathsf{Signals}(b')$ cannot appear in either c_t or c_f. By construction, M' contains next-state transition paths $p_1 = j_0, \ldots, j_{i-1}, b', b, j_{i+1}, \ldots, j_k$ and $p_2 = j_0, \ldots, j_{i-1}, b', b''$, $j_{i+1}, \ldots, j_k$ such that the arrow type of (b, j_{i+1}) is the same as in M and (b'', j_{i+1}) is of the same arrow type as (b, j_{i+1}). Let $\langle j_0, j_k, c_{t_1}, c_{f_1}\rangle$ and $\langle j_0, j_k, c_{t_2}, c_{f_2}\rangle$ denote the next-state transitions corresponding to p_1 and p_2, respectively.

$\mathsf{Signals}(b)$ is in c_{t_1} iff $\mathsf{Signals}(b)$ is in c_t and that $\mathsf{Signals}(b'')$ is in c_{t_2} iff $\mathsf{Signals}(b)$ is in c_t. Similarly, $\mathsf{Signals}(b)$ is in c_{f_1} iff $\mathsf{Signals}(b)$ is in c_f and $\mathsf{Signals}(b'')$ is in c_{f_2} iff $\mathsf{Signals}(b)$ is in c_f. By construction, $\mathsf{Signals}(b')$ is in each of c_{t_1} and c_{f_2}. Since $\mathsf{Signals}(b'')$ equals $\mathsf{Signals}(b)$ by construction, whenever $\langle j_0, j_k, c_t, c_f\rangle$ is satisfied, one of $\langle j_0, j_k, c_{t_1}, c_{f_1}\rangle$ or $\langle j_0, j_k, c_{t_2}, c_{f_2}\rangle$ must be satisfied, since they are identical to the original next-state transition, with the exception that one adds $\mathsf{Signals}(b')$ to the true conditions and the other to the false conditions.

2. In the second case, the only flow-paths that need to be considered are those involving object b', since any flow-path involving b or b'' must involve b' by assumption. By construction, any true or false conditions arising from a flow-path containing b' are only affected with respect to the elements of $\mathsf{Signals}(b')$, $\mathsf{Signals}(b)$, or $\mathsf{Signals}(b'')$.

   By assumption, M has flow-paths of the form $j_0, \ldots, j_{i-1}, b', b, j_{i+2}, \ldots, j_k$ and $j_0, \ldots, j_{i-1}, b', b''$, $j_{i+2}, \ldots, j_k$ such that (b, j_{i+2}) and (b'', j_{i+2}) are of the same type. Since one of (b', b) and (b', b'') must be in A_T and the other in A_F; whether $\mathsf{Signals}(b')$ is in c_t or c_f is irrelevant in the case that $\mathsf{Signals}(b)$ or $\mathsf{Signals}(b'')$ are in c_t or c_f as specified by the arrow types of (b, j_{i+2}) and (b'', j_{i+2}). $\mathsf{Signals}(b)$ and $\mathsf{Signals}(b'')$ are equal by assumption. By construction, M' contains flow path $j_0, \ldots, j_{i-1}, b, j_{i+2}$ such that (b, j_{i+2}) has the same arrow type as in M. Any next-state transition or output condition along this path therefore satisfies a next-state transition or output condition along either of the original paths.

This leaves two flow-paths from M involving b' unconsidered: $j_0, \ldots, j_{i-1}, b', b, j_{i+2}, \ldots, j_k$ and $j_0, \ldots, j_{i-1}, b', b'', j'_{i+2}, \ldots, j'_k$, where j'_{i+2} is distinct from j_{i+2}. Arrows (b, j_{i+2}) and (b'', j_{i+2}) must be of the same type by assumption. Also by assumption, (b', b) is in A_T and (b', b'') is in A_F. By construction, M' contains flow-paths $j_0, \ldots, j_{i-1}, b, b', j_{i+2}, \ldots, j_k$ and $j_0, \ldots, j_{i-1}, b, b', j'_{i+2}, \ldots, j'_k$. Also by construction, (b', j_{i+2}) is in A'_T, (b', j'_{i+2}) is in A'_F, and (b, b') in M' is of the same type as (b, j_{i+2}) and (b'', j_{i+2}) in M. Therefore, regardless of the inclusion of $\mathsf{Signals}(b')$ and $\mathsf{Signals}(b)$ in c_t and c_f, M' contains a flow-path that is otherwise identical to the original flow-path that preserves the arrangement of these signals in c_t and c_f.

□

Lemma 32 *The test addition rule is sound relative to* $\models_b$.

Proof By construction, the only next-state transition paths and conditional output paths of M potentially affected by this rule are those involving object b.

Let $j_0, \ldots, j_{i-1}, b, j_{i+2}, \ldots, j_k$ be a next-state transition path in M. Let $\langle j_0, j_k, c_t, c_f \rangle$ be the next-state transition corresponding to this path. It must be established that whenever $\langle j_0, j_k, c_t, c_f \rangle$ is satisfied in M, it must also be satisfied in M'. Assume $\langle j_0, j_k, c_t, c_f \rangle$ is satisfied in M. Without loss of generality, assume $\mathsf{Signals}(b)$ is in c_t.

By construction, M' contains next-state transition path $j_0, \ldots, j_{i-1}, b, b', j_{i+2}, \ldots, j_k$ such that (b', j_{i+2}) is in both A'_T and A'_F. Therefore, regardless of the value of $\mathsf{Signals}(b')$, there is a next-state transition in M' that is satisfied for the same source and target state.

A similar argument establishes that this rule does not affect the output conditions between M and M'.

□

Lemma 33 *The test deletion rule is sound relative to* $\models_b$.

Proof By construction, the only next-state transition paths and conditional output paths of M potentially affected by this rule are those involving object b, since all paths including either b' or b'' must also be paths including b.

Let $j_0, \ldots, j_{i-1}, b, b', j_{i+2}, \ldots, j_k$ be a next-state transition path in M. Let $\langle j_0, j_k, c_t, c_f\rangle$ be the next-state transition corresponding to this path. It must be established that whenever $\langle j_0, j_k, c_t, c_f\rangle$ is satisfied in M, it must also be satisfied in M'. Assume $\langle j_0, j_k, c_t, c_f\rangle$ is satisfied in M. Without loss of generality, assume $\mathsf{Signals}(b')$ is in c_t. There must exist another next-state transition $\langle j_0, j_k, c'_t, c'_f\rangle$ such that c'_t equals c_t without $\mathsf{Signals}(b')$ and c'_f equals c_f plus $\mathsf{Signals}(b')$ since M is transitionally complete (Defn. 40). The value of $\mathsf{Signals}(b')$ is therefore irrelevant to these next-state transitions. By construction, M' contains next-state transition path $j_0, \ldots, j_{i-1}, b, j_{i+2}, \ldots, j_k$ such that the arrow (b, j_{i+2}) has the same type as (b, b') did in M. $\mathsf{Signals}(b)$ therefore plays the same role in either c_t or c_f as it did in the next-state transition in M. By construction, all other signals also continue to play the same, if any, roles that they did in the next-state transition in M. Whenever $\langle j_0, j_k, c_t, c_f\rangle$ is satisfied in M, it therefore must also be satisfied in M'.

A similar argument establishes that this rule does not affect the output conditions between M and M'.

□

Lemma 34 *The output propagation rule is sound relative to* $\models_b$.

Proof The two cases of this rule are considered in turn:

1. This rule does not affect the next-state transitions. Hence only the effects on output conditions need to be considered. Let $p = j_0, \ldots, j_{i-1}, b, b', j_{i+2}, \ldots, j_k$ be a flow-path in M such that b

is a conditional output object and b' is a conditional branch object. Let $\langle j_0, n, c_t, c_f \rangle$ be an output condition corresponding to the path $j_0, \ldots, j_{i-1}, b$. By the well-formedness rules on ASM charts (Defn 33), $\mathsf{Signals}(b)$ cannot be an element of either c_t or c_f. By construction, M' yields output conditions $\langle j_0, n, c'_t, c_f \rangle$ and $\langle j_0, n, c_t, c'_f \rangle$ where c'_t contains all elements of c_t plus $\mathsf{Signals}(b')$ and c'_f contains all elements of c_f plus $\mathsf{Signals}(b')$. Since the element in $\mathsf{Signals}(b')$ must be either true or false, one of these conditions must be satisfied whenever the original condition is satisfied. In addition, neither of these conditions is satisfied unless the original would also have been satisfied. No other conditional output paths need to be considered since by assumption there are no paths to b' that do not involve b.

2. This rule does not affect the next-state transitions. Hence only the affects on output conditions need to be considered. By assumption, M contains conditional output paths $j_0, \ldots, j_{i-1}, b', b$ and $j_0, \ldots, j_{i-1}, b', b''$ such that $\mathsf{Signals}(b)$ equals $\mathsf{Signals}(b'')$ for distinct b, b''. By the well-formedness conditions on ASM charts, one of (b', b) and (b', b'') must be in A_T and the other must be in A_F. Therefore, there are output conditions for M such that the value of $\mathsf{Signals}(b')$ is irrelevant. By construction, M' contains a conditional output path $j_0, \ldots, j_{i-1}, b$, which is also irrelevant as far as the value of $\mathsf{Signals}(b')$ is concerned. An output condition for $\mathsf{Signals}(b)$ is therefore satisfied in M iff it is satisfied in M'.

$\square$

Lemma 35 *The output collapse rule is sound relative to* $\models_b$.

Proof Since the output collapse rule does not affect the transition structure of an ASM chart, it suffices to show that all output conditions in M are also output conditions in M' and vice-versa. Only output condition paths involving objects b and b' need to be considered. By assumption, any flow-path to b' in M must pass through b. As M' is structurally identical to M in all output condition paths ending with b by construction, only output condition paths ending at b' need to be considered.

Let p be an output condition path $j_0, \ldots, j_k$ of M such that $j_k = b'$ and let output condition $\langle t_s, n, c_t, c_f \rangle$ correspond to p; by assumption, $j_{k-1} = b$. By construction, M' contains output condition path $j_0, \ldots, j_{k-1}$, where $\mathsf{Signals}(b)$ contains $\mathsf{Signals}(b')$. By Definition 35, an output condition $\langle t_s, n, c_t, c_f \rangle$ must therefore be generated in M'.

M' cannot contain any output conditions that were not originally in M. Since no new conditional output objects were added to M', any difference in output objects must arise from the merging of $\mathsf{Signals}(b')$ into $\mathsf{Signals}(b)$. By assumption, any conditional output objects preceding b in a flow path must also have preceded b', and no objects lay between b and b', so no conditional branch object can affect a change in output conditions when merging $\mathsf{Signals}(b')$ into $\mathsf{Signals}(b)$. All output conditions of M' were therefore output conditions of M.

□

Lemma 36 *The output duplicate rule is sound relative to* $\models_b$.

Proof Any next-state transition in M is in M' and vice-versa. The only paths of M that are affected by the output duplicate rule are those that contain the arrow (b, b_1). By construction, M' replaces every flow-path of the form b, b_1 with the flow-path b, b', b_1, where b' is a conditional output object. Conditional output objects do not affect next-state transitions, so the next-state transitions of M and M' are identical.

Any output condition in M is an output condition in M' and vice versa. The only output conditions possibly affected by the output duplicate rule are those involving conditional output object b. Let $j_0, \ldots, b$ be a conditional output path in M. By construction, M' contains conditional output paths $j_0, \ldots, b$ and $j_0, \ldots, b, b'$. Since every element of $\mathsf{Signals}(b)$ in M is in one of $\mathsf{Signals}(b)$ or $\mathsf{Signals}(b')$ by assumption, every conditional output object in M must also be in M'. M' does not contain any conditional output objects that are not in M. If it did, there would be an element of $\mathsf{Signals}(b)$ or $\mathsf{Signals}(b')$ that was not an element of $\mathsf{Signals}(b)$ in M. By assumption, this cannot occur, so M and M' have the same set of output conditions. □

Lemma 37 *The output state rule is sound relative to* $\models_b$.

Proof The two cases of this rule are considered separately.

1. Any next-state transition in M is in M' and vice-versa. The only paths of M that are affected by the first case of this rule are those that contain the flow-path b, b', b_1. By construction, M' replaces every flow-path of the form b, b', b_1 with the flow-path b, b_1. Conditional output objects do not affect next-state transitions, so the next-state transitions of M and M' are identical.

   Any output condition in M is an output condition in M' and vice-versa. The only output conditions possibly affected by this rule are those occurring in state b. Let $\langle b, n, c_t, c_f \rangle$ be a conditional output object in M'. If either c_t or c_f contains an element, then the conditional output path must extend beyond b_1 because there is no conditional branch object in state b before b_1. Output transitions past this point are unaffected by the rule due to a similar argument made for the non-effect on next-state transitions. If c_t and c_f are both empty, then either $n \in \mathsf{Signals}(b)$ or $n \in \mathsf{Signals}(b'')$ for some b'' such that there are no conditional output objects on the flow path from b to b''. In the latter case, the same argument about non-affectedness holds.

   The only remaining case, then, is $n \in \mathsf{Signals}(b)$ in M'. If $n \in \mathsf{Signals}(b)$ in M, then $n \in \mathsf{Signals}(b)$ in M' because the rule does not remove any Signals from b. If $n \notin \mathsf{Signals}(b)$ in M, then n had to have been in $\mathsf{Signals}(b')$ in M since the signals added to $\mathsf{Signals}(b)$ in M' are exactly those that were in b'. The output conditions of M and m' therefore must be identical.

2. Any next-state transition in M is in M' and vice-versa. The only paths of M that are affected by the output duplicate rule are those that contain the arrow (b, b_1). By construction, M' replaces every flow-path of the form b, b_1 with the flow-path b, b', b_1, where b' is a conditional output object. Conditional output objects do not affect next-state transitions, so the next-state transitions of M and M' are identical.

Any output condition of M is an output condition in M' and vice-versa. Arguments similar to those used in the case above eliminate from consideration all output conditions except those of the form $\langle b, n, c_t, c_f \rangle$ where c_t and c_f are empty. In addition, any such conditions are eliminated unless $n \in \mathsf{Signals}(b)$ or $n \in \mathsf{Signals}(b')$ in M'.

If $n \in \mathsf{Signals}(b)$ in M', then $n \in \mathsf{Signals}(b)$ in M since the rule does not add any new element to $\mathsf{Signals}(b)$. If $n \in \mathsf{Signals}(b')$ in M', then by construction $n \in \mathsf{Signals}(b)$ in M. Since there are no conditional branch objects between b and b', the output condition is the same whether $n \in \mathsf{Signals}(b)$ or $n \in \mathsf{Signals}(b')$. Since all elements of $\mathsf{Signals}(b)$ in M are guaranteed to be in one of these two sets, the output conditions in M and M' must be identical.

□

Lemma 38 *The state merge rule is sound relative to* $\models_b$.

Proof Define function τ from the states of M to the states of M' such that τ assigns b to b' and assigns every other state to itself. By Lemma 5, this rule is sound iff τ preserves the next-state transitions and output conditions of M in M'. Any next-state transitions or output conditions arising from states mapped to themselves in τ are unaffected, so only those next-state transitions and output conditions involving the state represented by b' need to be considered.

Let $\langle b', t_t, c_t, c_f \rangle$ be a next-state transition in M. By statement of Lemma 5, it must be shown that whenever this next-state transition is satisfied in M, next-state transition $\langle b, t_t, c_t, c_f \rangle$ is satisfied in M'. By assumption, M had the same transitions leaving b and b', so $\langle b, t_t, c_t, c_f \rangle$ must be a next-state transition in M'.

Let $\langle t_s, b', c_t, c_f \rangle$ be a next-state transition in M. By statement of Lemma 5, it must be shown that whenever this next-state transition is satisfied in M, next-state transition $\langle t_s, b, c_t, c_f \rangle$ is satisfied in M'. By construction, however, M' replaced all occurrences of b' with b, so $\langle t_s, b', c_t, c_f \rangle$ must be a next-state transition in M'.

Let $\langle b', n, c_t, c_f \rangle$ be an output condition in M. Assume c_t and c_f are both empty. Then by Definition 35, n is in $\mathsf{Signals}(b')$. By assumption, $\mathsf{Signals}(b)$ and $\mathsf{Signals}(b')$ are equal, so $\langle b, n, \emptyset, \emptyset \rangle$ must be an output condition in M'. Therefore, $\langle b', n, \emptyset, \emptyset \rangle$ is an output condition in M iff $\langle b, n, \emptyset, \emptyset \rangle$ is an output condition in M'.

Now assume one of c_t and c_f is not empty. Since all flow-paths leaving b are identical minus the first object to flow-paths leaving b' in M by assumption, $\langle b, n, c_t, c_f \rangle$ must also have been an output condition in M. This output condition is in M' since state b is unaffected by the rule. By a similar argument, no output condition could exist for state b that did not also exist for state b'. $\langle b', n, c_t, c_f \rangle$ is therefore an output condition in M iff $\langle b, n, c_t, c_f \rangle$ is an output condition in M'. □

Lemma 39 *The state split rule is sound relative to* $\models_b$.

Proof The proof given for the soundness of the state merge rule (Lemma 38) establishes that the ASM charts used in the state merge rule are behaviorally equivalent. Since the state split rule is simply the reverse operation of the state merge rule, the ASM charts appearing in this rule are also behaviorally equivalent, which is sufficient to establish soundness. □

4.3 CTL to CTL

The Axioms of CTL:

- All tautologies of propositional logic
- $\mathsf{EF}p \equiv \mathsf{E}[\mathsf{true}\mathsf{U}p]$
- $\mathsf{AG}p \equiv \neg\mathsf{EF}\neg p$
- $\mathsf{AF}p \equiv \mathsf{A}[\mathsf{true}\mathsf{U}p]$
- $\mathsf{EG}p \equiv \neg\mathsf{AF}\neg p$

- $\mathsf{EX}(p \vee q) \equiv \mathsf{EX}p \vee \mathsf{EX}q$
- $\mathsf{AX}p \equiv \neg\mathsf{EX}\neg p$
- $\mathsf{E}[p\mathsf{U}q] \equiv q \vee (p \wedge \mathsf{EXE}[p\mathsf{U}q])$
- $\mathsf{A}[p\mathsf{U}q] \equiv q \vee (p \wedge \mathsf{AXA}[p\mathsf{U}q])$
- $\mathsf{EX}\ \mathsf{true} \wedge \mathsf{AX}\ \mathsf{true}$
- $\mathsf{AG}(r \rightarrow (\neg q \wedge \mathsf{EX}r)) \rightarrow (r \rightarrow \neg\mathsf{A}[p\mathsf{U}q])$
- $\mathsf{AG}(r \rightarrow (\neg q \wedge \mathsf{EX}r)) \rightarrow (r \rightarrow \neg\mathsf{AF}q)$
- $\mathsf{AG}(r \rightarrow (\neg q \wedge (p \rightarrow \mathsf{AX}r))) \rightarrow (r \rightarrow \neg\mathsf{E}[p\mathsf{U}q])$
- $\mathsf{AG}(r \rightarrow (\neg q \wedge \mathsf{AX}r)) \rightarrow (r \rightarrow \neg\mathsf{EF}q)$
- $\mathsf{AG}(p \rightarrow q) \rightarrow (\mathsf{EX}p \rightarrow \mathsf{EX}q)$

The Inference Rules of CTL:

- If $\vdash p$ and $\vdash p \rightarrow q$, then $\vdash q$ (modus ponens)
- If $\vdash p$ then $\vdash \mathsf{AG}p$ (Generalization)

These axioms and inference rules are reproduced from [28], where they are proven sound and complete.

4.4 SOL to SOL

The rules of SOL are the standard inference rules of second order logic.

$$\frac{\varphi, \psi}{\varphi \wedge \psi} \quad \frac{\varphi \wedge \psi}{\varphi} \quad \frac{\varphi \wedge \psi}{\psi} \quad \frac{\varphi, \varphi \rightarrow \psi}{\psi} \quad \frac{\forall x \varphi(x)}{\varphi(t)} \quad \frac{\varphi(t)}{\forall x \varphi(x)}$$

In $\forall$-elimination (second rule from right), t must be free for x in φ; formally, this requires one of the following cases to hold [26]:

- φ is an atomic formula
- φ is of the form $\neg\varphi_1$ or $\varphi_1 \wedge \varphi_2$ and t is free for x in φ_1 and φ_2
- φ is of the form $\forall y \psi$ and either x is not in the free variables of φ or y is not the free variables of t and t is free for x in ψ.

In $\forall$-introduction (rightmost rule), x cannot occur free in any assumption on which $\varphi(t)$ depends. This list of rules does not include one for inferring the negation of a statements from a contradiction because this rule is presented more generally when proofs are defined (Defn. 97).

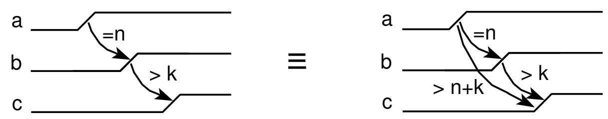
4.4.1 Soundness

The soundness of these rules follows trivially from the definition of SOL.

4.5 Timing Diagram to Timing Diagram

Unless otherwise noted, each timing diagram in this section is defined relative to both $\models_{\mathrm{Inv}}$ and $\models_{\mathrm{Iter}}$.

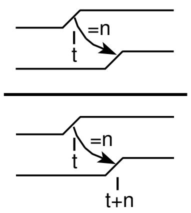
Rule 14 If there are successor edge objects (b_1, b_2) and (b_2, b_3), then by transitivity there can be a successor edge object from b_1 to b_3. The time-bound on the new edge should be the sum of the time-bounds on the original edges. Such transitive edges can also be removed from a diagram.



Timing diagram T' is derivable from timing diagram T by *transitive edges* iff either

1. T contains successor edge objects (b_1, b_2) and (b_2, b_3) and T' is obtained from T by adding successor edge object (b_1, b_3) with time bound $[l, u]$ such that
   (a) $l = l_1 + l_2$ where l_1 and l_2 are the lower bounds associated with (b_1, b_2) and (b_2, b_3), respectively, if bounds are defined on these objects, else 1 if no bound is defined on an object.
   (b) $u = u_1 + u_2$ where u_1 and u_2 are the upper bounds associated with (b_1, b_2) and (b_2, b_3), respectively, if bounds are defined on these objects, else ∞ if no bound is defined on an object.
2. T contains successor edge objects (b_1, b_2), (b_2, b_3), and (b_1, b_3) with time bound expressions $[l_1, u_1]$, $[l_2, u_2]$, and $[l_1 + l_2, u_1 + u_2]$, respectively, and T' is obtained from T by removing successor edge object (b_1, b_3) and its associated time-bound.

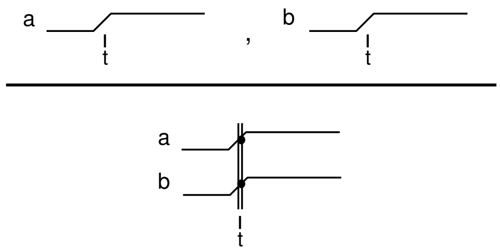
Rule 15 If a timing diagram contains a successor edge object with a time bound of the form $= n$ and a time anchor on the source event, a time anchor can be placed on the target event labeled with the sum of n and the discrete time expression labeling the source event.



Timing diagram T' is derivable from timing diagram T by *time label bounds* iff T contains a time anchor (b_1, e_1), there is a successor edge (b_1, b_2) with a time-bound expression of the form $= n$, and

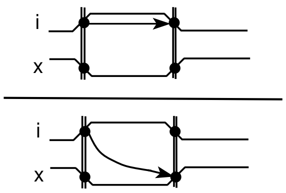
T' is obtained from T by adding time anchor $(b_2, e_1 + n)$.

Rule 16 If two timing diagrams contain time anchors with a common discrete time expression, then they can be merged into one diagram with a concurrent edge object connecting the similarly anchored events.



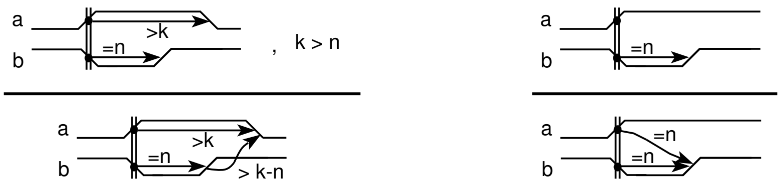
TD$^{\text{TI}}$timing diagram T'' is derivable from TD$^{\text{TI}}$timing diagrams T and T' by *anchor align* if T contains time anchor (b, e), T' contains time anchor (b', e), T and T' have no free variables in common outside of those in e, and T'' is the union of T and T' plus a concurrent edge object connecting b and b', labeled with a time anchor with discrete time expression e.

Rule 17 If a successor edge object exists between two concurrent edge objects, the edge may be duplicated or slid to connect any other pair of objects along those same objects.



Timing diagram T' is derivable from timing diagram T by *bounds slide* if T contains concurrent edge objects b_i and b_j connected by a successor edge object and T' is obtained from T by adding successor edge object (b'_i, b'_j) with the same time bound object as (b_i, b_j) where b'_i is involved in the same concurrent edge object as b_i and b'_j is involved in the same concurrent edge object as b_j. T' may or may not contain $(b_i.b_j)$.

Rule 18 If two successor edge objects share an endpoint, it is often possible to derive some information about the relationship between their non-similar endpoints. This rule captures the inferences that can be made about the time-bounds between the non-similar endpoints.



Timing diagram T' is derivable from timing diagram T by *derived bounds* iff one of the following cases holds:

1. T contains a concurrent edge object at time point p_a, a successor edge object (b_1, b_2) with b_1 at p_a, a time-bound expression on (b_1, b_2), some other transition object b_3 at time point p_a, and T' is obtained from T by adding successor edge (b_1, b_3) with the same duration label as (b_3, b_2).

2. T contains a concurrent edge object at time point p_a and a successor edge object (b_1, b_2) containing a duration label such that b_2 is at time point p_a, T contains some other transition object b_3 at time point p_a, and T' is obtained from T by adding successor edge (b_3, b_2) with the same duration label as (b_1, b_3).

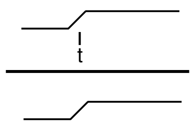
3. T contains successor edges (p_a, p_b) and (p_a, p_c) with bounds $[l_1, u_1]$, $[l_2, u_2]$, respectively, and T' is obtained from T by adding a new object with time-bound expression as indicated in the table below. In the table headings, e_1 and e_2 are the time-bound expressions on (p_a, p_c) and (p_b, p_c) respectively. "Condition" indicates additional information that must hold for the corresponding bound to be derived. e_3 is the time bound expression for the new object. b_b and b_c represent the objects corresponding to time points p_b and p_c in the successor edge objects. If there is no table entry corresponding to the bounds on (p_a, p_c) and (p_b, p_c), then no inference can be drawn.

e_1	e_2	condition	new object	e_3
$=n$	$=k$	$n=k$	$(p_b, \{b_b, b_c\})$	
		$n<k$	(p_b, p_c)	$=k-n$
		$n>k$	(p_c, p_b)	$=n-k$
$=n$	$<k$	$n=k$	(p_c, p_b)	
		$n>k$	(p_c, p_b)	$[n-k+1, n-1]$
$=n$	$>k$	$n=k$	(p_b, p_c)	
		$n<k$	(p_b, p_c)	$<k-n$

4. T contains successor edges (p_a, p_c) and (p_b, p_c) with bounds $[l_1, u_1]$, $[l_2, u_2]$, respectively and T' is obtained from T by adding a new object with time-bound expression as indicated in the table below.

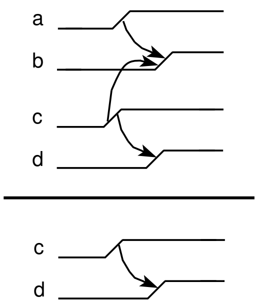
e_1	e_2	condition	new object	e_3
$=n$	$=k$	$n=k$	$(p_b, \{b_b, b_c\})$	
		$n<k$	(p_c, p_b)	$=k-n$
		$n>k$	(p_b, p_c)	$=n-k$
$=n$	$<k$	$n=k$	(p_b, p_c)	
		$n>k$	(p_b, p_c)	$[n-k+1, n-1]$
$=n$	$>k$	$n=k$	(p_c, p_b)	
		$n<k$	(p_c, p_b)	$<k-n$

Rule 19 If a time anchor expression does not relate to any other time anchor expression or open assumption, the anchor can be removed from the diagram. This rule corresponds to forall generalization in second-order logic.



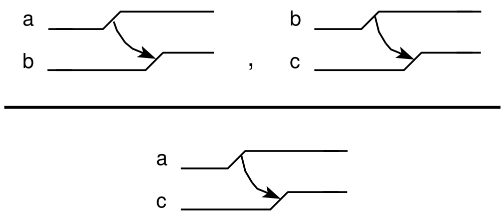
Timing diagram T' is derivable from timing diagram T by *time generalization* iff T' is obtained from T by removing a time anchor containing a free variable that is not free in any assumption used to derive T or in any other anchor expression in T.

Rule 20 A timing diagram interpreted under the invariant semantics that has more than one tail object derives any of its single-tail diagrams.



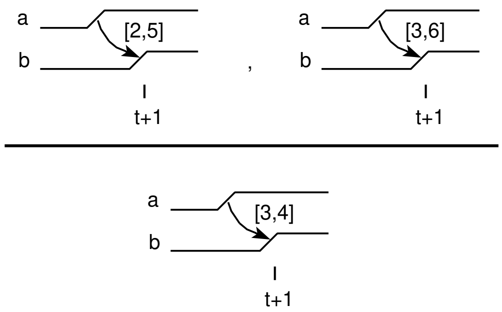
Timing diagram T' is derivable from timing diagram T by *single-tail elimination* iff T is a timing diagram interpreted under the invariant semantics and T' is a single-tail diagram arising from T.

Rule 21 Invariantly-interpreted timing diagrams give rise to transitive relationships between events. If one timing diagram guarantees event e_1 from event e_2, and another timing diagram guarantees event e_3 from e_2, then e_3 must be guaranteed from e_1.



Timing diagram T'' is derivable from timing diagrams T and T' by *composition* if T and T' are both interpreted invariantly, if the partition P_2 of T contains the same objects and constraints as partition P_1 of T', and T'' is interpreted invariantly and has partition P_1 of T and partition P_2 of T'.

Rule 22 A set of pairwise-event isomorphic timing diagrams that are interpreted invariantly under the same assume-guarantee partitioning can be merged into a single timing diagram reflecting the possible bounds from each diagram.



Timing diagram $T = \langle P, W, E_S, E_C, B, A\rangle$ can be derived from timing diagrams $T_1, \ldots, T_n$ by *bounds merge* iff T is event isomorphic (Defn. 52) to each T_i, T and each T_i are all interpreted invariantly under the same assume-guarantee partitioning, $T_1, \ldots, T_n$ are time-bound expression compatible (Defn. 53), and for each successor edge object e in E_S, the maximum lower bound l_m in a time-bound expression for e over all the T_i's is no larger than the minimum upper bound u_m in a time-bound expression for e over all the T_i's and T assigns time-bound expression $[l_m, u_m]$ to e in B.

4.5.1 Soundness

Establishing that an object-index assignment satisfies T iff it satisfies T' is sufficient in each of the following proofs. None of the rules in this section add new transition or level objects. As the iff condition requires that the particular object-index assignments located by the semantic definitions are not affected, this is sufficient to establish soundness.

Lemma 40 *The transitive edges rule is sound relative to both* $\models_{\mathrm{Inv}}$ *and* $\models_{\mathrm{Iter}}$.

Proof Let I be an object-index assignment that satisfies T. By Definition 63, $l_1 \leq I(b_2) - I(b_1) \leq u_1$ and $l_2 \leq I(b_3) - I(b_2) \leq u_2$. Solving each of these for $I(b_2)$ and substituting yields inequalities $l_1 + l_2 \leq I(b_3) - I(b_1) \leq u_1 + u + 2$. These inequalities are precisely those required for time bound expression $[l_1 + l_2, u_1 + u_2]$ to be satisfied on edge (b_1, b_3). If there is no time bound expression associated with either (b_1, b_2) or (b_2, b_3), the respective difference between the assigned indices is at least 1 by Definition 63. As a result any object-index assignment satisfying T must satisfy T'. Any object-index assignment that satisfies T' must satisfy T since the successor edge objects of T are a subset of those of T'.

□

Lemma 41 *The time label bounds rule is sound relative to both* $\models_{\mathrm{Inv}}$ *and* $\models_{\mathrm{Iter}}$.

Proof Let I be an object-index assignment that satisfies T and let ϕ be a function from temporal variables to natural numbers. By Definition 63, for every pair of time anchors (b_1, e_1) and (b_2, e_2), $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_1)[\phi] = I(b_2) - I(b_1)$. Adding a time anchor such that $\mathsf{TEval}(e_2)[\phi] - \mathsf{TEval}(e_1)[\phi] = I(b_2) - I(b_1)$ therefore does not affect whether I satisfies T'. In this case, any object-index assignment satisfying T' must satisfy T since the same time bound expressions appear in both diagrams.

□

Lemma 42 *The anchor align rule is sound relative to both* $\models_{\mathrm{Inv}}$ *and* $\models_{\mathrm{Iter}}$.

Proof Let T and T' be as assumed in the statement of the anchor align rule. Let w be any word that describes both T and T'. Then by Definition 71, w must satisfy objects b and b' at index e. w therefore satisfies T'' at index e by definition of satisfying a concurrent edge object (Defn. 62). w must also satisfy the remaining requirements on T'' since the remaining objects were either in T or in T', both of which were satisfied by w; the no common free variable restriction guarantees that no new constraints between events exist as a result of the merger.

□

Lemma 43 *The bounds slide rule is sound relative to both* $\models_{\text{Inv}}$ *and* $\models_{\text{Iter}}$.

Proof Successor edge objects and their bounds are checked relative to the time points they connect. Since this rule does not affect the time points connected by any successor edge object, the rule is sound.

□

Lemma 44 *The derived bounds rule is sound relative to both* $\models_{\text{Inv}}$ *and* $\models_{\text{Iter}}$.

Proof Let I be an object-index assignment that satisfies T. Each case is considered in turn:

1. By Definition 63, $l \leq I(b_2) - I(b_1) \leq u$ where $[l, u]$ is the time-bound object associated with (b_1, b_2). T' is identical to T with the exception of successor edge object (b_3, b_2) that also has time-bound expression $[l, u]$. $I(b_1) = I(b_3)$ by the assumptions in the lemma, so $l \leq I(b_2) - I(b_3) \leq u$. I must therefore satisfy T'. Any I that satisfies T' would have to satisfy T since the successor edge objects and time bound expressions of T are subsets of those of T'.

2. By Definition 63, $l \leq I(b_2) - I(b_1) \leq u$ where $[l, u]$ is the time-bound object associated with (b_1, b_2). T' is identical to T with the exception of successor edge object (b_1, b_3) that also has time-bound expression $[l, u]$. $I(b_2) = I(b_3)$ by the assumptions in the lemma, so $l \leq I(b_3) - I(b_1) \leq u$. I must therefore satisfy T'. Any I that satisfies T' would have to satisfy T since the successor edge objects and time bound expressions of T are subsets of those of T'.

3. Each line of the table must be considered in turn to show that I satisfies T'. Any I satisfying T' must satisfy T since all objects of T are objects of T'.

   (a) If $e_1 = n$ and $e_2 = n$, then p_b and p_c must be the same time point, so $I(p_b) = I(p_c)$. A concurrent edge object containing objects b_b and b_c may be introduced without affecting the satisfaction of T.

(b) If $e_1 = n$ and $e_2 = k$ where $n < k$, then $I(p_b) < I(p_c)$. A successor edge object (b_b, b_c) can therefore be added to T' without affecting satisfaction. Since $I(p_b) - I(p_a) = n$ and $I(p_c) - I(p_a) = k$, $I(p_c) - I(p_b) = k - n$.

(c) If $e_1 = n$ and $e_2 = k$ where $n > k$, then $I(p_c) < I(p_b)$. A successor edge object (b_c, b_b) can therefore be added to T' without affecting satisfaction. Since $I(p_b) - I(p_a) = n$ and $I(p_c) - I(p_a) = k$, $I(p_b) - I(p_c) = n - k$.

(d) If $e_1 = n$ and $e_2 < n$, then $I(p_c) < I(p_b)$; successor edge (b_c, b_b) can therefore be introduced without affecting the satisfaction of T.

(e) If $e_1 = n$, $e_2 < k$, and $k < n$, then $I(p_c) < I(p_b)$; successor edge (b_c, b_b) can therefore be introduced without affecting the satisfaction of T. In addition, $n - k + 1 \leq I(p_b) - I(p_c)$ and $I(p_b) - I(p_c) \leq n - 1$ since $e_2 < n$.

(f) If $e_1 = n$ and $e_2 > n$, then $I(p_b) < I(p_c)$. Successor edge (b_b, b_c) can therefore be introduced without affecting the satisfaction of T.

(g) If $e_1 = n$, $e_2 > k$, and $n < k$, then $I(p_b) < I(p_c)$. Successor edge (b_b, b_c) can therefore be introduced without affecting the satisfaction of T. In addition $I(p_c) - I(p_b) < k - n$.

4. This proof is similar to that of the previous case.

□

Lemma 45 *The time generalization rule is sound relative to both* $\models_{\mathrm{Inv}}$ *and* $\models_{\mathrm{Iter}}$.

Proof Let I be an object-index assignment that satisfies T. T' has fewer restrictions to check of I, but the same objects. I must therefore satisfy T'. If I is an object-index assignment that satisfies T', it must satisfy T because there was no relationship between the removed time anchor and any other, since the time anchor variable did not appear in any other anchor expression.

□

Lemma 46 *The single-tail elimination rule is sound relative to* $\models_{\mathrm{Inv}}$.

Proof This follows from Theorem 8, Chapter 3, Section 3.2.

□

Lemma 47 *The composition rule is sound relative to* $\models_{\text{Inv}}$.

Proof Let I be an object-index assignment that satisfies both T and T'. If I does not assign indices to all objects in P_1 or fails to satisfies some constraint on P_1 of T, then I satisfies T'' since P_1 of T'' is the same as P_1 of T by construction. If I does assign indices to all objects in P_1 that satisfy the constraints in P_1 of T, then by Definition 67, I assigns indices to all of T and those indices satisfy the constraints of T. Since P_1 of T' is contained in T, all objects and constraints in P_1 of T are therefore satisfied by I. I satisfies T' by assumption, so I satisfies P_2 of T''; I therefore satisfies T''.

□

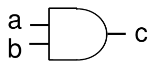
Lemma 48 *The bounds merge rule is sound relative to* $\models_{\text{Inv}}$.

Proof Let I be an object-index assignment that satisfies each T_i in $T_1, \ldots, T_n$. Since T and all the T_i's contain the same objects, I can only violate T in some time-bound expression. Let e be a successor edge object in T with time-bound expression $[l, u]$. By definition, l is the maximum of all the lower bounds for e assigned in some T_i and u is the minimum of the upper bounds for e assigned in some T_i. I must therefore respect both l and u unless $l > u$; the case when $l > u$ is disallowed by the definition of the rule, so I must satisfy T.

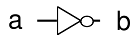
□

4.6 Between Circuit Diagrams and SOL

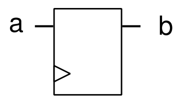
Rule 23 A circuit diagram can be translated at the gate level to a SOL expression using the expected correlation between the circuit diagram icons, the logical connectives, and the time variables.



$$\forall t((a(t) \wedge b(t)) \leftrightarrow c(t))$$



$$\forall t(a(t) \leftrightarrow \neg b(t))$$



$$\forall t(a(t) \leftrightarrow b(t+1))$$

SOL expression E is derivable from circuit diagram C by *observe* iff one of the following cases holds:

1. C contains an and gate with interface ports labeled a, b, and c, and E is $\forall t((a(t) \wedge b(t)) \leftrightarrow c(t))$.
2. C contains an inverter with interface ports labeled a and b, and E is $\forall t(a(t) \leftrightarrow \neg b(t))$.
3. C contains a unit-delay with interface ports labeled a and b, and E is $\forall t(a(t) \leftrightarrow b(t+1))$.

4.6.1 Soundness

Lemma 49 *The observe rule is sound on SOL expressions and circuit diagrams relative to* $\models_s$ *on circuit diagrams.*

Proof Let E be a SOL expression and let C be a circuit diagram such that $C \vdash E$. It must be shown that for any device D, if $D \models_s C$ then $D \models E$. Each case of the rule is proven in turn:

1. Assume E is $\forall t((a(t) \wedge b(t)) \leftrightarrow c(t))$. Then C contains an and gate with interface ports labeled a, b, and c. By Definition 28, since $D \models_s C$, the connection points labeled by a, b, and c must correspond to ports p_a, p_b, and p_c such that D contains $\mathsf{AndGate}(p_a, p_b, p_c)$. By the definition of AndGate (Defn. 2), given any assignment i, $i(p_c) = i(p_a) * i(p_b)$. It is clear, then that in any

assignment to the ports of D, $i(p_c) = 1$ iff both $i(p_a) = 1$ and $i(p_b) = 1$. It is straightforward to show that $D \models E$ under these circumstances.

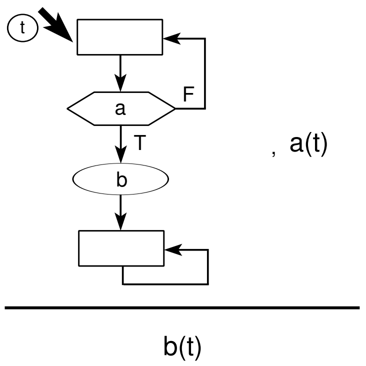
2. Assume E is $\forall t(a(t) \leftrightarrow \neg b(t))$. Then C contains an inverter with interface ports labeled a and b. By Definition 28, since $D \models_s C$, the connection points labeled by a and b must correspond to ports p_a and p_b such that D contains $\mathsf{Inverter}(p_a, p_b)$. By the definition of $\mathsf{Inverter}$ (Defn. 2), given any assignment i, $i(p_b) = 1 - i(p_a)$. It is straightforward to show that $D \models E$ under these circumstances.

3. Assume E is $\forall t(a(t) \leftrightarrow b(t+1))$. Then C contains a unit-delay with interface ports labeled a and b. By Definition 28, since $D \models_s C$, the connection points labeled by a and b must correspond to ports p_a and p_b such that D contains $\mathsf{UnitDelay}(p_a, p_b)$. By the definition of $\mathsf{UnitDelay}$ (Defn. 2), given any assignment sequence i, $i_{i+1}(p_b) = i(p_a)_i$. It is straightforward to show that $D \models E$ under these circumstances.

□

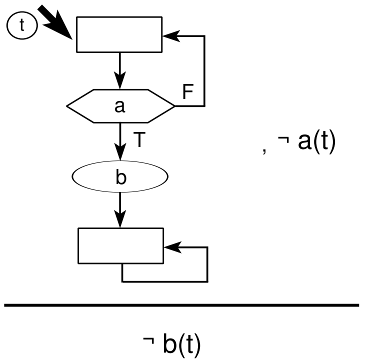
4.7 Between ASM Charts and SOL

Rule 24 Given an ASM chart with a current state marker, if the conditions required for an output condition are specified as SOL expressions, the SOL expression corresponding to that output condition can be inferred.



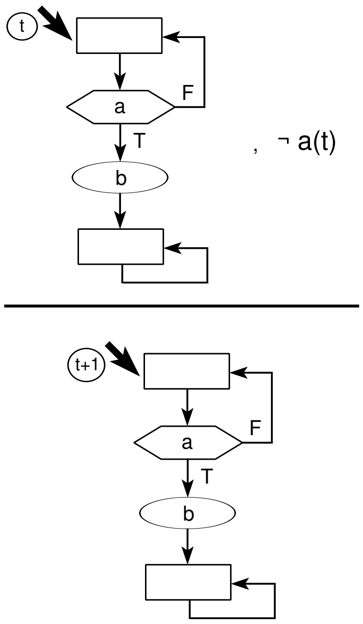
SOL expression $N(t)$ is derivable from time-annotated ASM chart M and SOL formulas $f_1, \ldots, f_n$ of the form $N(t)$ or $\neg N(t)$ for $N \in \mathsf{SigNames}$ and t the time-annotation of M by *output condition enabled* iff M contains a current state marker at state s containing discrete time expression t, and there exists a output condition for N satisfied by s and the assignment consistent with $f_1, \ldots, f_n$.

Rule 25 Given an ASM chart with a current state marker, if an external signal is not asserted in the marked state, or if an external signal is not asserted by a satisfied next-transition path, the negation of that signal is true in the current state.



SOL expression $\neg N(t)$ is derivable from time-annotated ASM chart M and a (possibly empty) set of SOL formulas $f_1, \ldots, f_n$ of the form $N(t)$ or $\neg N(t)$ for $N \in \mathsf{SigNames}$ and t the time-annotation of M by *output unenabled* iff M contains a current state marker at state s containing discrete time expression t, $f_1, \ldots, f_n$ satisfy some next state transition of M from s, and there does not exist an output condition for N satisfied by s and the assignment consistent with $f_1, \ldots, f_n$.

Rule 26 Given an ASM chart with a current state marker, if the conditions required for a next-state transition are specified as SOL expressions, the ASM with the appropriate next-state indicated by the current state marker can be inferred.

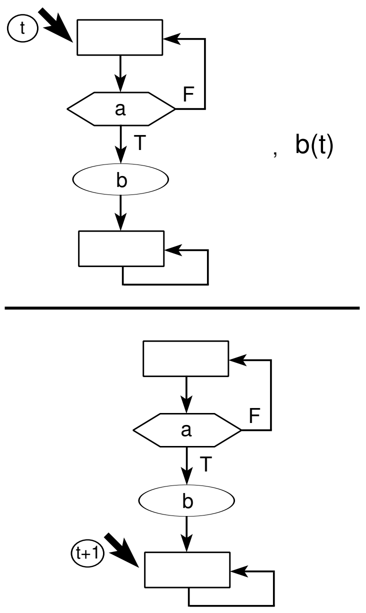


ASM chart M' is derivable from time-annotated ASM chart M and SOL expressions $f_1, \ldots, f_n$ of the form $N(t)$ or $\neg N(t)$ for $N \in \mathsf{SigNames}$ and t the time-annotation of M by *transition enabled* iff M has a current state marker at state s with discrete time label t, M' is structurally equivalent to M with a current state marker at state s' containing discrete time expression $t + 1$, and s' is the next state of s under the assignment consistent with $f_1, \ldots, f_n$.

Rule 27 Given an ASM with a current state marker, if there is a next-state transition that loops back to the current state, the ASM chart remains in that state as long as the conditions satisfying that transition are satisfied. This rule is derivable by repeated applications of the transition enabled rule (Rule 26).

ASM chart M' is derivable from time-annotated ASM chart M and SOL expressions $f_1, \ldots, f_n$ of the form $N(t)$ or $\neg N(t)$ for $N \in \mathsf{SigNames}$ by *Looping* if M and M' are structurally equivalent and in the same state s, there is a transition from s to itself satisfied by all assignments consistent with $f_1, \ldots, f_n$, and $f_1, \ldots, f_n$ hold at all moments of discrete time from that denoted in M to that denoted in M'.

Rule 28 Given a time-annotated ASM chart and an external signal asserted at the discrete time indicated in the current state marker, if that signal is asserted along only one transition path in the current state, the state at the end of that transition path must be the next state of the ASM chart. This rule is derivable from the output condition enabled rule (Rule 24), output unenabled rule (Rule 25), and transition enabled rule (Rule 26).



ASM chart M' is derivable from ASM chart M and SOL expression $N(t)$ by *output transition* if M is time-annotated with time t at state s, M and M' are structurally equivalent, M contains only one next-state transition path along which N is enabled and M' is time annotated with time $t+1$ in state s' where s' is the target state in the next-state transition path along which N is enabled.

4.7.1 Soundness

Lemma 50 *The output condition enabled rule is sound relative to run-describes on ASM charts.*

Proof Let M be a time-annotated ASM chart with time-annotation t at state cs, let $f_1, \ldots, f_n$ be SOL formulas, let D be a device and let $R_D = D_{Y_1}, \ldots, D_{Y_k}$ be a run of D.

Assume t contains no free variables. Assuming $R_D \models M$, $D \models_s M$ and there exist surjective state map τ_t and complete signal map τ_n that capture D and M by Definition 46. Assuming also that each f_i describes R_D, the state of D_{Y_t} is cs and the value of each signal covered by an f_i in D_{Y_t} is 0 if f_i is a negative literal and 1 if f_i is a positive literal. By assumption, the output condition for N is satisfied by cs and the values determined $f_1, \ldots, f_n$. By the definition of τ_t and τ_n capturing M and D (Definition 42), N must have value 1 in D_{Y_t}. $R_D \models N(t)$ by Definition 79.

If t contains free variables, let $v_1, \ldots, v_k$ denote the free variables in t and let $c_1, \ldots, c_k$ be temporal constants. The argument that for any $[c_1, \ldots, c_k]$, a run describes $N(t)[c_1, \ldots, c_k]$ when it describes $M[c_1, \ldots, c_k]$ and each $f_j[c_1, \ldots, c_k]$ is similar to that for the no free variables case.

$\square$

Lemma 51 *The output unenabled rule is sound relative to run-describes on ASM charts.*

Proof This proof is similar to the previous proof, noting that external signals not asserted in the current state are required to carry value 0 by Definition 42.

$\square$

Lemma 52 *The transition enabled rule is sound relative to run-describes on ASM charts.*

Proof Let M be a time-annotated ASM chart with time-annotation t at state cs, let $f_1, \ldots, f_n$ be SOL formulas, let D be a device and let $R_D = D_{Y_1}, \ldots, D_{Y_k}$ be a run of D.

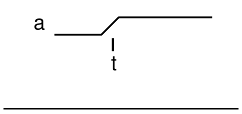
Assume t contains no free variables. Assuming $R_D \models M$, $D \models_s M$ and there exist surjective state map τ_t and complete signal map τ_n that capture D and M by Definition 46. Assuming also that each f_i describes R_D, the state of D_{Y_t} is cs and the value of each signal covered by an f_i in D_{Y_t} is 0 if f_i is a negative literal and 1 if f_i is a positive literal. By assumption, the values determined by $f_1, \ldots, f_n$ satisfy the next-state transition from state cs to state cs'. By the definition of τ_t and τ_n capturing M and D (Definition 42), $\tau_t(cs')$ must be the state of $D_{Y_{t+1}}$. $R_D \models M'$ by Definition 79.

If t contains free variables, let $v_1, \ldots, v_k$ denote the free variables in t and let $c_1, \ldots, c_k$ be temporal constants. The argument that for any $[c_1, \ldots, c_k]$, a run describes $M'[c_1, \ldots, c_k]$ when it describes $M[c_1, \ldots, c_k]$ and each $f_j[c_1, \ldots, c_k]$ is similar to that for the no free variables case.

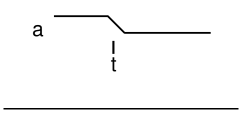
$\square$

4.8 Between Timing Diagrams and SOL

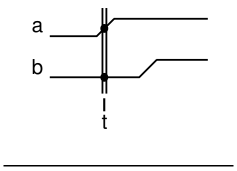
Rule 29 When a timing diagram has a time anchor, the values appearing on the waveforms at the anchor can be stated as SOL expressions over the names of the waveforms.



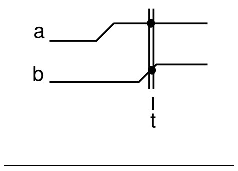
$\neg a(t) \wedge a(t+1)$



$a(t) \wedge \neg a(t+1)$



$\neg b(t)$



$a(t)$

SOL expression e is derivable from timing diagram T by *Observe* iff one of the following cases holds:

1. T contains a time anchor with label t to a rising transition object on waveform a and e is $\neg a(t) \wedge a(t+1)$.

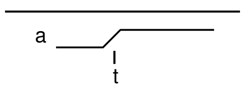
2. T contains a time anchor with label t to a falling transition object on waveform a and e is $a(t) \wedge \neg a(t+1)$.

3. T contains a time anchor with label t to a low-level object on waveform a and e is $\neg a(t)$.

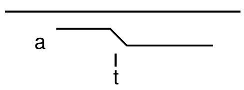
4. T contains a time anchor with label t to a high-level object on waveform a and e is $a(t)$.

Rule 30 If SOL expressions indicate that a transition occurs on a given signal at a given time, that transition can be depicted in an anchored $\mathrm{TD}^{\mathrm{TI}}$timing diagram.

$\neg a(t) \wedge a(t+1)$



$a(t) \wedge \neg a(t+1)$



$\mathrm{TD}^{\mathrm{TI}}$timing diagram T is derivable from SOL expression e by *Apply* iff one of the following cases holds:

1. e is $\neg a(t) \wedge a(t+1)$ and T is in $\mathrm{TD}^{\mathrm{TI}}$and consists of a rising transition object for on signal a with a time anchor labeled t on the rising transition object.

2. e is $a(t) \wedge \neg a(t+1)$ and T is in $\mathrm{TD}^{\mathrm{TI}}$and consists of a falling transition object for on signal a with a time anchor labeled t on the falling transition object.

4.8.1 Soundness

Lemma 53 *The observe rule is sound between timing diagrams and SOL relative to both* $\models_{\mathrm{Inv}}$ *and* $\models_{\mathrm{Iter}}$.

Proof

Let T be a timing diagram containing a time anchor (b, t). Assume t contains no free variables. Let D be a device with ports P. Let R_D be any run of device D such that $\mathcal{W}(R_D)_P$ anchor describes T. By Definition 71, $\mathcal{W}(R_D)_P$ describes T and has $\mathcal{W}(R_D)_P(t)$ satisfying b. The assignment in D_{Y_t}, the t^{th} dynamic device in R_D, is consistent with the requirements for $\mathcal{W}(R_D)_P(t)$ satisfying b by the definition of $\mathcal{W}(R_D)_P$ (Defn. 19). Specifically:

1. If b is a rising transition object for waveform N, the assignment in D_{Y_t} assigns 0 to N and the assignment in $D_{Y_{t+1}}$ assigns 1 to N.

2. If b is a falling transition object for waveform N, the assignment in D_{Y_t} assigns 1 to N and the assignment in $D_{Y_{t+1}}$ assigns 0 to N.

3. If b is a low-level object for waveform N, the assignment in D_{Y_t} assigns 0 to N.

4. If b is a high-level object for waveform N, the assignment in D_{Y_t} assigns 1 to N.

These cases are sufficient to establish that R_D must describe derived SOL expression e.

□

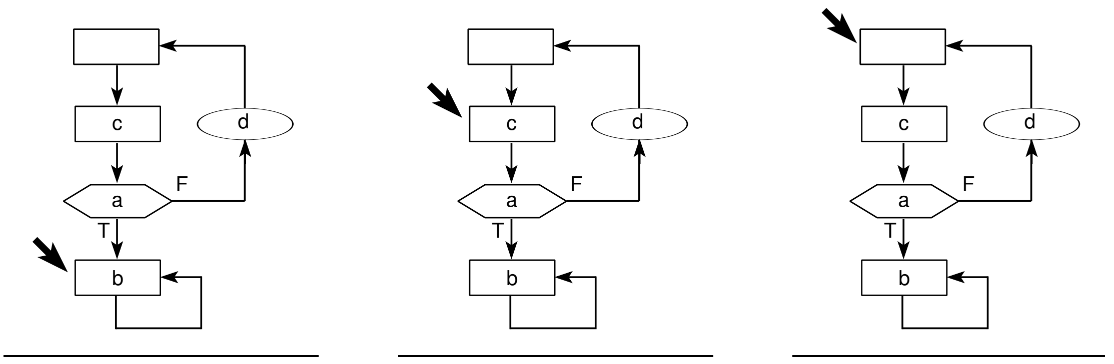
Lemma 54 *The apply rule is sound between timing diagrams and SOL relative to both* $\models_{\text{Inv}}$ *and* $\models_{\text{Iter}}$.

Proof If e is SOL expression $\neg a(t) \wedge a(t+1)$, then in any run $R_D = \langle D_{Y_0}, D_{Y_1}, \ldots \rangle$ of a device D such that e describes R_D, dynamic device D_{Y_t} in R_D must assign 0 to a and dynamic device $D_{Y_{t+1}}$ must assign 1 to a. $\mathcal{W}(R_D)$ must be consistent with these assignments by the definition of $\mathcal{W}(R_D)_P$ (Defn. 19). $\mathcal{W}(R_D)$ must be described by T by Definition 71. The argument for the other case is similar.

□

4.9 Between ASM Charts and CTL

Rule 31 These rules allow CTL formulas to be read from grounded ASM charts.



$\mathsf{AG}\ b$ $\qquad$ $c \wedge \mathsf{EX}\ b$ $\qquad$ $\mathsf{EF\ AG}\ b$

Let M be an ASM chart with a time-free current state marker attached to state t and let φ be a CTL formula. Let $p \in \mathsf{SigNames}$. φ is derivable from M by *CTL observe* iff one of the following cases holds:

1. φ is a boolean expression over only internal signal names of M that holds along every next-state transition from state t.

2. $\varphi = \mathsf{EX}\psi$ where ψ is derivable by CTL observe from M in state t', where there exists a next state transition in M from state t to state t'.

3. $\varphi = \mathsf{AX}\psi$ where ψ is derivable by CTL observe from M in every state t' such that there exists a next state transition in M from state t to state t'.

4. $\varphi = \mathsf{E}[\psi \mathsf{U} \gamma]$ and either

   (a) γ is derivable by CTL observe in state t

   (b) ψ is derivable by CTL observe in state t and $\mathsf{E}[\psi \mathsf{U} \varphi]$ is derivable by CTL observe in some next-state t' of t.

5. $\varphi = \mathsf{A}[\psi \mathsf{U} \gamma]$ and either

(a) γ is derivable by CTL observe in state t

(b) ψ is derivable by CTL observe in state t and $\mathsf{A}[\psi \mathsf{U} \gamma]$ is derivable by CTL observe in every next-state t' of t.

6. $\varphi = \neg\mathsf{E}[\psi \mathsf{U} \gamma]$ and either

   (a) $\neg\gamma$ is derivable by CTL observe in every state t' reachable from t

   (b) on every transition path from t to a state from which γ is derivable by CTL observe, there exists a state t'' such that t'' appears before the first state t' from which γ is derivable by CTL observe and $\neg\psi$ is derivable by CTL observe from state t''.

7. $\varphi = \neg\mathsf{A}[\psi \mathsf{U} \gamma]$ and either

   (a) there exists some infinite transition path from t such that $\neg\gamma$ is derivable by CTL observe in every state along the path

   (b) on some transition path from t to a state from which γ is derivable by CTL observe, there exists a state t'' such that t'' appears before the first state t' from which γ is derivable by CTL observe and $\neg\psi$ is derivable by CTL observe from state t''.

4.9.1 Soundness

Lemma 55 *CTL observe is sound relative to $\models_{\text{fs}}$ on ASM charts.*

Proof Let D be a device and assume $D \models_{\text{fs}} M$. Let K_D is the derived Kripke structure for D (Defn. 76). By the definition of $\models_{\text{fs}}$ (Defn. 42), there exists an ASM chart M' such that $D \models_s M'$ and M is the functional reduction of M'. By the definition of functional reduction (Defn. 41), every transition and output condition in M must be in M'. The definition of $\models_s$ (Defn. 42) essentially states that every next-state transition and output condition of M' is mimiced in D. Assume M' (and hence D) is in the state t indicated by the current-state marker. The proof proceeds by cases on φ. In each case, it suffices to prove that $(K_D, t) \models \varphi$, since that establishes $D \models \varphi$ by Definition 77).

1. Assume φ is a boolean expression over internal signal names of M that holds along every next-state transition from state t. By construction, φ is therefore true in every state (t, a) of K_D, where a is an input assignment for D, so $(K_D, t) \models \varphi$.

2. $\varphi = \mathsf{EX}\psi$ where ψ is derivable by CTL observe from M in state t', where there exists a next-state transition in M from state t to state t'. Let a be an assignment sequence such that a and t satisfy the next-state transition to t'. By construction, K_D contains a state corresponding to (t, a). Also by construction, K_D contains a transition from (t, a) to (t', a'), where a' is any input assignment for D. Since ψ holds in t' by assumption, $(K_D, t) \models \mathsf{EX}\psi$ by the semantics of CTL (Defn. 77).

3. $\varphi = \mathsf{AX}\psi$ where ψ is derivable by CTL observe from M in every state t' such that there exists a next state transition in M from state t to state t'. Let t' be any such state and let a be an assignment sequence such that a and t satisfy the next-state transition to t'. By construction, K_D contains a state corresponding to (t, a). Also by construction, K_D contains a transition from (t, a) to (t', a'), where a' is any input assignment for D. Since ψ holds in t' by assumption, $(K_D, t) \models \mathsf{AX}\psi$ by the semantics of CTL (Defn. 77).

4. $\varphi = \mathsf{E}[\psi \mathsf{U} \gamma]$ and either

   (a) γ is derivable by CTL observe in state t

   (b) ψ is derivable by CTL observe in state t and $\mathsf{E}[\psi \mathsf{U} \gamma]$ is derivable by CTL observe in some next-state t' of t.

   Assume γ is derivable by CTL observe in state t. Then for every state (t, a) in K_D where a is an input assignment for D, $(K_D, t) \models \gamma$. As a result, $(K_D, t) \models \mathsf{E}[\psi \mathsf{U} \gamma]$ by the semantics of CTL.

   Assume ψ is derivable by CTL observe in state t and $\mathsf{E}[\psi \mathsf{U} \gamma]$ is derivable by CTL observe in some next-state t' of t. By an inductive argument, there must exist some chain of states $t, t_1, \ldots, t_n$ such that $t_1 = t'$, γ is derivable by CTL observe in state t_n, and ψ is derivable by CTL observe in each state $t, t_1, \ldots, t_{n-1}$. By the semantics of CTL, $(K_D, t_n) \models \gamma$, $(K_D, t) \models \psi$, and $(K_D, t_i) \models \psi$ for all $1 \leq i < n$, so $(K_d, t) \models \mathsf{E}[\psi \mathsf{U} \gamma]$.

5. $\varphi = \mathsf{A}[\psi \mathsf{U} \gamma]$ and either

   (a) γ is derivable by CTL observe in state t

   (b) ψ is derivable by CTL observe in state t and $\mathsf{A}[\psi \mathsf{U} \gamma]$ is derivable by CTL observe in every next-state t' of t.

   Assume γ is derivable by CTL observe in state t. Then for every state (t, a) in K_D where a is an input assignment for D, $(K_D, t) \models \gamma$. Therefore, $(K_D, t) \models \mathsf{A}[\psi \mathsf{U} \gamma]$ by the semantics of CTL.

   Assume ψ is derivable by CTL observe in state t and $\mathsf{A}[\psi \mathsf{U} \gamma]$ is derivable by CTL observe in every next-state t' of t. Let t' be any next state of t. An inductive argument shows there must exist some chain of states $t, t_1, \ldots, t_n$ such that $t_1 = t'$, γ is derivable by CTL observe in state t_n, and ψ is derivable by CTL observe in each state $t, t_1, \ldots, t_{n-1}$. By the semantics of CTL, $(K_D, t_n) \models \gamma$, $(K_D, t) \models \psi$, and $(K_D, t_i) \models \psi$ for all $1 \leq i < n$, so $(K_D, t) \models \mathsf{A}[\psi \mathsf{U} \gamma]$.

6. $\varphi = \neg \mathsf{E}[\psi \mathsf{U} \gamma]$ and either

   (a) $\neg\gamma$ is derivable by CTL observe in every state t' reachable from t

   (b) on every transition path from t to a state from which γ is derivable by CTL observe, there exists a state t'' such that t'' appears before the first state t' from which γ is derivable by CTL observe and $\neg\psi$ is derivable by CTL observe from state t''.

   Assume $\neg\gamma$ is derivable by CTL observe in every state t' reachable from t. Then there does not exist a state t' reachable from t such that γ is derivable from t' by CTL observe. Therefore, there does not exist a state (t', a') reachable from any state (t, a) in K_D such that $(K_D, t') \models \gamma$. As a result, $(K_D, t) \models \neg \mathsf{E}[\psi \mathsf{U} \gamma]$.

   Assume that on every transition path from t to a state from which γ is derivable by CTL observe, there exists a state t'' such that t'' appears before the first state t' from which γ can be derived by CTL observe and $\neg\psi$ is derivable by CTL observe from state t''. Then there does not exist a sequence of states in K_D from any state (t, a) such that $(K_D, t'') \models \gamma$ and $(K_D, t') \models \psi$ for all states t' from t to t'', so $(K_D, t) \models \neg \mathsf{E}[\psi \mathsf{U} \gamma]$ by the CTL semantics.

7. $\varphi = \neg\mathsf{A}[\psi\mathsf{U}\gamma]$ and either

   (a) there exists some infinite transition path from t such that $\neg\gamma$ is derivable by CTL observe in every state along the path

   (b) on some transition path from t to a state from which γ is derivable by CTL observe, there exists a state t'' such that t'' appears before the first state t' from which γ is derivable by CTL observe and $\neg\psi$ is derivable by CTL observe from state t''.

   Assume there exists some infinite transition path from t such that $\neg\gamma$ is derivable by CTL observe in every state along the path. Then for every state t' along that path, $(K_D, t') \models \neg\gamma$, so $(K_D, t) \models \neg\mathsf{A}[\psi\mathsf{U}\gamma]$.

   Assume that on some transition path from t to a state from which γ is derivable by CTL observe, there exists a state t'' such that t'' appears before the first state t' from which γ can be derived by CTL observe and $\neg\psi$ is derivable by CTL observe from state t''. Let $t_1, \ldots, t_n$ denote this path, where $t_1 = t$ and $t_n = t'$. Assume $t_i = t''$. Then $(K_D, t_i) \models \neg\psi$ and $(K_D, t_j) \models \neg\gamma$ for some $j < i$. Therefore, by the CTL semantics, $(K_D, t) \models \neg\mathsf{A}[\psi\mathsf{U}\gamma]$.

□

4.10 Between Timing Diagrams and CTL

Rule 32 Section 3.1 of Chapter 3 presents a translation algorithm from a subset of timing diagrams into CTL. The CTL translation of any timing diagram can be inferred from that timing diagram. Sample instances of the rule appear in Examples 27, 28, and 29 in Chapter 3.

CTL formula F is derivable from timing diagram T by *CTL Observe* iff F is derivable from CTL_T (Defn. 91).

4.10.1 Soundness

Lemma 56 *The CTL observe rule is sound relative to* $\models_{\mathrm{Inv}}$.

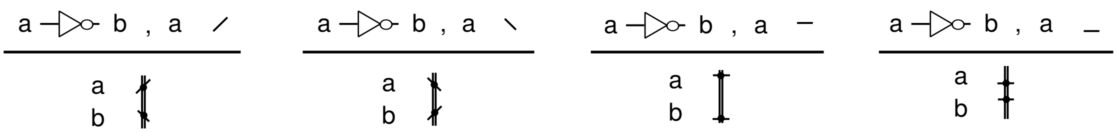
Proof This follows from Theorem 4, Section 3.1.

□

4.11 Between Circuit Diagrams and Timing Diagrams

The rules in this section allow a timing diagram to be constructed from a circuit diagram. The rules require that the construction be based on some initial waveform. In each of these rules, the inputs to the circuit element are the initial waveforms. Additional rules could be derived from these to allow the outputs to be the initial waveforms. There is one rule for each basic circuit component in the logic.

Rule 33



Timing diagram T' is derivable from timing diagram T and circuit diagram C by *inverter timing* iff C contains an inverter (p_a, p_b) with names a and b associated with p_a and p_b, respectively, T contains a waveform for a, T does not contain a waveform for b, and T' is obtained from T by adding a waveform for b and concurrent edge objects defined as follows:

1. For each high-level object (p_1, p_2) in a, add new low-level object (p_1, p_2) to b.
2. For each low-level object (p_1, p_2) in a, add new high-level object (p_1, p_2) to b.

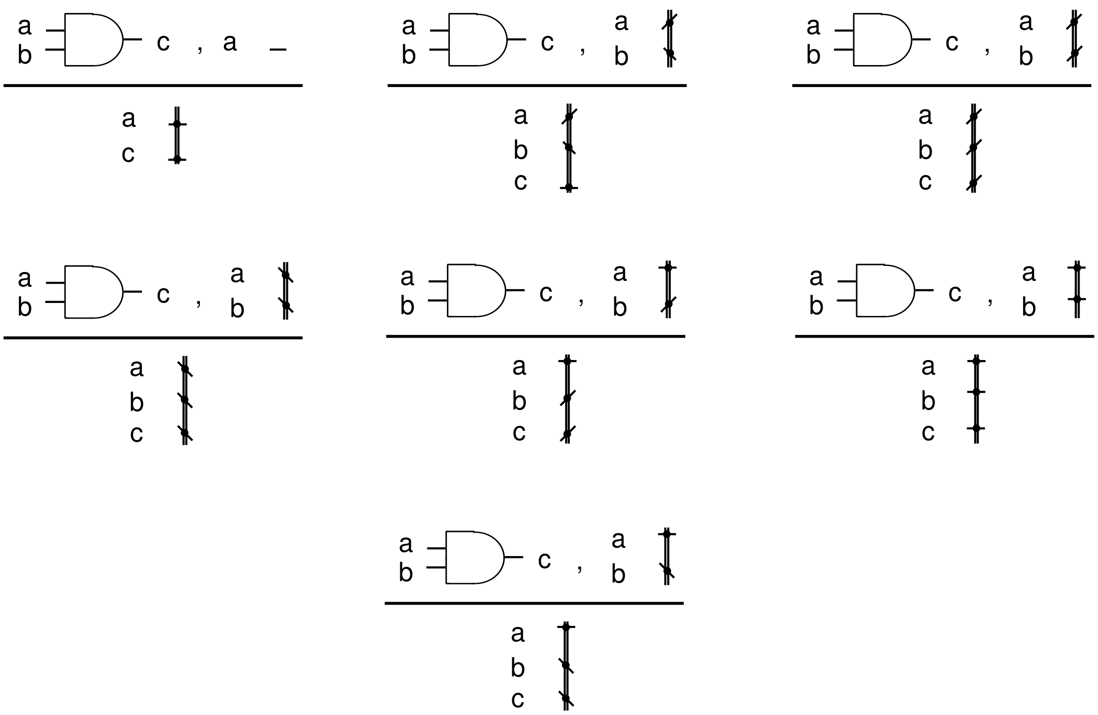
3. For each unknown-level object (p_1, p_2) in a, add new unknown-level object (p_1, p_2) to b.

4. For each rising transition object p_1 in a, add new falling transition object p_1 to b.

5. For each falling transition object p_1 in a, add new rising transition object p_1 to b.

6. For each time point at there is a transition object on a, add a concurrent edge object that involves both a and b.

Rule 34



Timing diagram T' is derivable from timing diagram T and circuit diagram C by *and gate timing* iff C contains an and gate (p_a, p_b, p_c) with names a, b, and c associated with p_a, p_b, and p_c, respectively, T contains waveforms for a and b, T does not contain a waveform for c, and T' is obtained from T by adding a waveform for c and concurrent edge objects defined as follows:

1. For each time point at which a has a low-level object, c has a low-level object.

2. For each time point at which a has a rising transition object and b has a falling transition object, c has a low-level object.

3. For each time point at which a and b both have rising transition objects, c has a rising transition object.

4. For each time point at which a and b both have falling transition objects, c has a falling transition object.

5. For each time point at which a has a high-level object and b has a rising transition object, c has a rising transition object.

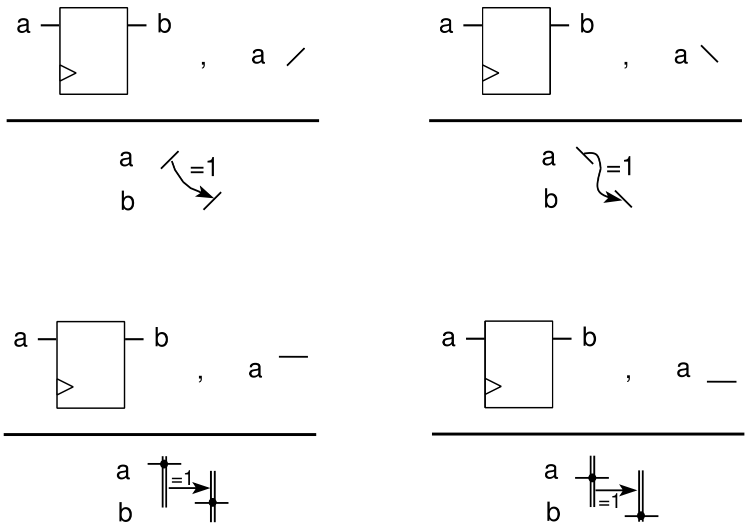
6. For each time point at which a and b both have high-level objects, c has a high-level object.

7. For each time point at which a has a high-level object and b has a falling transition object, c has a falling transition object.

8. For each time point not covered under one of the above definitions, c has an unknown-level object.

9. For each time point at which a concurrent edge object involves both a and b, that object is extended to include c.

Rule 35



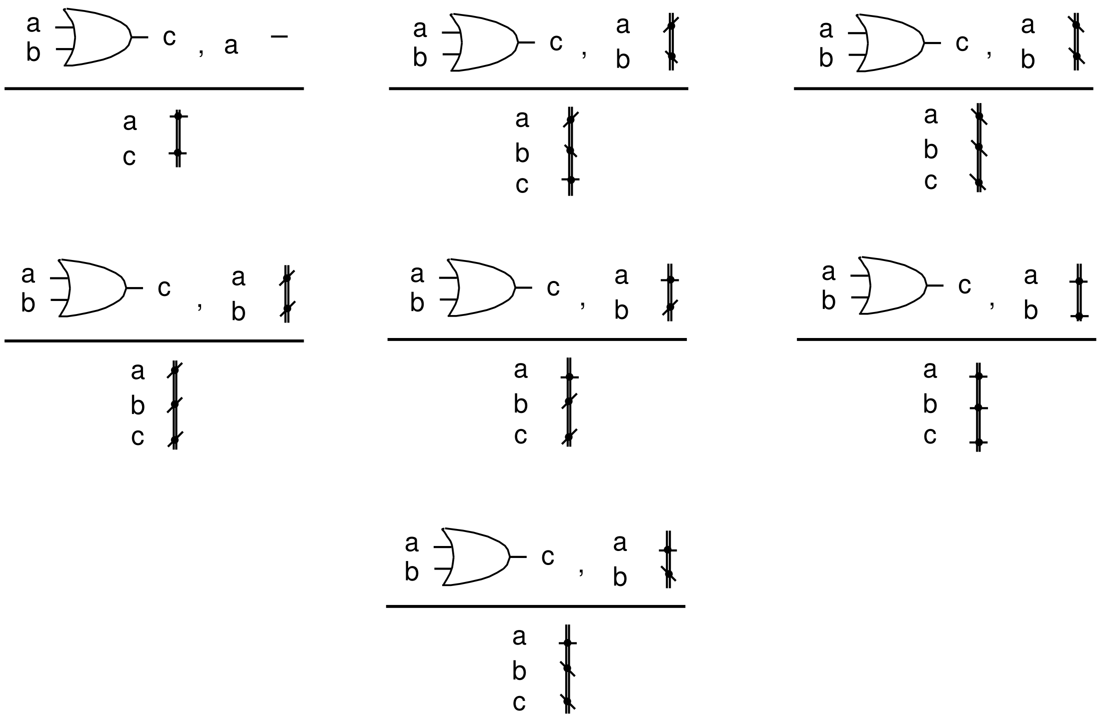
Timing diagram T' is derivable from timing diagram T and circuit diagram C by *unit delay timing* iff C contains a unit delay element (p_a, p_b) with names a and b associated with p_a and p_b, respectively, T contains a waveform for a, T does not contain a waveform for b, and T' is obtained from T by adding a waveform for b and concurrent edge objects and successor edge objects defined as follows:

1. For each time point p_i larger than the minimal point, b has the same object at p_i as a had at p_{i-1}.

2. b has an unknown-level object at the minimal point.

3. For each time point p_i at which a has a transition object, there exists a successor edge object from that transition object to the transition object on b at p_{i+1} with duration marker $= 1$.

Rule 36 The or gate rules is derivable from the rules for inverters, and gates, and the definition of the or component (Defn. 7).



Timing diagram T' is derivable from timing diagram T and circuit diagram C by *or gate timing* iff C contains an or gate (p_a, p_b, p_c) with names a, b, and c associated with p_a, p_b, and p_c, respectively, T contains waveforms for a and b, T does not contain a waveform for c, and T' is obtained from T by adding a waveform for c and concurrent edge objects defined as follows:

1. For each time point at which a has a high-level object, c has a high-level object.

2. For each time point at which a has a falling transition object and b has a rising transition object, c has a high-level object.

3. For each time point at which a and b both have falling transition objects, c has a falling transition object.

4. For each time point at which a and b both have rising transition objects, c has a rising transition object.

5. For each time point at which a has a low-level object and b has a falling transition object, c has a falling transition object.

6. For each time point at which a and b both have low-level objects, c has a low-level object.

7. For each time point at which a has a low-level object and b has a rising transition object, c has a rising transition object.

8. For each time point not covered under one of the above definitions, c has an unknown-level object.

9. For each time point at which a concurrent edge object involves both a and b, that object is extended to include c.

4.11.1 Soundness

Lemma 57 *The inverter timing rule is sound relative to $\models_s$ on circuit diagrams and both $\models_{\text{Inv}}$ and $\models_{\text{Iter}}$ on timing diagrams.*

Proof Let D be a device such that $D \models_s C$ and $\mathcal{L}(D) \models T$. C contains $\mathsf{InverterIcon}(p_a, p_b)$ by assumption. By the definition of $\models_s$ (Defn. 28), D must contain $\mathsf{Inverter}(\sigma(p_a), \sigma(p_b))$, where σ is the depiction map used for $\models_s$. Any assignment for D must reflect the relationship $\sigma(p_b) = 1 - \sigma(p_a)$ by the definition of an $\mathsf{Inverter}$ (Defn. 2); every letter in $\mathcal{L}(D)$ must therefore have either $a = 0$ and $b = 1$ or $a = 1$ and $b = 0$.

$\mathcal{L}(D) \models T$ by assumption. Let I be an object-index assignment over word W that satisfies T. The only differences between T and T' is that T' contains an additional waveform for b and some new concurrent edge objects. By construction, it is clear that the object on b at each time point is the opposite object to that on a at that time point. By the definition of satisfies (Defn. 63), I must assign opposite values to opposite type objects: for example, a high-level object must be assigned 1 and a low-level object must be assigned 0. Since W must have either $a = 0$ and $b = 1$ or $a = 1$ and

$b = 0$ in each index, I must satisfy the signal patterns of T'. The concurrent edge objects added do not affect I since they are introduced only at transition points and those indices have already been shown to satisfy the signal patterns.

□

Lemma 58 *The and gate timing rule is sound relative to $\models_s$ on circuit diagrams and both $\models_{\mathrm{Inv}}$ and $\models_{\mathrm{Iter}}$ on timing diagrams.*

Proof Let D be a device such that $D \models_s C$ and $\mathcal{L}(D) \models T$. C contains $\mathsf{AndGateIcon}(p_a, p_b, p_c)$ by assumption. By the definition of $\models_s$ (Defn. 28), D must contain $\mathsf{AndGate}(\sigma(p_a), \sigma(p_b), \sigma(p_c))$, where σ is the depiction map used for $\models_s$. Any assignment for D must reflect the relationship $\sigma(p_c) = \sigma(p_a) * \sigma(p_b)$ by the definition of an $\mathsf{AndGate}$ (Defn. 2). Every letter in $\mathcal{L}(D)$ must therefore have $c = 1$ iff $a = 1$ and $b = 1$.

$\mathcal{L}(D) \models T$ by assumption. Let I be an object-index assignment over word W that satisfies T. The only differences between T and T' is that T' contains an additional waveform for c and some additional concurrent edge objects. Given the values required by the satisfaction relationship for each type of level and transition object, it is straightforward to verify that the objects added to T' respect the restriction that $c = 1$ iff $a = 1$ and $b = 1$. I must therefore satisfy the signal patterns of T'. The concurrent edge objects added do not affect I since they are introduced only at transition points and those indices have already been shown to satisfy the signal patterns.

□

Lemma 59 *The unit delay timing rule is sound relative to $\models_s$ on circuit diagrams and both $\models_{\mathrm{Inv}}$ and $\models_{\mathrm{Iter}}$ on timing diagrams.*

Proof Let D be a device such that $D \models_s C$ and $\mathcal{L}(D) \models T$. C contains $\mathsf{UnitDelayIcon}(p_a, p_b)$ by assumption. By the definition of $\models_s$ (Defn. 28), D must contain $\mathsf{UnitDelay}(\sigma(p_a), \sigma(p_b))$, where σ is the depiction map used for $\models_s$. Any run for D must reflect the relationship $\sigma(p_b)(i+1) = \sigma(p_a)(i)$ by the definition of UnitDelay (Defn. 2).

$\mathcal{L}(D) \models T$ by assumption. Let I be an object-index assignment over word W that satisfies T. The only difference between T and T' is that T' contains an additional waveform for b. By construction, at each time point of T', b contains the same type of object that was on a at the previous time point. Hence, the signal patterns of T' are all restricted by relationship $\sigma(p_b)(i+1) = \sigma(p_a)(i)$. T' has an unknown level object on b at p_0, so any value for b at p_0 satisfies T'. The new successor edge objects added to T' from T are from the transition objects on a to their corresponding transition objects on b. Relationship $\sigma(p_b)(i+1) = \sigma(p_a)(i)$ assures that those objects meet their time bound of =1.

□

4.12 CTR Rules

Rule 37 CTR respects several rules of algebraic manipulation over processes; the operators are defined in Chapter 2, Section 2.2.6.

$$\frac{A}{A+B} \qquad \frac{A \parallel B}{B \parallel A} \qquad \frac{A+B}{B+A} \qquad \frac{(A;B) \parallel C}{A \parallel C; B \parallel C} \qquad \frac{(A+B) \parallel C}{A \parallel C + B \parallel C} \qquad \frac{(A+B);C}{A;C+B;C}$$

4.12.1 Soundness

Lemma 60 *The CTR rules are sound.*

Proof These follow trivially from the definitions in Chapter 2, Section 2.2.6.

□

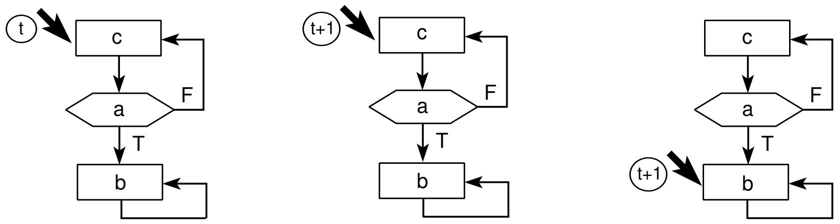


Figure 4.1: Diagrams to demonstrate the methods of proof.

4.13 Methods of Proof

Methods of proof are reasoning steps that apply to information in any representation; accordingly, they are defined relative to the language model. They support operations such as breaking into cases and locating information common to a set of cases.

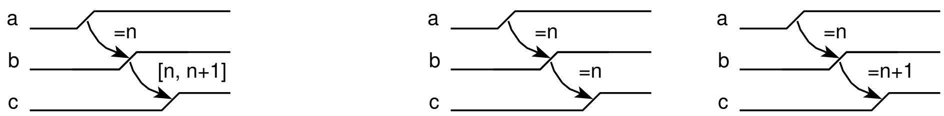
Inference rules are only useful in proof-checkers if testing whether they are applied correctly is decidable. All of the inference rules presented in previous sections are described as simple, decidable, syntactic transformations. Methods of proof, however, are defined as semantic operations, rather than syntactic operations. Their decidability relies on the decidability of the language operations they employ. In particular, they rely on the decidability of language emptiness and containment as well as on the effective constructability of language union and intersection. For regular languages, these problems are decidable and constructable, respectively. Timing diagrams and CTR are the two non-regular HHL representations; the variant of SOL supported in monadic, which is equivalent to the regular languages [81]. Section 3.3.1 argues that timing diagrams languages are always non-empty. Section 3.3.2 discussed the decidability of language containment for timing diagrams. Whether unions and intersections of timing diagram languages are effectively constructable are still open questions. All of these rules rely on more than just language emptiness, so the rules are only applicable to circuit diagrams, ASM charts, SOL formulas, CTL formulas, regular timing diagrams, and regular CTR formulas; these formulas are called *regular*.

Rule 38 A proof can be split into subcases at a formula F if a set of formulas $F_1, \ldots, F_n$ is identified such that every word in the language of F is in the language of some F_i; each F_i serves as the first formula in some subcase. For example, in Figure 4.1, the rightmost two ASM charts enumerate all possible next states of the leftmost ASM chart.

Let $F, F_1, \ldots, F_n$ be regular formulas. $F_1, \ldots, F_n$ *exhaust* F iff the language of F is contained in the union of the languages of the F_i's.

One particular case of *exhaust* is supported on non-regular timing diagrams. The *bounds merge* rule (Rule 22) indicates when a set of timing diagrams collectively represent the bounds of a timing diagram to which they are event isomorphic (Defn. 52). Conversely, this rule identifies instances when a set of timing diagrams collectively imply the bounds of a timing diagram to which they are event-isomorphic.

Example 36 The two timing diagrams on the right exhaust the bounds of the timing diagram on the left.



Rule 39 The dual of splitting into exhaustive cases is merging results obtained from individual cases. A formula is derived by merging cases iff its language is contained in the intersection of results obtained from a set of exhaustive formulas. Formula $b(t+1) \vee c(t+1)$ is derivable by merge on the two rightmost ASM charts in Figure 4.1.

Let $F_1, \ldots, F_n$ exhaust some formula F. Let $F_1', \ldots, F_n'$ be formulas such that each F_i' derives from F_i. Formula F' is derived from $F_1', \ldots, F_n'$ by *exhaustive case merge* iff the language of F' is contained in the intersection of the languages of the F_i''s.

Similarly, there is a bounds merge supported for non-regular timing diagrams through the *bounds* *merge* inference rule (Rule 22).

Rule 40 HHL, like *Hyperproof*, supports proofs of nonconsequence. Nonconsequence results hold when two scenarios are consistent with the given information, one which supports the desired result and one that does not. Such proofs rely on the ability to state when all of the assumptions made up until a particular point are satisfiable. In terms of language containment, this corresponds to testing that the language formed by intersecting the open assumptions is non-empty. If the leftmost diagram in Figure 4.1 is assumed, and the middle and right diagrams are subsequently assumed, each in a separate subcase, CTA (for check truth of assumptions) would hold of either the middle or the right diagram because the open assumptions are consistent. These two CTA results could establish that formula $c(t+1)$ is not a consequence of the leftmost ASM chart, since there exists one scenario in which it is true (the middle diagram) and one in which it is false (the rightmost diagram).

Let $F_1, \ldots, F_k$ be formulas. CTA is valid for $F_1, \ldots, F_k$ if the intersection of their languages is non-empty.

Rule 41 Cases that are shown to be inconsistent do not need to be considered further in a proof. In HHL, a case is inconsistent if its language is empty. Any case in which both the middle and right ASM charts in Figure 4.1 were asserted would close, since a chart can have only one current state at any particular time.

Close is valid for a set of formulas $F_1, \ldots, F_n$ if the language of their intersection is empty.

Rule 42 It is often possible to find a simpler representation for the language corresponding to an HHL formula. In this case, the new formula is a simplification of the original. In the language model, simplification is equivalent to an assertion that the languages of the two formulas are equivalent; each algebra rule is an instance of simplification. This rule is included to support assertions of language equivalence not captured by inference rules.

Formula F' is obtained from formula F by *simplification* if the language of F is contained in the language of F'.

Although the methods are not decidable for non-regular timing diagrams, there are probabilitic algorithms for querying whether a method applies to a given situation. Chapter 3 constructs a 2DCM M_T given timing diagram T. Containment is decidable for 2DCM and unions and intersections of 2DCM are effectively constructable [48]. The methods can therefore be tested against the partial timing diagram languages accepted by M_T. For example, suppose T_1 and T_2 are non-regular timing diagrams. Consider the intersection of M_{T_1} and M_{T_2}. If this intersection is non-empty, *CTA* must be valid for T_1 and T_2 since the languages accepted by each M_{T_i} is contained in the language of T_i. If the intersection is empty, no information can be determined regarding *CTA*. Similar techniques are applicable to probabilitically test *close* and to search for counterexamples to *exhaust* and *merge*.

4.14 Proofs

HHL supports natural deduction proofs, defined in the style of Prawitz [69] and Eberle [27]. Intuitively, a proof is a sequence of valid formulas, each of which is derivable from earlier formulas in the sequence. Checking whether a particular sequence of formulas represents a valid proof involves examining the order of the formulas and their nesting levels in subproofs. As per Eberle [27], strings of natural numbers capture nesting information, as demonstrated in Example 37.

Example 37 Each line in the Fitch-style proof on the left is represented by the corresponding string on the right.

F_1	1*	F_1
F_2	1*	F_2
F_3	1	F_3
F_4	1,1*	F_4
F_5	1,1	F_5
F_6	1,2*	F_6
F_7	1,2	F_7
F_8	1,2	F_8
F_9	1	F_9

Each string is called an *address*. Given addresses a_i and a_j, $a_i \preceq a_j$ denotes that a_i is a prefix of a_j, including the case where $a_i = a_j$. If $a_i \preceq a_j$ and the length of a_i is one less than the length of a_j, a_i *immediately precedes* a_j.

Proofs may contain certain status phrases as well as formulas. For example, to assert that a set of cases exhausts an earlier formula, the word **Exhaustive** is placed in the formula sequence. Additional status phrases are **Case Closed** for asserting instances of *Close* and **Assumptions Satisfied** for asserting instances of *CTA*.

Formally, a *proof sequence* is a sequence of the form $a_1F_1, a_2F_2, \ldots, a_n, F_n$, where each a_i is an address and each F_i an HHL formula or one of the status phrases. Proofs consist of a set of formulas called the *initial assumptions* and a proof sequence. The *assumptions* of a proof are those formulas either in the set of initial assumptions, or whose address contains at least two digits and does not appear as an address earlier in the sequence; the two digits restriction is required since top level assumptions are captured in the separate set. An assumption F_i is *visible* to formula F_j iff $a_i \preceq a_j$ or if F_i is in the set of initial assumptions. In Example 37, the starred formulas are assumptions and F_1 and F_2 are initial assumptions.

Definition 97 indicates which proof sequences correspond to valid proofs. Each case of the definition describes a step by which a formula or status phrase may be added to the proof; a name for the step is given in parenthesis. These names form the justifications that a user would give when developing proofs with the respective steps. Case 2 uses a shorthand (**Rule** j) to refer to any of the inference rules previously defined; in practice, the names of the rules would be cited as the actual justifications, as demonstrated in the examples in Chapter 5. The conditions within each case verify that formulas are only derived from formulas in their nesting scope and that assumptions are discharged properly for forms of implication introduction. For example, the last condition for Case 4 requires that only the antecedent assumption be discharged through restrictions on the addresses.

Definition 97 A *proof* in HHL is a pair $\langle \Gamma, S \rangle$, where Γ contains the initial assumptions and S is a proof sequence, such that for each i, $1 \leq i \leq n$, one of the following cases holds:

1. (**Axiom**) F_i is an axiom of CTL.
2. (**Rule** j) F_i is obtained from some $F_{j_1}, \ldots, F_{j_k}$ by Rule j such that for all j_h, $j_h < i$ and $a_{j_h} \preceq a_i$.
3. (**Proof Result**) There exists a proof $\langle \Gamma', S' \rangle$ such that F_i is the last formula in S' and all elements of Γ' are assumptions visible to F_i.
4. ($\rightarrow$ **Intro**) $F_i = F_j \rightarrow F_k$ for some $j, k < i$ such that
   - F_i, F_j, F_k are all SOL formulas or all CTL formulas,
   - F_j is an assumption visible to F_k,
   - $a_j = a_k$, and
   - a_i immediately precedes a_j.
5. (**Timing Diagram** $\rightarrow$ **Intro**) F_i is a $\mathrm{TD}^{\mathrm{TI}}$ timing diagram and there exist $\mathrm{TD}^{\mathrm{TI}}$ timing diagrams F_j and F_k such that
   - $j < i$ and $k < i$,
   - F_j is the assume portion of F_i,
   - F_i is isomorphic to F_k,
   - F_j is an assumption visible to F_k,

- $a_j = a_k$, and
- a_i immediately precedes a_j.

6. (**Close**) $F_i =$ **Case Closed** and there exist $F_{j_1} \ldots F_{j_k}$ such that *close* (Rule 41) is valid for $F_{j_1} \ldots F_{j_k}$ and for each j_h, $j_h < i$ and $a_{j_h} \preceq a_i$.

7. (**Contradiction**) $F_i = \neg F_j$ for some $j < i$ such that

   - F_i is either an SOL formula or a CTL formula,
   - F_j is an assumption visible to F_i,
   - a_i immediately precedes a_j, and
   - there exists $F_k =$ **Case Closed** such that $a_k = a_j$.

8. (**Cases Exhaustive**) $F_i =$ **Exhaustive** and there exist F_j and $F_{k_1}, \ldots, F_{k_h}$ such that

   - F_{k_1} through F_{k_h} are all assumptions,
   - either F_{k_1} through F_{k_h} exhaust F_j (Rule 38) or F_j follows by *bounds merge* from F_{k_1} through F_{k_h} (Rule 22),
   - $a_i = a_j$, and
   - a_i immediately precedes each of a_{k_1} through a_{k_h}.

9. (**Merge**) F_i an is HHL formula and there exists $F_j =$ **Exhaustive** such that $j < i$, $a_i = a_j$, and F_i is derived from the last formula in each exhaustive case by *exhaustive case merge* (Rule 39) or *bounds merge* (Rule 22).

10. (**CTA**) F_i =**Assumptions Satisfied** and *CTA* is valid for all assumptions visible to F_i (Rule 40).

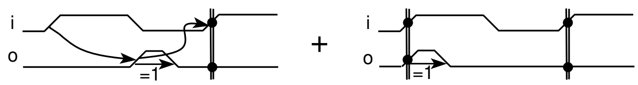
5

Examples

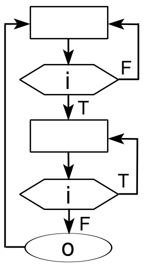
This chapter presents two verification examples in HHL. The first, the single pulser, demonstrates how HHL can be used to derive timing constraints from circuit diagrams. The second, an elevator controller, demonstrates the use of HHL for reasoning about parametric timing constraints.

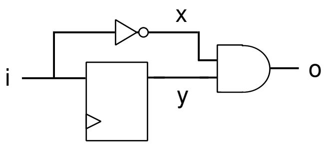
5.1 The Single Pulser

This proof adds the missing justifications to the proof from Chapter 1, Example 3. Two requirements comprise its behavioral specification: it produces exactly one output pulse for each input pulse and all output pulses are of unit duration. Assuming the additional restriction that the output pulse must occur before the next input pulse begins, the specification can be written as a CTR expression in which each timing diagram has the first rising transition object on i as its assume portion and is interpreted invariantly under strict satisfaction.



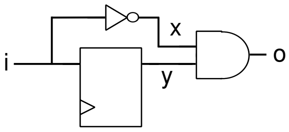
A single pulser implementation can be defined using either an ASM chart or a circuit diagram.



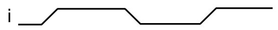


The behavioral equivalence of these two models is easily established using existing verification tools. The following proof that this implementation satisfies the timing diagram behavioral specification uses the circuit diagram implementation.

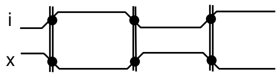
Proof 1



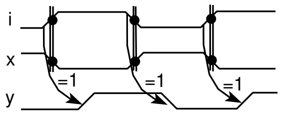
Given



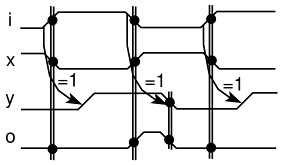
Given



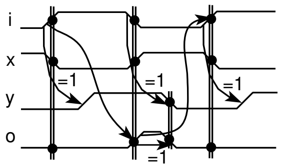
Inverter Rules [Rule 33]



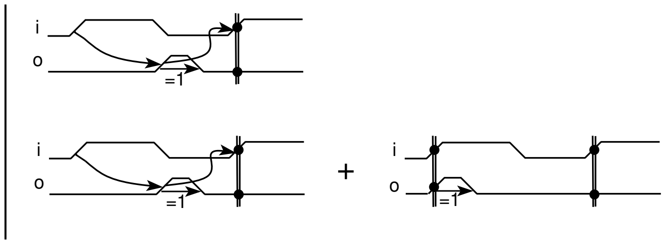
Delay Rules [Rule 35]



And Rules [Rule 34]



Bounds Slide [Rule 17]



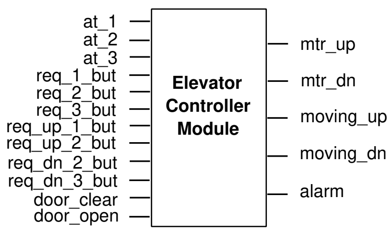
Simplification [Rule 42]

CTR Or Intro [Rule 37]

5.2 The Elevator Controller

Timing constraints are generally handled in verification through timed automata [2]. Most timed automata models, however, accept only concrete constraints, rather than symbolic or parametric constraints. An extension of timed automata to handling certain parametric constraints appears in [3]; tool support for these techniques is still forthcoming.

The goal is to design a controller for a three-floor elevator. The elevator should be equipped with an alarm that goes off if the door is kept open past a certain amount of time. The elevator takes requests for service as inputs and produces five output pulses: *mtr_up* and *mtr_dn*, for driving the motors that move the elevator, *moving_up* and *moving_dn* to be used to indicate the direction of the elevator to passengers waiting on floors, and *alarm*. A user may request to go up or down at any floor (inputs req_up_i_but and req_dn_i_but, for each floor i), or may request to go to any floor from inside the elevator (inputs req_i_but, for each floor i). In addition, the environment sends signals to the controller indicating when the elevator is at each floor (inputs at_i), when the door is open (input door_open), and when the door area is clear (door_clear, to indicate when the doors can safely be closed). The interface appears as follows:



The main components of the design are the scheduler, which determines which floor to go to next, and the floor managers (one for each floor), which control when to open and close the doors. The design uses two timers, one to indicate how long to leave the doors open at a floor and another to indicate how much time should be given for the door to clear before sounding the alarm. Additional minor components are needed to generate the pulses and store the current floor. The design is expressed as a collection of HHL formulas: the scheduler ASM chart appears in Figure 5.1. the floor manager ASM chart appears in Figure 5.2, the current floor ASM chart appears in Figure 5.3, the timer modules appear in Figure 5.6 and other interface components appear in Figures 5.4, 5.5, and 5.7.

In general, the inputs from the environment may occur at any time. The exceptions are straightforward and include conditions such as: until the doors have opened for the first time, it should not be possible for a service request to be generated inside the elevator. The environmental assumptions appear in Figure 5.8.

Several properties should be verified of the controller, intuitively described as follows:

1. The doors should only open at a floor if the elevator is at that floor.
2. The doors should only open at a floor if there was a request for service at or to that floor.
3. The doors should close after a reasonable amount of time.
4. Every request for service should eventually be handled.
5. Once the elevator starts moving in a particular direction, it should continue to move in that direction until all requests in that direction have been handled.
6. The elevator doesn't move unless the doors are closed.

These properties can be formally stated in HHL as follows. Each timing diagram is interpreted invariantly under non-strict satisfaction assuming the head object is the assume portion.

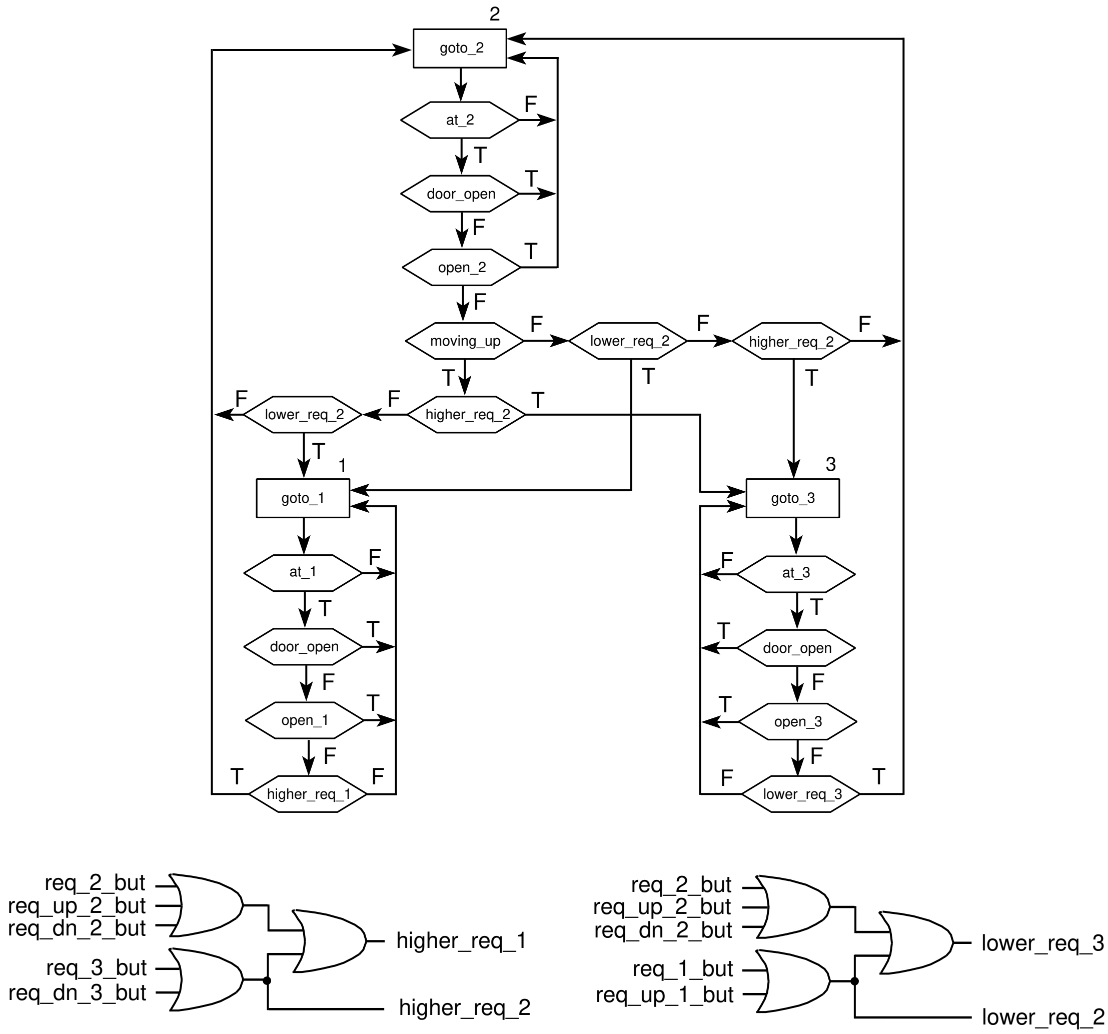


Figure 5.1: ASM chart for scheduler module with circuitry for associated signals.

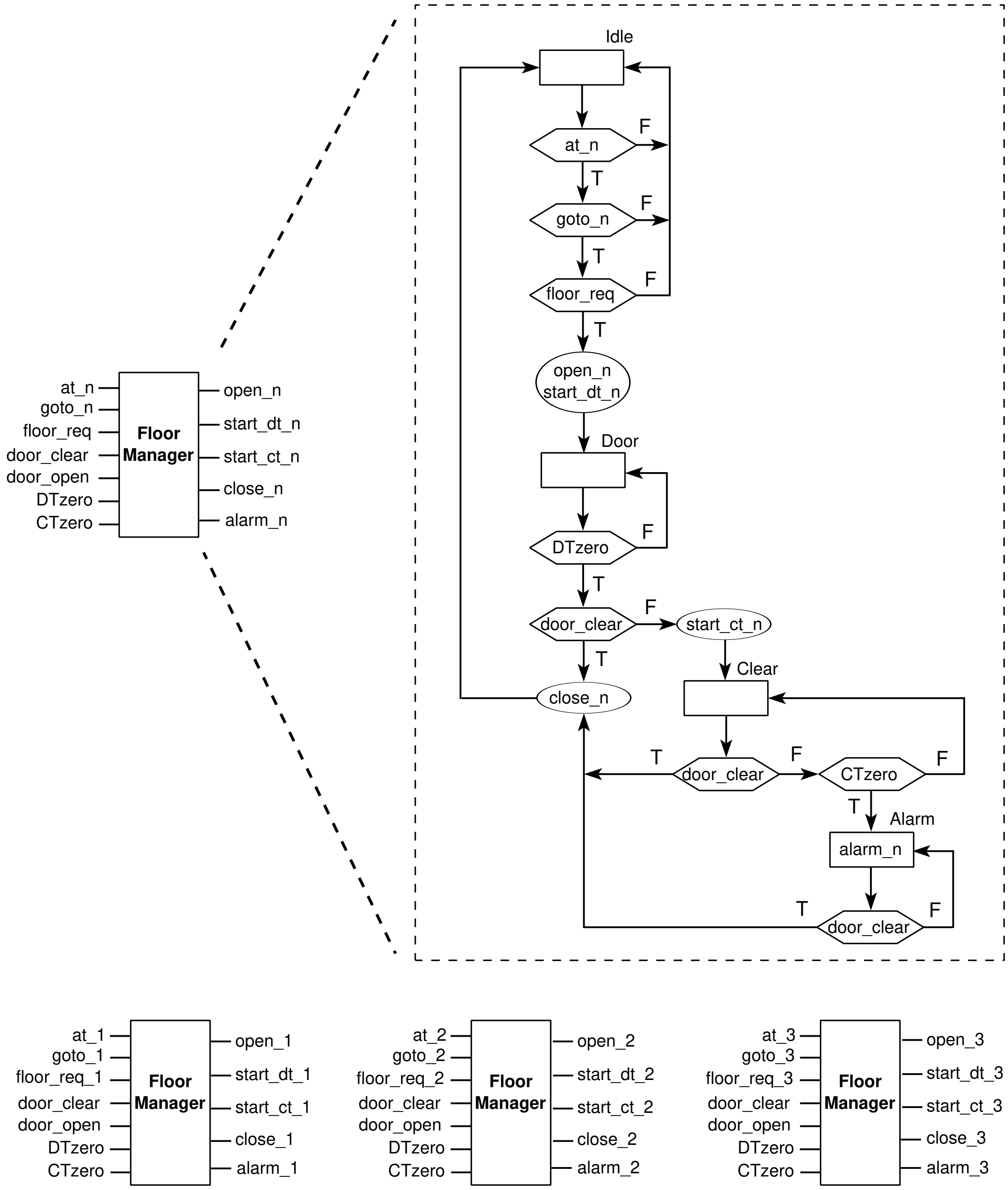


Figure 5.2: ASM chart for floor manager.

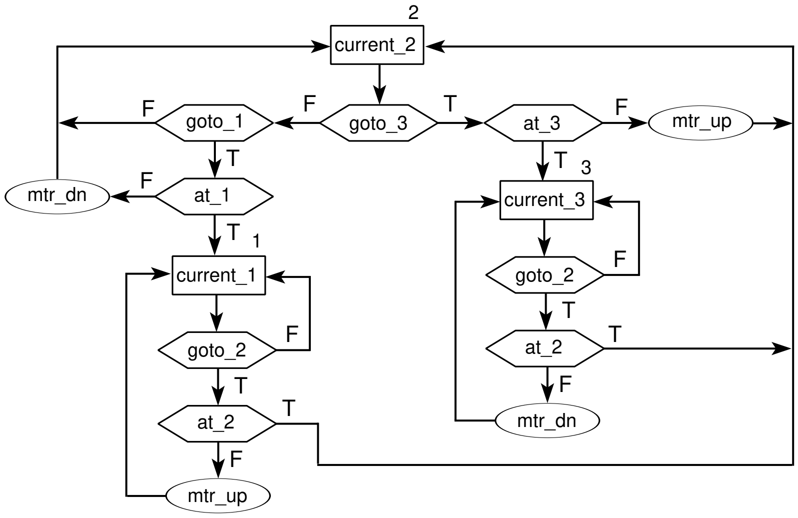


Figure 5.3: Current floor ASM chart. This chart also produces the outputs *mtr_up* and *mtr_dn*.

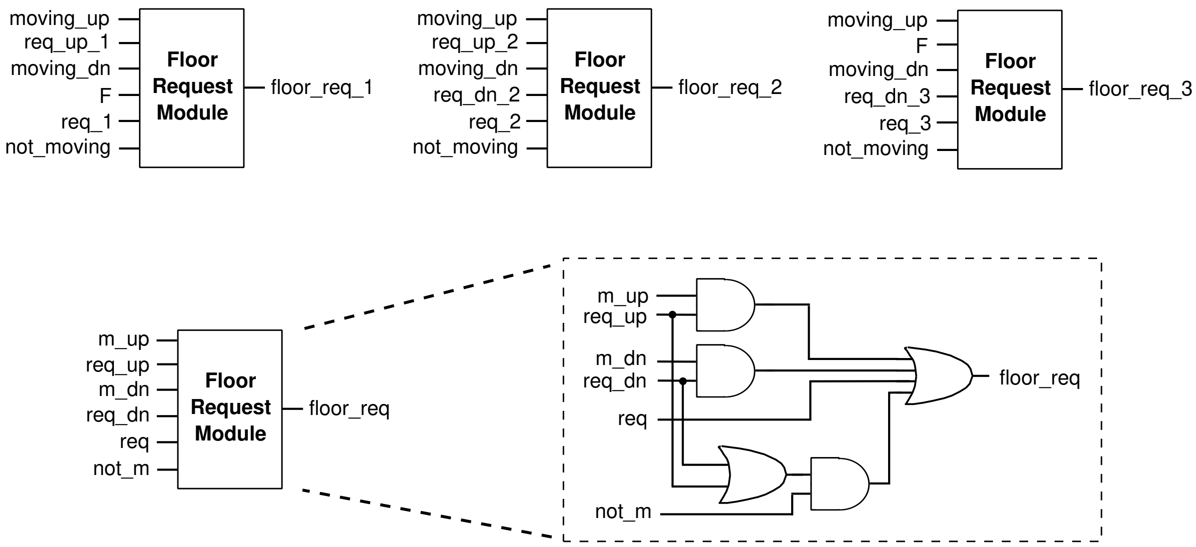


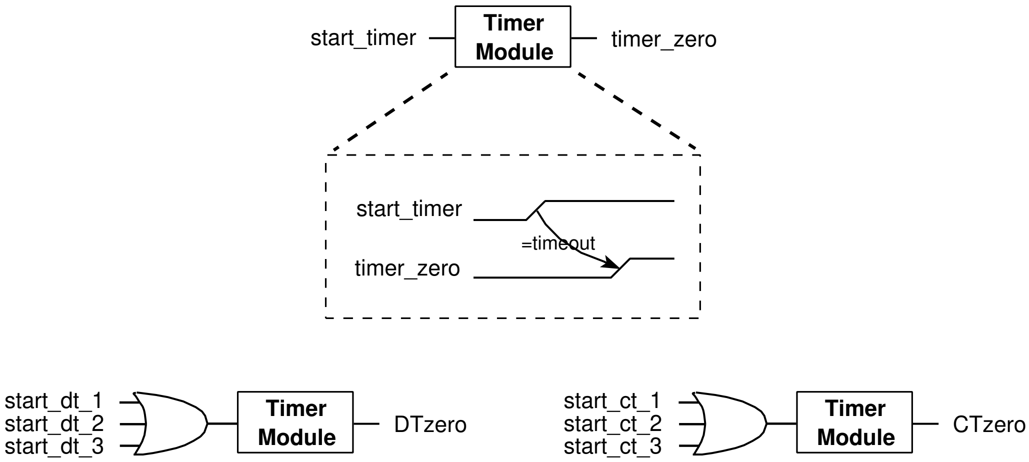
Figure 5.4: Modular design of the floor request components.

$$
\begin{aligned}
\text{moving\_up}(t) &= \text{mtr\_up}(t) \vee (\text{moving\_up}(t-1) \wedge \text{any\_req}(t) \wedge \neg\text{mtr\_dn}(t)) \\
\text{moving\_dn}(t) &= \text{mtr\_dn}(t) \vee (\text{moving\_dn}(t-1) \wedge \text{any\_req}(t) \wedge \neg\text{mtr\_up}(t)) \\
\text{not\_moving}(t) &= \neg\text{mtr\_up}(t) \wedge \neg\text{mtr\_dn}(t) \wedge (\neg\text{any\_req}(t) \vee \text{not\_moving}(t-1))
\end{aligned}
$$

$$\text{not\_moving}(0) = \mathsf{T} \qquad \text{moving\_up}(0) = \mathsf{F} \qquad \text{moving\_dn}(0) = \mathsf{F}$$

$$\text{alarm}(t) = \text{alarm\_1}(t) \vee \text{alarm\_2}(t) \vee \text{alarm\_3}(t)$$

Figure 5.5: Definition of moving and alarm signals.



$$\forall t(\neg\text{start\_timer}(t) \wedge \text{start\_timer}(t+1)) \rightarrow \neg\text{timer\_zero}(t+1)$$

Figure 5.6: Timer module. The timing diagram is interpreted iteratively with non-strict satisfaction and the rising transition object on start_timer as the assume portion.

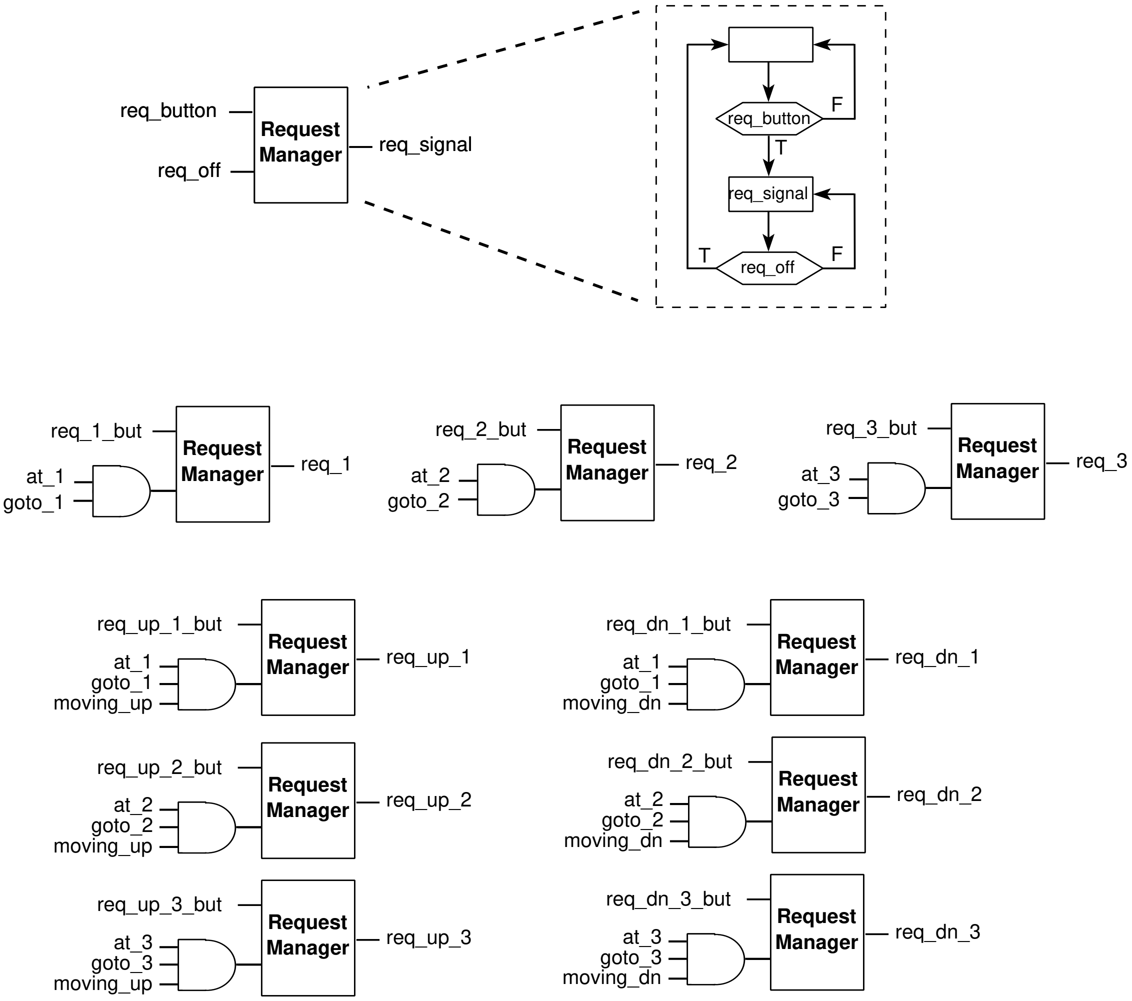


Figure 5.7: Request Manager module.

$$\mathsf{AG}\ ((\mathrm{open\_1} \vee \mathrm{open\_2} \vee \mathrm{open\_3}) \rightarrow \mathsf{AF}\ \mathrm{door\_open})$$
$$\mathsf{AG}\ ((\mathrm{close\_1} \vee \mathrm{close\_2} \vee \mathrm{close\_3}) \rightarrow \mathsf{AF}\ \neg\ \mathrm{door\_open})$$
$$\neg\ (\mathrm{req\_1\_but} \vee \mathrm{req\_2\_but} \vee \mathrm{req\_3\_but})\ \mathsf{U}\ \mathrm{door\_open}$$
$$\mathsf{AG}\ (\mathrm{at\_1} \rightarrow \mathsf{A}[\mathrm{at\_1}\ \mathsf{U}\ \neg\ \mathrm{goto\_1}])$$
$$\mathsf{AG}\ (\mathrm{at\_2} \rightarrow \mathsf{A}[\mathrm{at\_2}\ \mathsf{U}\ \neg\ \mathrm{goto\_2}])$$
$$\mathsf{AG}\ (\mathrm{at\_3} \rightarrow \mathsf{A}[\mathrm{at\_3}\ \mathsf{U}\ \neg\ \mathrm{goto\_3}])$$

Figure 5.8: Environmental assumptions on the elevator controller.

1. $\mathsf{AG}\ ((\mathrm{open\_1} \rightarrow \mathrm{at\_1}) \wedge (\mathrm{open\_2} \rightarrow \mathrm{at\_2}) \wedge (\mathrm{open\_3} \rightarrow \mathrm{at\_3}))$

2. $\mathsf{AG}\ (\mathrm{open\_1} \rightarrow \mathrm{req\_1\_but} \vee \mathrm{req\_up\_1\_but} \vee \mathrm{req\_dn\_1\_but})$

   $\mathsf{AG}\ (\mathrm{open\_2} \rightarrow \mathrm{req\_2\_but} \vee \mathrm{req\_up\_2\_but} \vee \mathrm{req\_dn\_2\_but})$

   $\mathsf{AG}\ (\mathrm{open\_3} \rightarrow \mathrm{req\_3\_but} \vee \mathrm{req\_up\_3\_but} \vee \mathrm{req\_dn\_3\_but})$

   $\mathsf{AG}\ (\mathrm{req\_1\_but} \rightarrow (\mathrm{req\_1\_but}\ \mathsf{U}\ \mathrm{open\_1}))$

   $\mathsf{AG}\ (\mathrm{req\_2\_but} \rightarrow (\mathrm{req\_2\_but}\ \mathsf{U}\ \mathrm{open\_2}))$

   $\mathsf{AG}\ (\mathrm{req\_3\_but} \rightarrow (\mathrm{req\_3\_but}\ \mathsf{U}\ \mathrm{open\_3}))$

   $\mathsf{AG}\ (\mathrm{req\_up\_1\_but} \rightarrow (\mathrm{req\_up\_1\_but}\ \mathsf{U}\ \mathrm{open\_1}))$

   $\mathsf{AG}\ (\mathrm{req\_up\_2\_but} \rightarrow (\mathrm{req\_up\_2\_but}\ \mathsf{U}\ \mathrm{open\_2}))$

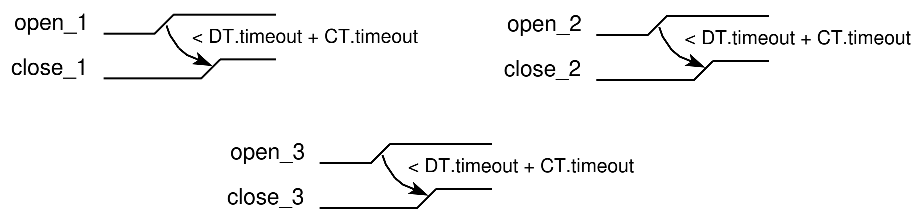
   $\mathsf{AG}\ (\mathrm{req\_up\_3\_but} \rightarrow (\mathrm{req\_up\_3\_but}\ \mathsf{U}\ \mathrm{open\_3}))$

   $\mathsf{AG}\ (\mathrm{req\_dn\_1\_but} \rightarrow (\mathrm{req\_dn\_1\_but}\ \mathsf{U}\ \mathrm{open\_1}))$

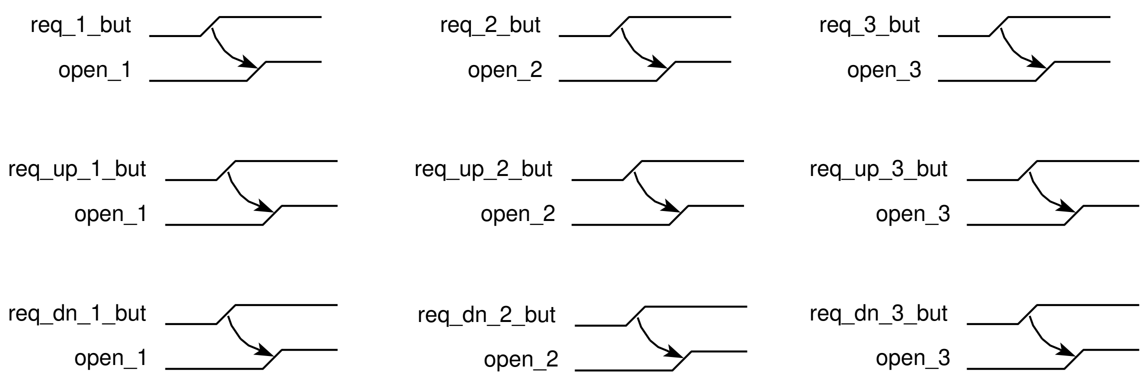
   $\mathsf{AG}\ (\mathrm{req\_dn\_2\_but} \rightarrow (\mathrm{req\_dn\_2\_but}\ \mathsf{U}\ \mathrm{open\_2}))$

   $\mathsf{AG}\ (\mathrm{req\_dn\_3\_but} \rightarrow (\mathrm{req\_dn\_3\_but}\ \mathsf{U}\ \mathrm{open\_3}))$

3.



4.



5. $\mathsf{AG}\ (\text{moving\_up} \rightarrow \text{moving\_up}\ \mathsf{U}\ [(\text{at\_1} \wedge \neg(\text{req\_2\_but} \vee \text{req\_up\_2\_but} \vee \text{req\_3\_but})) \vee (\text{at\_2} \wedge \neg\ \text{req\_3\_but}) \vee \text{at\_3}])$

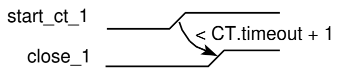
   $\mathsf{AG}\ (\text{moving\_dn} \rightarrow \text{moving\_dn}\ \mathsf{U}\ [(\text{at\_3} \wedge \neg(\text{req\_2\_but} \vee \text{req\_dn\_2\_but} \vee \text{req\_1\_but})) \vee (\text{at\_2} \wedge \neg\ \text{req\_1\_but}) \vee \text{at\_1}])$

6. $\mathsf{AG}\ (\text{door\_open} \rightarrow (\neg\ \text{mtr\_up}\ \mathsf{U}\ \neg\ \text{door\_open}))$

   $\mathsf{AG}\ (\text{door\_open} \rightarrow (\neg\ \text{mtr\_dn}\ \mathsf{U}\ \neg\ \text{door\_open}))$

Most of these properties correspond to regular languages and can hence be verified using existing automated tools. The conditions that the door closes in a reasonable amount of time (Property 3) do not. They must be verified using an interactive proof. Given the similarity of the timing diagrams listed in Property 3, the proofs are explicitly provided only for the first floor. The proofs are given at a fairly low level of detail for purpose of demonstration. In practice, it is assumed that there would be derived rules corresponding to more of the steps used in the proofs. The proofs appear in a natural deduction format. As the ASM charts are so large, those charts that would appear as steps in the proofs have been isolated into Figures 5.9 and 5.10. Lines in the proofs that would normally contain one of these ASM charts instead contain a small gray icon and a reference to one of these figures.

The first proof is an intermediate step in the proof of property 3. It establishes that if the clear timer is set off, the close door signal is asserted before that timer expires under the assumption that the alarm doesn't go off.



Intuitively, the proof is clear from the floor manager ASM chart (Figure 5.2). The floor manager must go to state *Clear* after asserting *start_ct*. If *door_clear* is asserted before *CTZero*, the *close* signal is asserted. Otherwise, the alarm goes off, which contradicts an assumption in the proof. The proof is based on the assumption that the floor manager starts in the *Idle* state.

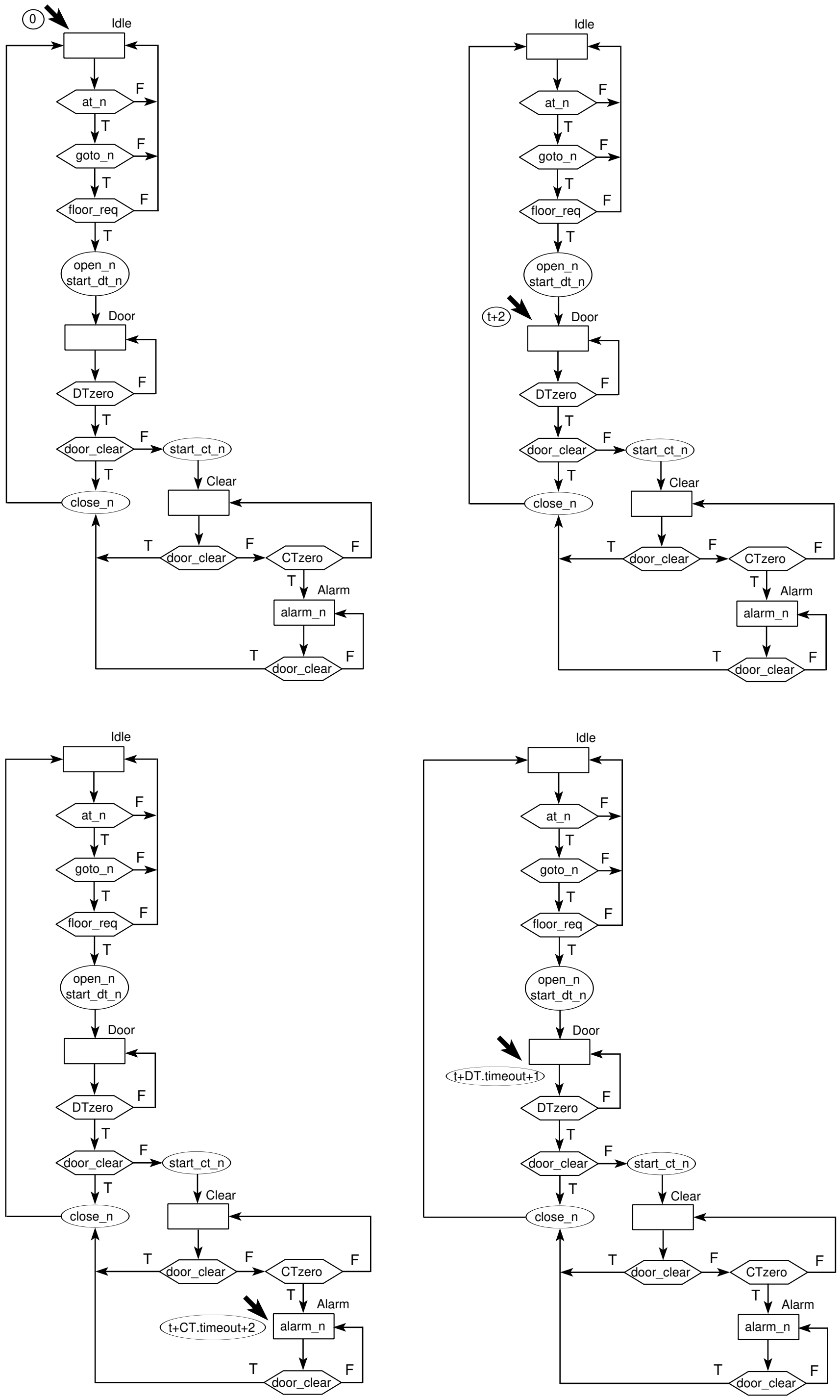


Figure 5.9: ASM charts needed in sample proof.

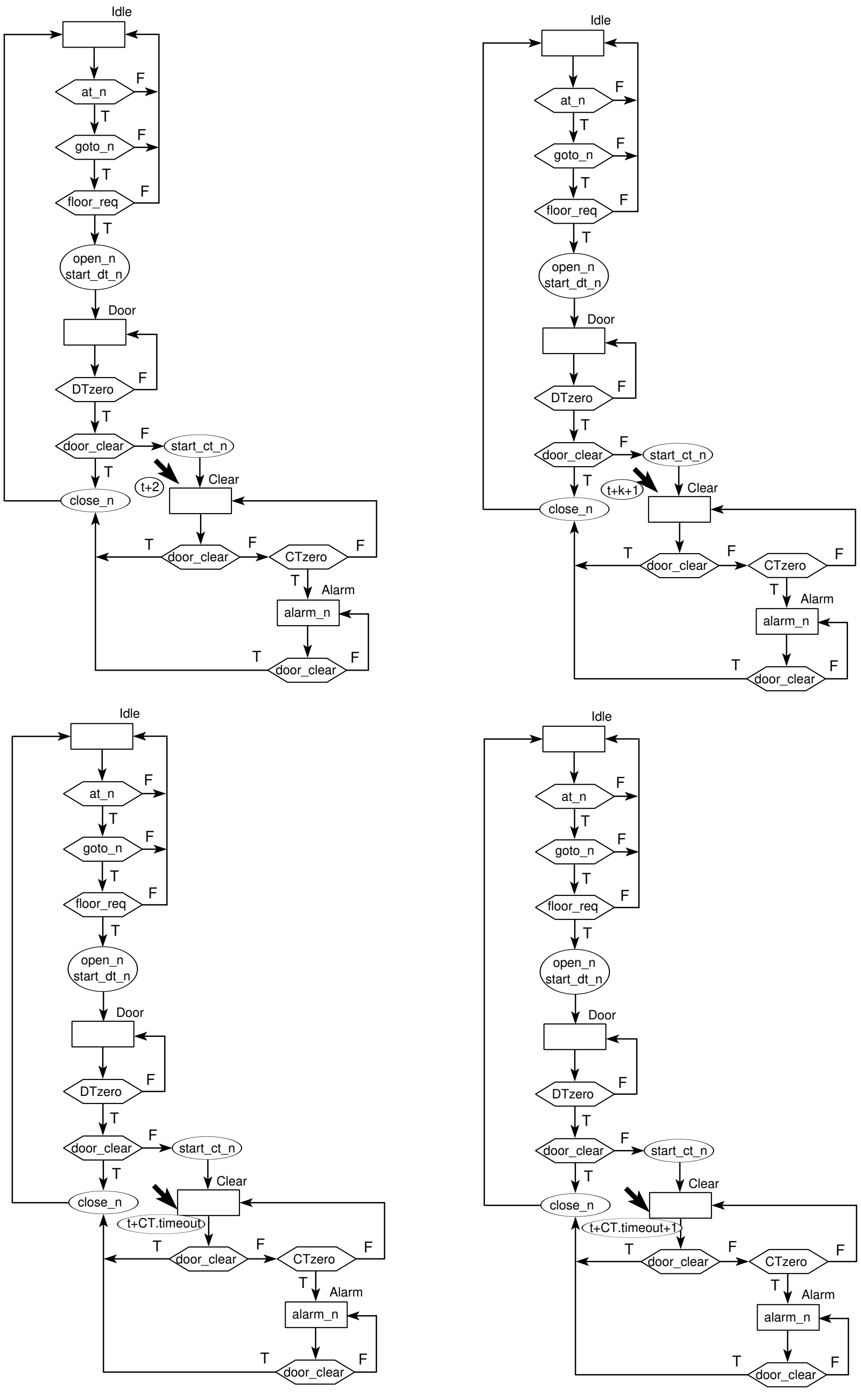


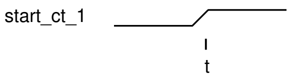
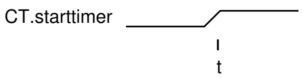
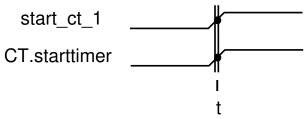
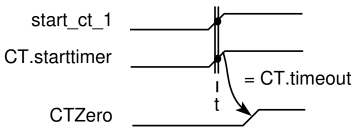
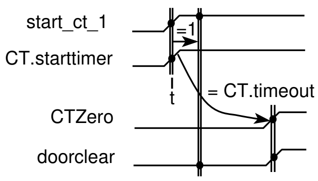
Figure 5.10: ASM charts needed in sample proof.

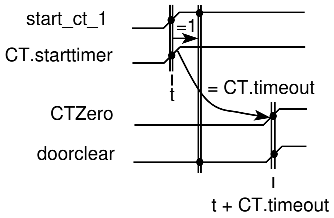
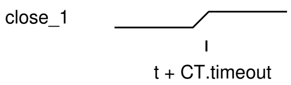
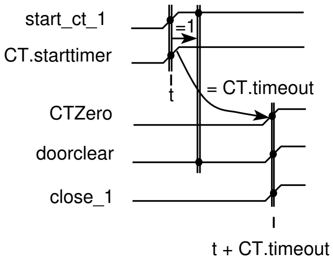
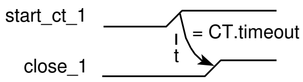
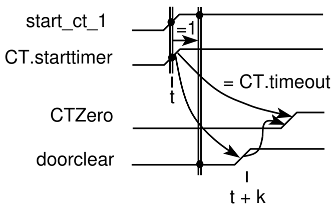
The proof also relies on the property

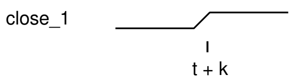
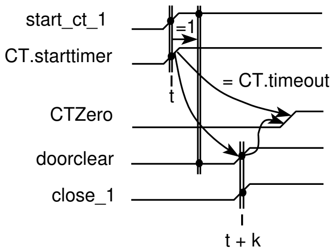
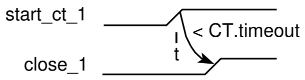
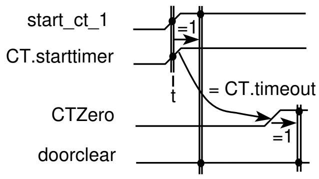
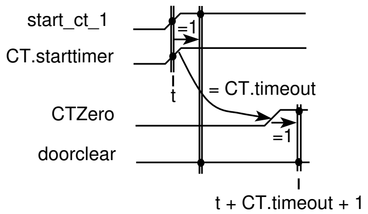
$$\forall t(\text{start\_ct\_1}(t+1) \rightarrow (\neg\text{start\_ct\_2}(t) \wedge \neg\text{start\_ct\_3}(t)))$$

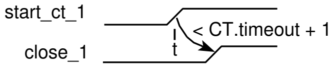
This property can either be proven in HHL or can be deferred to an existing language containment tool since it is regular. The latter has been done in this case; the proof appears in Appendix B.

Proof 2

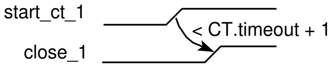
Grounded ASM (Figure 5.9, top left)	Given
AG$\neg$alarm	Given
	Assume
$\neg$start_ct_1$(t) \wedge$ start_ct_1$(t+1)$	Observe [Rule 29]
$\neg$CT.starttimer(t)	COSPAN Proof
CT.starttimer$(t+1)$	Or Rules [Rule 36]
	Apply [Rule 30]
	Anchor Align [Rule 16]
	Defn. timer [Figure 5.6]
$\neg$CTZero$(t+1)$	Defn. timer [Figure 5.6]
ASM Chart(Figure 5.10, top left)	Output Transition [Rule 28]
	Assume

	Time Label Bounds [Rule 15]
ASM Chart(Figure 5.10, bottom left)	Looping [Rule 27]
$\neg$doorclear$(t + \text{CT.timeout})$	Observe [Rule 29]
$\neg$CTZero$(t + \text{CT.timeout})$	Observe [Rule 29]
ASM Chart(Figure 5.10, bottom right)	Transition Enabled [Rule 26]
doorclear$(t + \text{CT.timeout} + 1)$	Observe [Rule 29]
Close_1$(t + \text{CT.timeout} + 1)$	Output Enabled [Rule 24]
$\neg$Close_1$(t + \text{CT.timeout})$	Observe [Rule 29]
	Apply [Rule 30]
	Anchor Align [Rule 16]
	Simplification [Rule 42]
	Assume
ASM Chart(Figure 5.10, top right)	Looping [Rule 27]
doorclear$(t + k + 1)$	Observe [Rule 29]

Close_1$(t+k+1)$	Output Enabled [Rule 24]
$\neg$Close_1$(t+k)$	Observe [Rule 29]
	Apply [Rule 30]
	Anchor Align [Rule 16]
	Simplification [Rule 42]
	Assume
	Time Label Bounds [Rule 15]
ASM Chart(Figure 5.10, bottom right)	Looping [Rule 27]
$\neg$doorclear$(t + \text{CT.timeout} + 1)$	Observe [Rule 29]
CTZero$(t + \text{CT.timeout} + 1)$	Observe [Rule 29]
ASM Chart(Figure 5.9, bottom left)	Transition Enabled [Rule 26]
alarm_1$(t + \text{CT.timeout} + 1)$	Output Enabled [Rule 24]
alarm$(t + \text{CT.timeout} + 1)$	Or Rules [Rule 36]
Case Closed	Close [Rule 41]
Exhaustive	Exhaust [Rule 38]

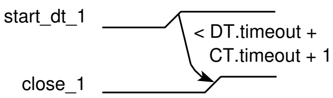


Merge [Rule 39]



Time Generalization [Rule 19]

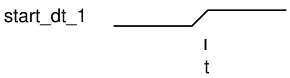
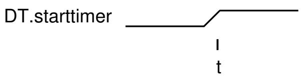
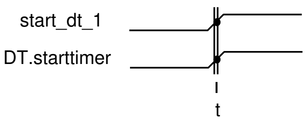
The second proof uses the first to establish the desired property

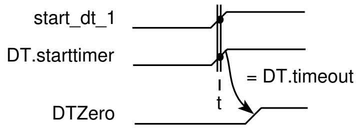


It uses similar assumptions to the first proof: that the alarm never goes on, the floor manager starts in the *Idle* state, and the following property has previously been proven

$$\forall t(\text{start\_dt\_1}(t+1) \rightarrow (\neg\text{start\_dt\_2}(t) \wedge \neg\text{start\_dt\_3}(t)))$$

Proof 3

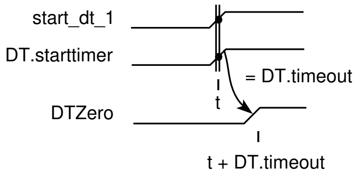
Grounded ASM (Figure 5.9, top left)	Given
AG$\neg$alarm	Given
	Assume
$\neg\text{start\_dt\_1}(t) \wedge \text{start\_dt\_1}(t+1)$	Observe [Rule 29]
$\neg\text{DT.starttimer}(t)$	COSPAN Proof
$\text{DT.starttimer}(t+1)$	Or Rules [Rule 36]
	Apply [Rule 30]
	Anchor Align [Rule 16]



Defn. timer [Figure 5.6]

$\neg$DTZero$(t+1)$ — Defn. timer [Figure 5.6]

ASM Chart (Figure 5.9, top right) — Output Transition [Rule 28]



Time Label Bounds [Rule 15]

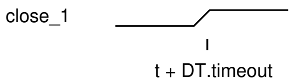
DTZero$(t+\text{DT.timeout}+1)$ — Observe [Rule 29]

ASM Chart (Figure 5.9, bottom right) — Looping [Rule 27]

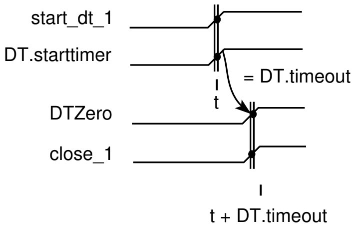
doorclear$(t+\text{DT.timeout}+1)$ — Assume

$\neg$close_1$(t+\text{DT.timeout})$ — Output Unenabled [Rule 24]

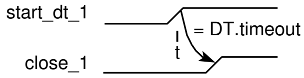
close_1$(t+\text{DT.timeout}+1)$ — Output Enabled [Rule 24]



Apply [Rule 30]



Anchor Align [Rule 16]

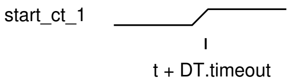


Simplification [Rule 42]

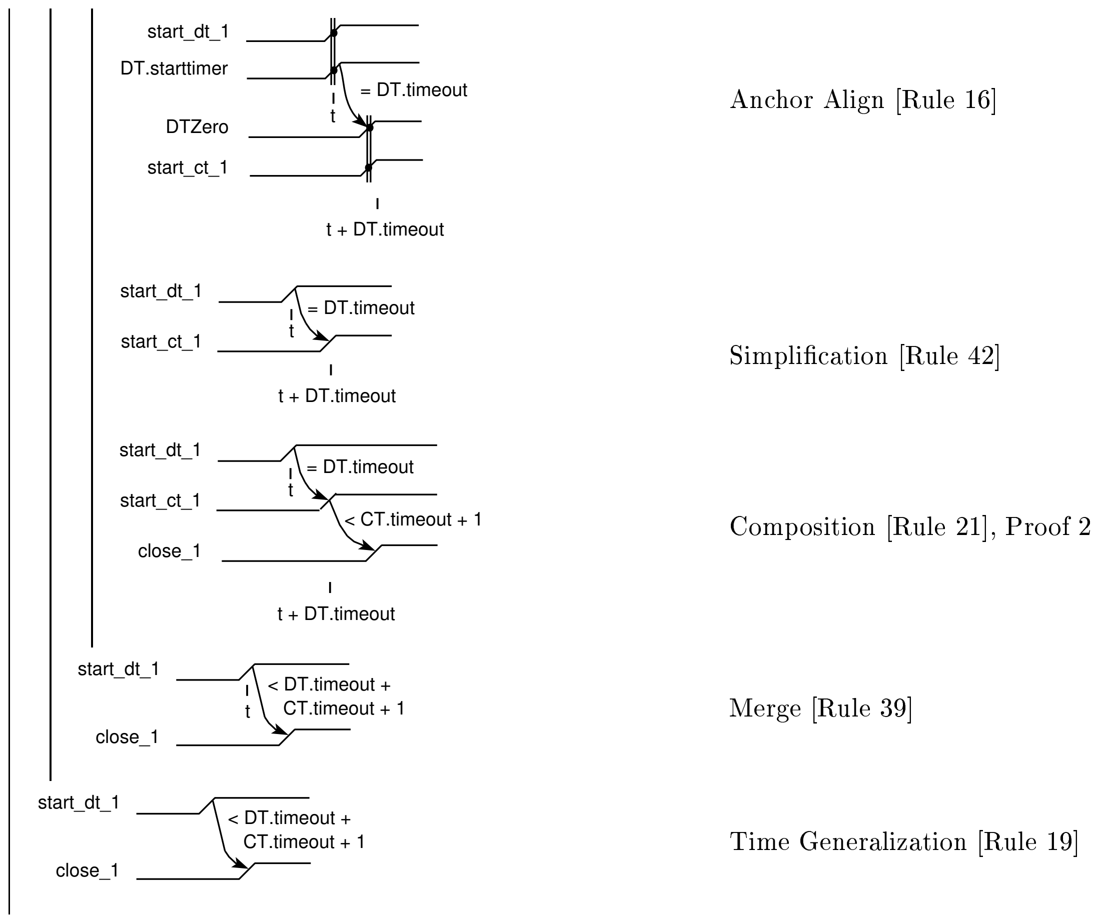
$\neg$doorclear$(t+\text{DT.timeout}+1)$ — Assume

$\neg$start_ct_1$(t+\text{DT.timeout})$ — Output Unenabled [Rule 24]

start_ct_1$(t+\text{DT.timeout}+1)$ — Output Enabled [Rule 24]



Apply [Rule 30]



The remaining two cases of Property 3 have similar proofs. The proofs of the remaining properties have been done in COSPAN [82]; the COSPAN statements of the design, the properties, and a description of the COSPAN system appear in Appendix B. Admittedly, working between COSPAN and HHL without a provable syntax translation is not rigorous. However, as the remaining properties can be handled by existing automated tools, there is little purpose to demonstrating their deductive proofs in HHL.

The COSPAN verification actually detects a potential deadlock in the circuit while verifying the "requests eventually answered" properties (Item 4). The deadlock arises if a person inside the elevator repeatedly requests the current floor. Assume a person gets on the elevator at the second floor and the door is clear when the DT timer expires. The second floor manager asserts the *close* signal and enters the Idle state at time t. Assume the person requests the second floor from inside the elevator at time t. The elevator, having just seen the *close* signal, may still be asserting the *door_open* signal (since it does take some amount of time for the door to physically close). While the elevator has been servicing the second floor, the scheduler has been in the state labeled 2, the *goto_2* state (Figure 5.1). Looking at the scheduler ASM, the scheduler once again asserts *goto_2* at time t since the door is still open. The floor manager (Figure 5.2), in state *Idle*, finds *at_2*, *goto_2*, and *floor_req* all true, so it opens the doors at the second floor at time t. This process can go on indefinitely, in which case the elevator would deadlock.

The deadlock could be eliminated by changing the environment code so that *door_open* becomes false as soon as a *close* signal is asserted. This is a poor solution. In reality, elevator doors do not close instantaneously. The design would verify under the altered environment model, but the controller chip itself would then fail in practice. Environment models should reflect how signals behave in practice as accurately as possible.

Another solution is to insert some delay in the floor manager between the time the *close* signal is asserted and the return to the *Idle* state. Figure 5.11 shows the floor manager with this alteration, where the transition to *Idle* occurs after the door has physically closed. Using the new floor manager, all of the desired properties now verify. Notice that this change does not affect the proofs given earlier as those proofs rely on an unchanged portion of the floor manager ASM chart.

The deadlock error demonstrates the power of automated verification. Attempting to prove the request answered properties deductively in HHL would have eventually located the problem, but the proof is tedious and complicated. Once the deadlock is discovered, the diagrams allow a user to visualize the conditions causing the problem very nicely. Experience recommends the use of automated techniques to test deadlock conditions whenever possible.

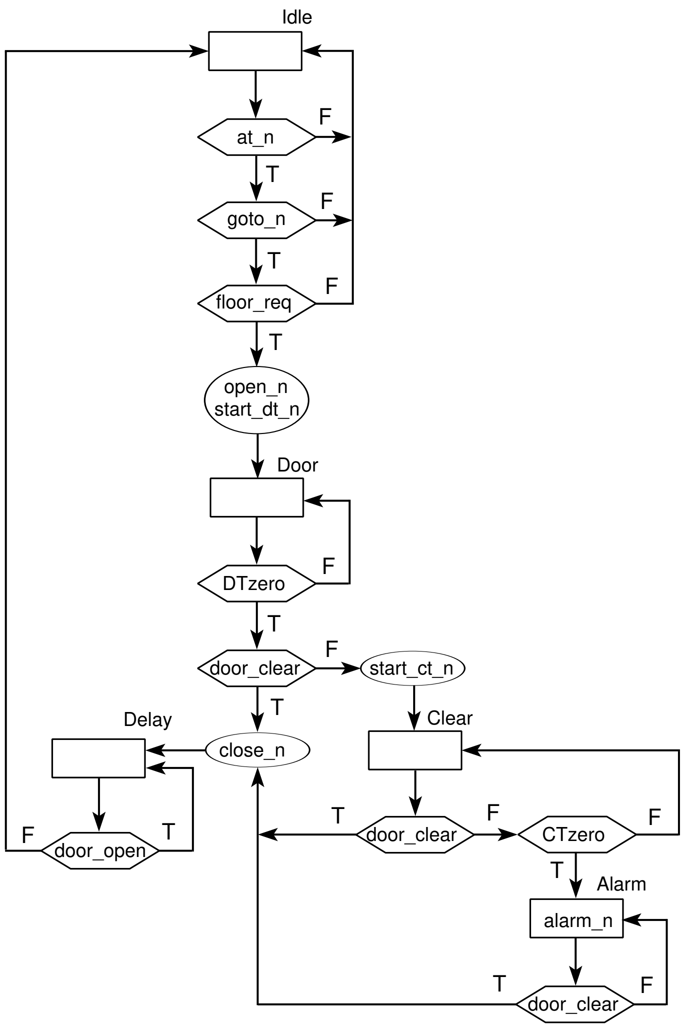


Figure 5.11: Corrected version of the floor manager ASM chart

6

Conclusions

> The principles, objects, and relationships offered by [finite state machines, typed set theory, algebra, logic] are absurdly different from the principles, objects, and relationships about which these engineers are concerned. [73]
>
> Pamela Zave

> It is particularly important that you understand our approach to the details of digital hardware. The overriding emphasis is to let the problem solution dictate the hardware, rather than allowing premature commitments to hardware to coerce the solution. [70]
>
> Franklin P. Prosser and David E. Winkel

Formal methods aims to provide mathematical foundations for system design. Such a foundation is most effective when it models actual practice. Unfortunately, most formal methods tools are rooted in mathematical logic, which is not the *lingua-franca* of hardware design. Hence, formal methods appears to coerce, rather than support, actual practice.

Diagrams are used extensively in hardware design practice; they convey architectural details, timing details, and control-flow information clearly and compactly. This dissertation develops HHL, a formal logic of diagrammatic and sentential design representations that provides a mathematical foundation for reasoning about and verifying designs using diagrams. Such a mathematical theory is essential if tools are to eventually support how these representations are used in practice; without a theory, tool support for diagrammatic representations remains grounded in conventional formal logic and thus fails to provide genuine support for practice.

By treating diagrams as formal, logical objects, HHL attempts to mimic how people reason with diagrammatic information. Designers do not mentally translate diagrams into sentential logic. If anything, the reverse occurs: people organize certain sententially presented information, such as the sentential logic specifications of the handshaking protocol in Example 2, by drawing diagrams. HHL eliminates the sentential logic middleman for such information.

HHL integrates design practice and verification methodology through its semantic basis in language theory. Language operations provide clean characterizations of verification techniques, such as abstractions and decompositions. Although these language-based techniques have been used primarily in algorithmic verification systems, they are equally applicable for deductive, interactive verification. Decompositions resemble the splitting of proofs into subcases through **Cases Exhaustive**. Abstraction is used extensively in both theorem-proving and model-checking, although in rather different ways. Language theory therefore provides a uniform framework for both algorithmic and interactive verification.

Within language-theoretic approaches to verification, HHL provides a clean integration of algorithmic and interactive verification through its timing diagram notations. Timing diagrams are often, if not always, syntactically classifiable with regards to Chomsky language classes. This characterization enables us to determine when algorithmic verification is applicable via existing tools from a theoretical standpoint. Other tools supporting both algorithmic and interactive verification, such as PVS, use a separate notation to state problems suitable for model-checking than for theorem-proving. HHL provides a cleaner integration of the two techniques from an expressibility standpoint due to its notational uniformity. It also prevents translation errors arising from attempting to manually move problems between differing theorem-proving and model-checking notations.

As a logical formalization, HHL's greatest contribution lies in its treatment of timing diagrams.

Timing diagrams are the least standardized of HHL's diagrammatic representations in practice, both syntactically and semantically. This work identifies a set of parameters that uniformly formalize several common uses of timing diagrams. It further investigates standard verification questions, such as language containment, on non-regular timing diagrams. The resulting decision procedures suggest that timing diagrams have structural features of particular utility to verification that should be investigated further.

Like any logical theory of design practice, HHL opens the window to discovering new paradigms for formal methods. Diagrams naturally represent structural and spatial information; diagrammatic logic can therefore be viewed as a form of structural logic. Structural information has already proven useful in formal methods through binary-decision diagrams, which dramatically increased the capabilities of automated verification [17, 61]; it has proven useful in mathematical theorem-proving through Barker-Plummer and Bailin's GROVER/& system. HHL provides a framework for further studies of structural information in formal methods.

Viewed as a structural logic, HHL forms the next step in an evolution of formal methods logics as demonstrated by the handshaking protocol discussed in Example 2, Chapter 1. In the mid-eighties, Gordon advocated higher-order logics for hardware reasoning due to their generality [35]. Temporal logics, previously advocated by Pnueli, have since become extremely popular, partially because their implicit time variables provide more compact statements of properties [68]. Diagrammatic logics compact representations further still through implicit structural information.

This evolution moves towards domain-specific formal methods tools and away from general-purpose reasoning systems. Such movement echos Rushby's apprehension as quoted in the introduction: "a formal specification method built around a particular diagrammatic or tabular notation may have rather restricted application, and limited mechanized support for general forms of analysis" [72]. This is true. Although heterogeneous logics could theoretically circumvent this problem, tools supporting diagrammatic representations would likely be used for applications suited to those diagrams. Specializing tools, however, supports practice because reasoning tends to be domain-specific: as stated by Lutz in the IEEE roundtable, "Engineers rarely (if ever) resort to first principles. Instead, they use tables, charts, and equations that capture the phenomena of interest $\cdots$ the mathematics, while abstract, is expressed in terms of the problem at hand" [73].

6.1 Future Work

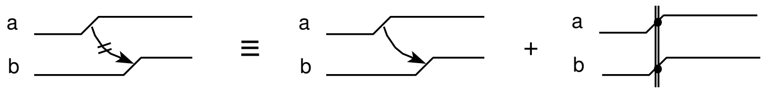
This work suggests several future projects, some directly related to HHL and some in the general areas of verification and diagrammatic reasoning. Implementing a tool based on HHL is of primary importance. HHL is currently a mathematical foundation; its practical feasibility must be investigated in terms of both designer usability and verification support. The envisioned tool will contain editors for each of the diagrammatic representations and an interactive proof development facility for entering both sentential formulas and proofs in general. Decision procedures supporting algorithmic verification will be provided in addition to the deductive system. Once implemented, the tool will provide a framework for investigating several questions:

- Which types of hardware reasoning proofs are easier to develop in HHL as opposed to existing sentential formalisms?
- Diagrammatic logics have yielded more compact proofs than sentential logics in other problem domains [9]. Is this true for HHL? For what types of problems is the difference most dramatic?
- Is diagrammatic scalability a problem for large designs? If so, how can it be controlled?

On a more abstract level, the impact of structural information on formal methods techniques should be examined in more detail. For example, what is the relationship between CTR representations and the automata that capture the same temporal relationships? In the worst case, translating temporal logic formulas to automata takes time exponential in the size of the formula [85]. Does the structure of the CTR formula affect the size of the resulting automata in certain cases? Can we identify certain transformation rules and general forms that minimize the automaton size blowup for certain expressions? The insights provided by such results would be useful for improving the complexity of automata-theoretic verification for the associated cases.

Developing efficient decision procedures for the regular containment problem would also be useful to the verification community. Such procedures would extend the applicability of algorithmic verification to general timing diagram languages, as opposed to purely regular languages. The existing results should also be extended, if possible, to Büchi automata, which are more widely used in verification practice than final-state finite automata.

HHL's timing diagrams provide notations for two temporal relationships: concurrency and succession. Some operators express combinations of these two relationships, such as CTL's AF operator. Section 3.1 demonstrated that HHL needs a disjunction of two timing diagrams to express AF. While new notations could be added to HHL for AF and other operators, a more general and flexible option is to develop diagrammatic macro systems. Such a project involves defining pattern matching systems over diagrammatic notations; the detail given by diagrammatic representations makes this challenging. For example, suppose the following expression intends to define the AF macro:



Does this expression define the macro only for rising transitions on a or for any events on a? Presumably, the latter interpretation is desirable, but this requires a pattern matcher to distinguish between abstract and concrete events. A related issue is determining a set of notations that form a suitable basis for a diagrammatic macro system. Barwise and Etchemendy have argued against the likelihood of finding a Hilbert's-program for diagrams, a set of diagrammatic notations that could capture all diagrammatic logics [8]. Accordingly, diagrammatic macro systems would likely be highly specialized to particular problem domains.

Finally, the work done here for hardware systems should be investigated in the context of software systems. As noted by Parnas, software formal methods suffers from similar notational difficulties to hardware formal methods: "We need a lot more work on notation. The notation that is purveyed by most formal methods researchers is cumbersome and hard to read" [73]. Software formal methods differs from hardware formal methods in several ways. Hardware systems are usually finite-state, which lends them naturally to algorithmic verification, while infinite-state software systems must be abstracted to finite-state systems before algorithmic verification can be applied. In the context of diagrams, hardware systems are defined at a lower level than software systems; small diagrammatic elements easily capture hardware design, while diagrams are less commonly used in software design. A heterogeneous logic for software design would therefore differ substantially from HHL, but would, I hope, accord many of the same benefits.

A

Scheme Code for CTL Translation Algorithm

```
;; Kathi Fisler
;; Implementation of timing diagram→CTL translation procedures

;; The following code refers to two undefined functions: GenObj and
;; SigPat, each of which takes a single natural number, representing a
;; time point, as an argument. General definitions of these functions
;; require a Scheme representation of timing diagrams, which does not
;; exist at this time. Dummy functions were defined for GenObj and
;; SigPat for purposes of testing this code.

;; Constraint? is true if no table has i+1 as first item in header

(define Constraint?
  (lambda (i tables)
    (ormap (lambda (table) (= (caar table) (add1 i))) tables)))

;; rem-tables locates the table whose first column is labeled i+1,
;; extracts all rows with count in that column, strips off the first
;; column from those rows, and creates a new table from them. All
;; tables whose first column is not labeled i+1 are left untouched.
;; This implements the table modification procedure.

(define rem-tables
  (lambda (tables i count)
    (cond ((null? tables) '())
          ((= (caaar tables) (add1 i))
           (let ((T (car tables)))
             (letrec ((rel-rows
                       (lambda (rows)
                         (cond ((null? rows) '())
                               ((= count (car rows))
                                (cons (car rows) (rel-rows (cdr rows))))
                               (else (rel-rows (cdr rows)))))))
               (if (null? (cdar T))
```

```
                    (rem-tables (cdr tables) i count)
                    (cons
                     (list (cdar T) (map cdr (cadr T)))
                     (rem-tables (cdr tables) i count))))))
            (else (cons (car tables) (rem-tables (cdr tables) i count))))))

;; table-at? is true if there is a table with first column labeled i+1
;; and a row with count in that column.

(define table-at?
  (lambda (tables i count)
    (ormap (lambda (table)
             (and (= (caar table) (add1 i))
                  (ormap (lambda (row)
                           (= (car row) count))
                         (cadr table))))
           tables)))

;; table-max? is true if there is a table with first column labeled
;; i+1 where count is the maximum value in that column.

(define table-max?
  (lambda (tables i count)
    (ormap (lambda (table)
             (and (= (caar table) (add1 i))
                  (andmap (lambda (row)
                            (<= (car row) count))
                          (cadr table))))
           tables)))

;; GenConstr generates the constraints relative to a particular time
;; point.
;;
;; i is the current time point
;; max is the maximum time point
;; count indicates how many indices since matched i
;; tables contain constraints remaining to be processed

(define GenConstr
  (lambda (i max count tables)
    (if (table-at? tables i count)
       (if (table-max? tables i count)
          `(AX(,@(GenCTL (add1 i) max (rem-tables tables i count))))
          `(AX(,@(GenCTL (add1 i) max (rem-tables tables i count))
               + (,@(SigPat i) * !,@(GenObj (add1 i)) *
                          ,@(GenConstr i max (add1 count) tables)))))
       `(AX(,@(SigPat i) * !,@(GenObj (add1 i)) *
                    ,@(GenConstr i max (add1 count) tables))))))

;; GenCTL is the main translation function. The translation depends
```

```
;; on whether the last time point is being processed, whether there
;; are any time bounds relating the current time point to the next
;; time point, and whether there are any signal patterns that must be
;; respected.
;;
;; i is the current time point
;; max is the maximum time point
;; tables contains the constraints remaining to be processed

(define GenCTL
  (lambda (i max tables)
    (cond ((> i max) 'true)
          ((= i max) `(,@(GenObj i) * ,@(SigPat(sub1 i))))
          ((Constraint? i tables) `(,@(GenObj i) * ,@(SigPat(sub1 i)) *
                                         ,@(GenConstr i max 1 tables)))
          ((eq? (SigPat(sub1 i)) 'true)
           `(,@(GenObj i) * AX AF(,@(GenCTL2 (add1 i) max tables))))
          (else
           `(,@(GenObj i) * ,@(SigPat (sub1 i)) *
                 AXA[(,@(SigPat (sub1 i)) * !,@(GenObj (add1 i)))
                      U ,@(GenCTL (add1 i) max tables)])))))

;; The format for a table is '(header . contents) where header is a
;; list of the time points being related and contents is a list of the
;; relative separations between each point. Example tables follow

(define table1
  '((2)
   ((3) (4) (5))))

(define table2
  '((3 4)
   ((1 2) (1 3) (1 4) (2 2) (2 3))))

(define table3
  '((5 6 7 8)
   ((1 1 2 3) (1 2 2 3) (2 1 1 2) (2 1 1 3) (2 1 2 1) (2 2 1 1)
          (4 3 2 1) (4 2 3 1) (4 2 1 1))))
```

B

COSPAN Verification of Elevator Controller

B.1 An Overview of COSPAN

COSPAN is a UNIX^{TM}-based tool for *coordination-specification analysis*. Designs are presented to COSPAN as a collection of processes, or specifications. The set of processes must represent a closed system — every input to a process must be the output of some other process in the collection. Desired properties, or *tasks*, are expressed as a form of ω-automata. COSPAN checks the design against the property using language-containment testing algorithms. If the containment fails, COSPAN produces an error track that identifies a particular word that violates the containment test. COSPAN supports several techniques for approaching state-explosion, though none were needed in the verification of the elevator controller.

The elevator controller design is presented in Section B.2; the elevator tasks appear in Section B.3. The remainder of this overview presents sufficient information about the syntax of S/R, the COSPAN input language, for the interested reader to study these code fragments.

The *process* is the central structure in a COSPAN specification. Each process, defined using the keyword **proc**, contains the following elements:

- An **import** statement, indicating the other processes from which the current process takes information. The process can access the selection variables (defined below) of any process it **imports**. If process A must access the state variables of another process, A should be defined using keyword **monitor** as opposed to keyword **proc**. Aside from these scoping rules, **proc** and **monitor** definitions are identical.

- Declaration of the process' state variables (keyword **stvar**). Each state variable is given a type, which must contain only a finite number of values. Examples of valid types for state variables include boolean, finite integer ranges, and user-defined enumerated types. All state variables must be initialized using an **init** statement; the initialization value may be either a single state or a range of states of the appropriate type. Conventionally, the name $ is used for the main state variable in a process.

- Declaration of the process' selection variables (keyword **selvar**). Selection variables are similar in spirit to outputs in conventional state machines. Each transition in a process enables certain selections. COSPAN nondeterministically chooses from the enabled selections, then resolves the next state of the system based on those choices. Selection variables may be used as conventional state machine outputs by assigning their values based on the current state of the process. Conventionally, the name # is used for the main selection variable in a process.

- The transition structure definition. COSPAN provides two mechanisms for defining state transitions: **trans** and **asgn**. The elevator design uses only the latter because the state transitions are entirely deterministic. The **trans** method is useful for defining systems with non-deterministic length pauses in the current state; more details can be found in the user manual supplied with the COSPAN system. The **asgn** statement uses two different assignment operators: "**asgn** := " makes the assignment in the current state and "**asgn**→" makes the assignment in the next state. The former is used for selection variables and local variables and the latter for state variables.

  The expressions within an **asgn** statement use operators $*$, $+$, and $\sim$ for conjunction, disjunction, and negation, respectively. Mathematical expressions use their standard notations. Expression $a?b|c$ represents "if a then b else c". All such expressions must contain an "else" clause. The keyword **else** is defined to be the negation of all previous cases in a compound "if" statement.

One other form of expression used in the elevator example is conjunction and disjunction over vector types. Given variable Var defined as a vector of n booleans (where n is a fixed, concrete number)

Var[n]: **boolean**

the disjunction of each Var[i] is expressed as

+[i **in** 0..n-1] Var[i]

COSPAN allows types of processes to be defined using keyword **proctype**. A **proctype** has the same body as a **proc**, with the addition of parameterized variables immediately following the word **proctype**. A **proctype** is instantiated as a **proc** using syntax

proc *procname*: *proctypename* (*parameters*)

Two other COSPAN statements appear in the following code. COSPAN processes are a form of ω-automata. As a result, processes must contain cycles, since they are finite-state systems by definition. Algorithmically, COSPAN operates by looking for cycles in the design that violate the given property. Not all cycles in a design reflect errors, however. For example, the elevator controller cycles in the same state if nobody ever requests service from the elevator. COSPAN provides two mechanisms for defining excepting behavior with regard to cycles: **cyset** and **recur**. A **cyset** indicates sets of states in which the system may cycle without causing an error; COSPAN ignores cycles in these sets during analysis. A **recur** edge is a transition in the system that may be crossed infinitely many times without creating a cycle in the system; COSPAN removes all **recur** edges from the graph before beginning analysis. These constructs must be used carefully — it is possible to mask real errors in the system if they coincide with cycles excepted using these constructs.

The tasks file makes use of the C pre-processor command **#ifdef**. This command allows all of the tasks to be written into a single file, but to be tested individually using command-line flags to COSPAN. For example, to test that the doors are never open at more than one floor simultaneously (**monitor** sim_open in the tasks file), COSPAN would be run with additional flag "-DSimOpen" (where "-D" means "define"). Each task in the file was tested individually, and all verified of the controller.

The tasks file also makes use of two template proctypes provided with COSPAN: STARVATION(hungry, fed) and STOP(condition). The former is satisfied if whenever the hungry parameter is true, eventually the fed parameter is true. The latter is satisfied if condition is never true of the system. The **proctype** definitions of both appear below. The STOP definition works because COSPAN generates an error if a state is ever reached from which there is no possible next-state transition.

```
proctype STARVATION( h, f : boolean )
    import h, f
    stvar $ : (-1, 0)
    recur $: *→0
    init $ := 0
    asgn $ → -1 ? ($=0)*h | 0 ? f | $
end STARVATION()
```

```
proctype STOP( c : boolean )
    import c
    stvar $ : (1)
    init 1
    trans true →1 : ~c;
end STOP()
```

B.2 The Controller Code

```
/* Kathi Fisler */
/* COSPAN model of the Elevator Example */

macro num_floors := 3,
      min_floor := 1,
      max_floor := num_floors,
      array_size := num_floors + 2,
      Door_Count := 20,
      Clear_Count := 10

type timer_value : (0..500)
type floor_num : (min_floor..max_floor)


#include <sr/1.h>
```

```
/* Timer proctype */
proctype Timer (count: timer_value; start: boolean)
    import count, start
    stvar $ : timer_value
    selvar # := $
    init $ := 0
    asgn $ → count ? start |
                $ - 1 ? ~start*($ > 0) |
                    0
end Timer()


/* The Elevator Module */
proc Elev
     import Elev_Env, Reqp
     stvar moving_st: (mup, mdown, off)
     selvar moving_up : (0, 1)
     selvar moving_dn : (0, 1)
     selvar not_moving : (0, 1)
     selvar sched_sel := Scheduler.#
     selvar open[num_floors+1]: boolean
     selvar close[num_floors+1]: boolean
     selvar alarm_active: boolean
     init moving_st := off
     asgn moving_st → mup     ? (Current.up: 1) |
                      mdown   ? (Current.dn: 1) |
                      off     ? (Reqp.#: 0) |
                      moving_st
     asgn moving_up := 1 ? (moving_st: mup) | 0,
          moving_dn := 1 ? (moving_st: mdown) | 0,
          not_moving := 1 ? (moving_st: off) | 0
     asgn alarm_active := +[i in min_floor..max_floor] Floors[i].alarm_on,
          [i in min_floor..max_floor]{open[i] := Floors[i].open_n,
                                      close[i] := Floors[i].close_n}
     asgn open[0] := false, close[0] := false

     /* Floor Manager */
     proctype Floor (n : floor_num)
          import n, DT, CT, Scheduler, Elev_Env, moving_up, moving_dn, not_moving
          stvar $ : (idle, door, clear, alarm, delay)
          selvar open_n, start_dt, start_ct, close_n, alarm_on : boolean
          init $ := idle
          macro same_dir := ((Elev_Env.req_up[n]: 1)*(moving_up: 1)) +
                            ((Elev_Env.req_dn[n]: 1)*(moving_dn: 1))
          asgn $ → idle  ? ($ = idle)*~((Elev_Env.at[n]: 1)*(Scheduler.# = n)*
                           ((Elev_Env.req[n]: 1)+same_dir+
                           ((not_moving: 1)*((Elev_Env.req_up[n]: 1)+
                           (Elev_Env.req_dn[n]: 1))))) +
                           ($ = delay) |
```

```
                  delay ? ($ = door)*(DT.# = 0)*(Elev_Env.door_clear: 1) +
                          ($ = clear)*(Elev_Env.door_clear: 1) +
                          ($ = alarm)*(Elev_Env.door_clear: 1) |
                  door  ? (($ = idle)*(Elev_Env.at[n]: 1)*(Scheduler.# = n)*
                          ((Elev_Env.req[n]: 1)+same_dir+
                          ((not_moving: 1)*((Elev_Env.req_up[n]: 1)+
                          (Elev_Env.req_dn[n]: 1))))) + (($ = door)*(DT.# > 0)) |
                  clear ? ($ = door)*(DT.# = 0)*(Elev_Env.door_clear: 0) +
                          ($ = clear)*~(Elev_Env.door_clear: 1)*(CT.# > 0) |
                  alarm
        asgn open_n := ($ = idle)*((Elev_Env.at[n]: 1)*(Scheduler.# = n)*
                       ((Elev_Env.req[n]: 1)+same_dir)),
             start_dt := ($ = idle)*((Elev_Env.at[n]: 1)*(Scheduler.# = n)*
                       ((Elev_Env.req[n]: 1)+same_dir)),
             start_ct := ($ = door)*(DT.# = 0)*(Elev_Env.door_clear: 0),
             close_n := ($ = door)*(DT.# = 0)*(Elev_Env.door_clear: 1) +
                       ($ = clear)*(Elev_Env.door_clear: 1) +
                       ($ = alarm)*(Elev_Env.door_clear: 1),
             alarm_on := ($ = alarm)
    end Floor()

    /* Scheduler */
    proc Scheduler
        import Elev_Env, moving_up, moving_dn, not_moving, open[]
        stvar $ : floor_num
        selvar # := $
        selvar higher_req[array_size], lower_req[array_size] : (0, 1)
        recur 1→2, 2→3
        init $ := min_floor
        asgn [i in min_floor..max_floor] {higher_req[i] := 1 ?
                                  +[j in i..max_floor] ((Elev_Env.req_up[j]: 1) +
                                  (Elev_Env.req_dn[j]: 1) + (Elev_Env.req[j]: 1)) | 0}
        asgn [i in min_floor..max_floor] {lower_req[i] := 1 ?
                                  +[j in min_floor..i] ((Elev_Env.req_up[j]: 1) +
                                  (Elev_Env.req_dn[j]: 1) + (Elev_Env.req[j]: 1)) | 0}
        asgn higher_req[0] := 0, higher_req[array_size-1]:= 0,
             lower_req[0] := 0, lower_req[array_size-1]:= 0
        asgn $ → $ + 1 ? (($ < max_floor)*(Elev_Env.at[$]: 1)*
                          (Elev_Env.door_open: 0)*(Elev.open[$]: 0)*
                          ((moving_up: 1)*(higher_req[$+1]: 1) +
                          (moving_dn: 1)*(lower_req[$-1]: 0)*(higher_req[$+1]: 1) +
                          (not_moving: 1)*(higher_req[$+1]: 1))) |
                 $ - 1  ? (($ > min_floor)*(Elev_Env.at[$]: 1)*
                          (Elev_Env.door_open: 0)*(Elev.open[$]: 0)*
                          ((moving_dn: 1)*(lower_req[$-1]: 1) +
                          (moving_up: 1)*(higher_req[$+1]: 0)*(lower_req[$-1]: 1) +
                          (not_moving:1)*(lower_req[$-1]: 1))) |
                 $
```

```
        end Scheduler

        proc Current
            import Scheduler, Elev_Env
            stvar $ : floor_num
            selvar # := $
            selvar up, dn : boolean
            init $ := min_floor
            asgn $ → $ + 1 ? (Scheduler.# > $)*(Elev_Env.at[$+1]: 1) |
                     $ - 1  ? (Scheduler.# < $)*(Elev_Env.at[$-1]: 1) |
                     $
            asgn up := (Scheduler.# > $)*(Elev_Env.at[$+1]: 0),
                 dn := (Scheduler.# < $)*(Elev_Env.at[$-1]: 0)
        end Current

        proc Floors[i < max_floor+1]: Floor(i)
        proc DT: Timer(Door_Count, +[i in min_floor..max_floor](Floors[i].start_dt: 1))
        proc CT: Timer(Clear_Count, +[i in min_floor..max_floor](Floors[i].start_ct: 1))
end Elev

proc Reqp
        import Elev_Env
        stvar $ : (0, 1)
        selvar # := $
        cyset $=0
        init $ := 0
        asgn $ → 0 ? *[i in min_floor..max_floor]((Elev_Env.req_up[i]: 0)*(Elev_Env.req_dn[i]: 0)*
                                                   (Elev_Env.req[i]: 0)) | 1
end Reqp

proc Elev_Env
        import Elev
        macro n := num_floors+1
        stvar req_up_st[n], req_dn_st[n], req_st[n], at_st[array_size],door_open_st : (0, 1)
        selvar req_up[n], req_dn[n], req[n], at[array_size], door_open,door_clear:(0,1)
        init [i in 0..n-1]{req_up_st[i] := 0, req_dn_st[i] := 0, req_st[i] := 0},
             at_st[min_floor] := 1,
             at_st[0] := 0,
             [i in min_floor+1..array_size-1]{at_st[i] := 0}, door_open_st := 0
        asgn [i in 0..n-1]{req_up[i] := req_up_st[i],
                           req_dn[i] := req_dn_st[i],
                           req[i] := req_st[i]},
              [i in 0..array_size-1]{at[i] := at_st[i]},
              door_open := door_open_st
        asgn [i in min_floor..max_floor]{req_up_st[i] → {0, 1} ? (req_up_st[i]: 0) |
                                                        0      ? (req_up_st[i]:1)*(Elev.moving_up: 1)*
                                                                 (at[i]: 1)*(Elev.sched_sel: i) |
                                                        1},
```

```
        [i in min_floor..max_floor]{req_dn_st[i] → {0, 1} ? (req_dn_st[i]: 0) |
                                                  0      ? (req_dn_st[i]:1)*(Elev.moving_dn: 1)*
                                                           (at[i]: 1)*(Elev.sched_sel: i) |
                                                  1},
        [i in min_floor..max_floor]{req_st[i] → {0, 1} ? (req_st[i]: 0) |
                                               0      ? (req_st[i]: 1)*(at[i]: 1)*
                                                        (Elev.sched_sel: i) |
                                               1},
        [i in min_floor..max_floor]{at_st[i] → {0, 1} ? *[j in min_floor..max_floor](at_st[j]: 0)*
                                                       (Elev.sched_sel: i) |
                                              0      ? (at_st[i]: 1)*~(Elev.sched_sel: i) |
                                              1      ? (at_st[i]: 1)*(Elev.sched_sel: i) |
                                              0},
        door_open_st → 0 ? (door_open_st: 1)*
                           +[i in min_floor..max_floor]Elev.close[i] |
                       1 ? +[i in min_floor..max_floor]Elev.open[i] +
                           ((door_open_st: 1)*
                            *[i in min_floor..max_floor]~Elev.close[i]) |
                       0,
        door_clear := {0, 1} ? (door_open: 1) | 1,
        req_up_st[0] → 0, req_dn_st[0] → 0, at_st[0] → 0,
        req_st[0] → 0, at_st[array_size-1] → 0
end Elev_Env

proc Constraint1[i < max_floor+1]:
        SF(1, 1 ? (i = 0) | ~(Elev.sched_sel: i) + (Elev_Env.at[i]: 1)*(Elev.sched_sel: i))

proc Constraint2: SF(1, (Elev_Env.door_clear: 1))

proc Constraint3: KILL((Elev_Env.req_up[max_floor]: 1) + (Elev_Env.req_dn[min_floor]: 1))

#include "myelev_tasksdiss.sr"
```

B.3 The Tasks

```
/* Kathi Fisler */
/* Tasks for the Elevator Example */

monitor startctcond
    import Elev.Floors[]
    stvar $ : (0, 1, -1)
    selvar # := $
    init $ := 0
    asgn $ →  1 ? ~Elev.Floors[1].start_ct *
                  (Elev.Floors[2].start_ct + Elev.Floors[3].start_ct) *
                  (($ = 0) + ($ = 1)) |
```

```
                -1 ? ($ = 1) * Elev.Floors[1].start_ct |
                 0 ? ($ = 1) * (Elev.Floors[1].start_ct +
                       (~Elev.Floors[2].start_ct * ~Elev.Floors[3].start_ct)) |
                $
end startctcond

monitor cttest: STOP((startctcond.#: -1))


monitor startdtcond
    import Elev.Floors[]
    stvar $ : (0, 1, -1)
    selvar # := $
    init $ := 0
    asgn $ → 1 ? ~Elev.Floors[1].start_dt *
                       (Elev.Floors[2].start_dt + Elev.Floors[3].start_dt) *
                       (($ = 0) + ($ = 1)) |
                -1 ? ($ = 1) * Elev.Floors[1].start_dt |
                 0 ? ($ = 1) * (Elev.Floors[1].start_dt +
                       (~Elev.Floors[2].start_dt * ~Elev.Floors[3].start_dt)) |
                $
end startdtcond

monitor dttest: STOP((startdtcond.#: -1))


#ifdef SimOpen

monitor sim_open:
    STARVATION((Elev.open[1]*Elev.open[2]) + (Elev.open[2]*Elev.open[3]) +
                (Elev.open[1]*Elev.open[3]), false)

#else

#ifdef DoorsMove

monitor closemove:
    STARVATION((Elev.Current.up: 1)*(Elev_Env.door_open: 1) +
                (Elev.Current.dn: 1)*(Elev_Env.door_open: 1), false)

#else

#ifdef DoorsAt

monitor doors_at_floors:
    STARVATION((Elev.open[1]*(Elev_Env.at[1]: 0)) + (Elev.open[2]*(Elev_Env.at[2]: 0)) +
                (Elev.open[3]*(Elev_Env.at[3]: 0)), false)

#else
```

```
#ifdef ReqAns

#ifdef f1

monitor req_ans1:
      STARVATION((Elev_Env.req[1]: 1)+(Elev_Env.req_up[1]: 1)+
                  (Elev_Env.req_dn[1]: 1), (Elev_Env.at[1]: 1))
#else
#ifdef f2

monitor req_ans2:
      STARVATION((Elev_Env.req[2]: 1)+(Elev_Env.req_up[2]: 1)+
                  (Elev_Env.req_dn[2]: 1), (Elev_Env.at[2]: 1))
#else

monitor req_ans3:
      STARVATION((Elev_Env.req[3]: 1)+(Elev_Env.req_up[3]: 1)+
                  (Elev_Env.req_dn[3]: 1), (Elev_Env.at[3]: 1))

#endif
#endif

#else

#ifdef ReqToOpen

#ifdef f1
monitor open1:
      STARVATION(Elev.open[1]*~((Elev_Env.req[1]: 1) + (Elev_Env.req_up[1]: 1) +
                  (Elev_Env.req_dn[1]: 1)), false)
#endif

#ifdef f2
monitor open2:
      STARVATION(Elev.open[2]*~((Elev_Env.req[2]: 1) + (Elev_Env.req_up[2]: 1) +
                  (Elev_Env.req_dn[2]: 1)), false)
#endif

#ifdef f3
monitor open3:
      STARVATION(Elev.open[3]*~((Elev_Env.req[3]: 1) + (Elev_Env.req_up[3]: 1) +
                  (Elev_Env.req_dn[3]: 1)), false)
#endif
#else


#ifdef DirChange
```

```
monitor highest_up_req
    import Elev_Env
    stvar $ : (min_floor..max_floor)
    selvar # := $
    init $ := min_floor
    asgn $ → 3 ? (Elev_Env.req[3]: 1) |
             2 ? else*((Elev_Env.req[2]: 1)+(Elev_Env.req_up[2]: 1)) |
             1
end

monitor lowest_dn_req
    import Elev_Env
    stvar $ : (min_floor..max_floor)
    selvar # := $
    init $ := max_floor
    asgn $ → 1 ? (Elev_Env.req[1]: 1) |
             2 ? else*((Elev_Env.req[2]: 1)+(Elev_Env.req_dn[2]: 1)) |
             3
end


#ifdef Up

monitor catch_max_up
    import Elev_Env, Elev, highest_up_req
    stvar $ : (0, 1, -1)
    cyset $=0
    recur 0 → 1
    init $ := 0
    asgn $ →  1 ? ($ = 0)*(Elev.Current.$ < highest_up_req.$)*(Elev.moving_up: 1) |
              0 ? ($ = 1)*(Elev.Current.$ = highest_up_req.$) |
             -1 ? ($ = 1)*(Elev.moving_up: 0)*(Elev.Current.$ < highest_up_req.$) |
              $
end

#else

monitor catch_min_down
    import Elev_Env, Elev, lowest_dn_req
    stvar $ : (0, 1, -1)
    cyset $=0
    recur 0 → 1
    init $ := 0
    asgn $ →  1 ? ($ = 0)*(Elev.Current.$ > lowest_dn_req.$)*(Elev.moving_dn: 1) |
              0 ? ($ = 1)*(Elev.Current.$ = lowest_dn_req.$) |
             -1 ? ($ = 1)*(Elev.moving_dn: 0)*(Elev.Current.$ > lowest_dn_req.$) |
              $
end
```

```
#endif

#else
#ifdef CatchAll
#ifdef up2

monitor catch_all_up
    import Elev, Elev_Env
    stvar $ : (0, 1, -1)
    cyset $=0
    recur 1 → 0
    init $ := 0
    asgn $ →  1 ? ($ = 0)*(Elev.Current.$ < 2)*~(Elev.open[2]: 1)*((Elev_Env.req[2]: 1)+
                  (Elev_Env.req_up[2]: 1)) |
              0 ? ($ = 1)*(Elev.open[2]: 1) |
             -1 ? ($ = 1)*(Elev.Current.$ > 2) |
              $
end

#endif

#ifdef dn2

monitor catch_all_dn
    import Elev, Elev_Env
    stvar $ : (0, 1, -1)
    cyset $=0
    recur 1 → 0
    init $ := 0
    asgn $ →  1 ? ($ = 0)*(Elev.Current.$ > 2)*~(Elev.open[2]: 1)*((Elev_Env.req[2]: 1)+
                  (Elev_Env.req_dn[2]: 1)) |
              0 ? ($ = 1)*(Elev.open[2]: 1) |
             -1 ? ($ = 1)*(Elev.Current.$ < 2) |
              $
end

#endif

#else
#ifdef DoorClose
#ifdef f1

monitor close1: STARVATION((Elev.open[1]: 1), (Elev_Env.door_open: 0))

#else
#ifdef f2

monitor close2: STARVATION((Elev.open[2]: 1), (Elev_Env.door_open: 0))
```

#else

monitor close3: STARVATION((Elev.open[3]: 1), (Elev_Env.door_open: 0))

#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif

Bibliography

[1] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. In *Proceedings of TACAS96, 2nd Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–48, March 1996.

[2] Rajeev Alur. *Techniques for Automated Verification of Real-time Systems*. PhD thesis, Stanford University, 1991.

[3] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 592–601, 1993.

[4] A. Aziz et al. HSIS: A BDD-based environment for formal verification. In *Proceedings of the Design Automation Conference*, 1994.

[5] Dave Barker-Plummer and Sidney C. Bailin. Proofs and pictures: Proving the Diamond Lemma with the GROVER theorem proving system. In *AAAI Symposium on Reasoning with Diagrammatic Representations*, November 1995.

[6] Dave Barker-Plummer, Sidney C. Bailin, and Samuel M. Ehrlichman. Diagrams and mathematics. Unpublished manuscript, Center for the Study of Language and Information (CSLI), Stanford University, November 1995.

[7] Jon Barwise and John Etchemendy. *Hyperproof*. CSLI Lecture Notes. University of Chicago Press, 1994.

[8] Jon Barwise and John Etchemendy. Heterogeneous logic. In Janice Glasgow, N. Hari Narayanan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, pages 211–234. MIT Press, 1995.

[9] Jon Barwise and John Etchemendy. Logic, proof, and reasoning. In Alan Makinowski, editor, *Companion to Logic*. Blackwell, To appear.

[10] Bachi Berkane, Simona Gandrabur, and Eduard Cerny. Timing diagrams:semantics and timing analysis. LASSO Laboratory, University of Montreal, 1996.

[11] Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. In *Proceedings of Computer Aided Verification (CAV)*. Springer-Verlag, June 1994. Lecture Notes in Computer Science 818.

[12] Bhaskar Bose, Steven D. Johnson, and Shyam Pullela. Integrating boolean verification with formal derivation. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 127–134. Elsevier, April 1993. Also published as Technical Report No. 372, Dept. of Computer Science, Indiana University.

[13] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[14] Robert S. Boyer and J. Strother Moore. Proof-checking, theorem-proving and program verification. Technical report, Institute for Computing Science and Computer Applications, University of Texas at Austin, January 1983.

[15] R.K. Brayton et al. VIS: A system for verification and synthesis. Technical Report UCB/ERL M95, Electronic Research Lab, University of California at Berkeley, 1995.

[16] I.A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, December 1995. Lecture Notes in Computer Science.

[17] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[18] J. Burch, E.M. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, June 1990.

[19] Maura Cerioli and José Meseguer. May I borrow your logic? In *Proceedings of MFCS, Lecture Notes in Computer Science*, 1993.

[20] E. Cerny and K. Khordoc. Interface specifications with conjunctive timing constraints: Realizability and compatibility. In *Second AMAST Workshop on Real-Time Systems*, June 1995.

[21] Soma Chaudhuri and Richard E. Ladner. Safety and liveness of ω-context-free languages. *Information Processing Letters*, 37:13–20, 1991.

[22] Tam-Anh Chu. *Synthesis of Self-timed VLSI circuits from graph-theoretic specfications.* PhD thesis, MIT, June 1987.

[23] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, pages 52–71. Springer-Verlag, 1981. Lecture Notes in Computer Science 131.

[24] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, 1989.

[25] National Semiconductor Corporation. LS/S/TTL logic databook, 1987.

[26] D. Van Dalen. *Logic and Structure.* Springer-Verlag, 1989.

[27] Ruth Eberle. *Diagrams and Natural Deduction: Theory and Pedagogy of Hyperproof.* PhD thesis, Indiana University, 1995. Department of Philosophy.

[28] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier Science Publishers, 1990.

[29] E.A. Emerson and J.Y. Halpern. "Sometimes" and "Not Never" revisted: Branching time strikes back. *Journal of the ACM*, 33(1):151–178, 1986.

[30] Werner Grass et. al. Transformation of timing diagram specifications into VHDL code. In *Proceedings of Computer Hardware Description Languages and Their Applications*, pages 659–668, August 1995.

[31] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design — interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *Formal VLSI Specification and Synthesis: VLSI Design-Methods-I*. North-Holland, 1990.

[32] Kathi Fisler. A canonical form for circuit diagrams. Technical Report TR432, Indiana University, May 1995.

[33] Seymour Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, 1966.

[34] Janice Glasgow, N. Hari Narayanan, and B. Chandrasekaran. *Diagrammatic Reasoning: Cognitive and Computational Perspectives*. MIT Press, 1995.

[35] M.J.C. Gordon. Why higher order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on VLSI*, pages 153–177. North Holland, 1986.

[36] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Syntesis*, pages 73–128. Kluwer Academic Publishers, 1987.

[37] O. Grumberg and D.E. Long. Model checking and modular verification. In *Proceedings of CONCUR '91: Second International Conference on Concurrency Theory*. Springer-Verlag, 1991. Lecture Notes in Computer Science, volume 527.

[38] Aarti Gupta. Formal hardware verification methods: A survey. In Robert Kurshan, editor, *Computer-Aided Verification*. Kluwer Academic Publishers, 1992. Reprinted from Formal Methods in System Design, Vol. 1, Nos. 2–3.

[39] John V. Guttag, James J. Horning with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[40] P. Halmos. *Lectures on Boolean Algebras.* Springer-Verlag, 1974.

[41] J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In *Proceedings of the Tenth International Colloquium on Automata, Language, and Programming*, pages 278–291. Springer-Verlag, 1983.

[42] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science, Ithaca NY*, pages 54–64, 1987.

[43] David Harel. On visual formalisms. Technical Report CMU–CS–87–126, Carnegie-Mellon University, June 1987.

[44] David Harel and Danny Raz. Deciding properties of nonregular programs. *SIAM Journal of Computing*, 22(4):857–874, August 1993.

[45] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[46] W.A. Hunt Jr. *FM 8501: A Verified Microprocessor.* PhD thesis, University of Texas at Austin, 1985.

[47] Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1):116–133, January 1978.

[48] Oscar H. Ibarra, Tao Jiang, Nicholas Tran, and Hui Wang. New decidability results concerning two-way counter machines and applications. In *Proceedings of the 20th International Colloquium on Automata, Languages, and Programming (ICALP)*, 1993. Lecture Notes in Computer Science, 700.

[49] International Telecommunication Union (ITU). Message sequence chart (MSC). ITU-T Recommendation Z.120, 1993.

[50] Geraint Jones and Mary Sheeran. Relations and refinement in circuit design. In Carroll Morgan and Jim Woodcock, editors, *Proceedings of the 3rd Refinement Workshop.* Springer-Verlag, 1991.

[51] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in ruby. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*. Springer-Verlag, 1993. Lecture Notes in Computer Science 669.

[52] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine, and A. Silburt. Integrating behavior and timing in executable specifications. In *CHDL*, pages 385–402, April 1993.

[53] Karim Khordoc and Eduard Cerny. Modeling cell processing hardware with action diagrams. In *International Symposium on Circuits and Systems*, 1994.

[54] R. Kumar, C. Blumenrohr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design — a survey. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 1996.

[55] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[56] R.P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In *Proceedings of Computer-Aided Verification (CAV)*, pages 166–179, 1993.

[57] Miriam E. Leeser. Using Nuprl for the verification and synthesis of hardware. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*. Prentice-Hall, 1992.

[58] Isabel Luengo. *Diagrams in Geometry*. PhD thesis, Indiana University, 1995. Department of Philosophy.

[59] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1982.

[60] Zohar Manna et al. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, 1994.

[61] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[62] Thomas Frederick Melham. Formalizing abstraction mechanisms for hardware verification in higher order logic. Technical Report TR 201, University of Cambridge Computer Laboratory, August 1990.

[63] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit: A case study exploring the boundary between formal reasoning systems. In Satnam Singh, Mary Sheeran, and Geraint Jones, editors, *Third Workshop on Designing Correct Circuits*, 1996.

[64] Cho Woo Moon. *Synthesis and Verification of Asynchronous Circuits from Graphical Specifications.* PhD thesis, University of California at Berkeley, 1992.

[65] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[66] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS:combining specification, proof checking, and model checking. To be presented at CAV '96, July 1996.

[67] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, 1986.

[68] Amir Pnueli. The temporal logic of programs. In *Proceedings of the Eighth Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[69] Dag Prawitz. *Natural Deduction: A Proof-Theoretic Study.* Almqvist & Wiksells, 1965.

[70] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design.* Prentice-Hall, 2nd edition, 1987.

[71] Y.S. Ramakrishna, L.K. Dillon, L.E. Moser, P.M. Melliar-Smith, and G. Kutty. A real-time interval logic and its decision procedure. In *Proceedings of the Thirteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 173–192, December 1993. Springer-Verlag Lecture Notes in Computer Science volume 761.

[72] John Rushby. Formal methods and digital systems validation for airborne systems. Contractor Report 4551, NASA, December 1993.

[73] Hossein Saiedian. An invitation to formal methods. *IEEE Computer*, 29(4):16–30, April 1996.

[74] Rainer Schlör. A prover for VHDL-based hardware design. In *Proceedings of the Conference on Hardware Description Languages and Their Applications*, August 1995.

[75] Rainer Schlör and Werner Damm. Specification and verification of system-level hardware designs using timing diagrams. In *Proc. European Conf. on Design and Automation*, Paris, February 1993.

[76] Sun-Joo Shin. *The Logical Status of Diagrams.* Cambridge University Press, 1994.

[77] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.

[78] V. Singhal and C. Pixley. The verification problem for safe replaceability. In D. Dill, editor, *Proc. 6th International Conference on Computer Aided Verification, Springer-Verlag Lecture Notes in Computer Science vol. 818*, pages 311–323, June 1994.

[79] SRI International, Formal Methods Group. The PVS2 verification system. Information available at `http://www.csl.sri.com/pvs.html`.

[80] Stenning, R. K. Cox, and J. Oberlander. Contrasting the cognitive effects of graphical and sentential logic teaching: reasoning, representation and individual differences. *Language and Cognitive Processes*, 10(3/4):333–354, 1995.

[81] Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity.* Birkhauser, 1994.

[82] Bell Laboratories/Lucent Technologies. Cospan verification system.

[83] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 133–191. Elsevier Science Publishers, 1990.

[84] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic and Computer Science (LICS)*, 1986.

[85] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1), November 1994.

Index

Kathryn Fisler

Curriculum Vita

Education

Ph.D. in Computer Science, Indiana University, August 1996.

M.S. in Computer Science, Indiana University, December 1992

B.A. in Computer Science and Asian Studies, cum laude with highest honors in Computer Science, Williams College, June 1991

Experience

Summer 1996	Member of Technical Staff, Lucent Technologies/Bell Laboratories.
Summer 1995	Member of Technical Staff, AT&T Bell Laboratories.
1994-1996	Graduate Student Research Fellow, AT&T Bell Laboratories.
1993-1994	Research Assistant, Visual Inference Lab, Department of Computer Science, Indiana University.
1991-1994	Summer Instructor, Department of Computer Science, Indiana University.
1991-1993	Associate Instructor, Department of Computer Science, Indiana University.

Awards and Honors

1996	Awarded National Science Foundation Postdoctoral Research Fellowship in Experimental Computer Science
1994-1996	AT&T Graduate Student Research Fellow
1994	Selected as NASA Langley Graduate Student Research Fellow (declined)
1993	NDSEG Graduate Fellowship Honorable Mention
1992	Indiana University Summer Research Incentive Fellowship

1992	Nominee for Lieber Outstanding Associate Instructor Award, Indiana University
1991, 1992	NSF Fellowship Honorable Mention
1991	Goldberg prize for Best Student Colloquium Speaker, Williams College

Publications

Fisler, Kathi. *Visualizing System Language Relationships with Logic.* Presented at the CADE 96 workshop on Visual Reasoning, July 1996.

Barwise, Jon and Ruth Eberle and Kathi Fisler. *Teaching Reasoning Using Heterogeneous Logic.* Presented at DIMACS workshop on Teaching Logic in an Illogical World, July 1996.

Fisler, Kathi. *Exploiting the Potential of Diagrams in Guiding Hardware Reasoning.* In Logical Reasoning with Diagrams, Gerard Allwein and Jon Barwise, eds. Oxford University Press, 1996.

Fisler, Kathi. *Using COSPAN to Partially Verify and Debug a Barcode Reader.* AT&T Bell Laboratories Technical Memorandum, December 1995.

Fisler, Kathi and Steven D. Johnson. *Integrating Design and Verification Environments Through a Logic Supporting Hardware Diagrams.* In Proceedings of Computer Hardware Descriptions Languages and Their Applications, August 1995.

Fisler, Kathi. *Extending Formal Reasoning with Support for Hardware Diagrams.* In Proceedings of the Second Conference on Theorem Provers in Circuit Design (Ramayya Kumar and Thomas

Kropf, eds.), September 1994.

Fisler, Kathi *A Canonical Form for Circuit Diagrams.* Indiana University Department of Computer Science Technical Report TR-432, April 1995.

Fisler, Kathi. *A Logical Formalization of Hardware Design Diagrams.* Indiana University Department of Computer Science Technical Report TR-416, September 1994.

Fisler, Kathi. *A Diagrammatic Logic for Reasoning About Asynchronous Hardware.* In "Working Papers on Diagrams and Logic" (Jon Barwise and Gerard Allwein, eds.), Indiana University Logic Group Preprint IULG-93-23, 1993.

Talks and Presentations

The Logic of Hardware Design Diagrams. University of Cincinnati, May 1996.

Advancing Practical Hardware Verification Through Diagrammatic Logic. Rice University, November 1995.

Diagrammatic Reasoning and its Application to Hardware Verification. Williams College, September 1994.

Formalizing Hardware Diagrams for Reasoning and Verification. Indiana University Applied Logic Lunch, February 1994.